Programming and Data Structures

Chittaranjan Mandal

Dept of Computer Sc & Engg
IIT Kharagpur

November 9, 2011

Table of Parts I

Table of Parts II

Part I

Introduction

Section outline

1. **Outline**
   - Resources
   - Course objectives

Resources

Web site `http://cse.iitkgp.ac.in/courses/pds/`

Books

- The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall of India
- Programming with C, Byron S. Gottfried, Schaum's Outline Series, 2nd Edition, Tata McGraw-Hill, 2006
- The Spirit of C by Henry Mullish and Herbert Cooper, Jaico Publishing House, 2006
- Any good book on ANSI C
- How to solve it by computer, R G Dromey, Prentice-Hall International, 1982

Course objectives

- ‘C’ programming
- Problem solving

‘C’ programming

- Easier part of the course
- Programs should be *grammatically* correct (easy)
- Programs should compile (easy)
- Good programming habits
- Know how to run programs
- What do we write the program for?
- Usually to solve a problem

‘C’ programming

- Easier part of the course
- Programs should be *grammatically* correct (easy)
- Programs should compile (easy)
- Good programming habits
- Know how to run programs
- What do we write the program for?
- Usually to solve a problem

'C' programming

- Easier part of the course
- Programs should be *grammatically* correct (easy)
- Programs should compile (easy)
- Good programming habits
- Know how to run programs
- What do we write the program for?
- Usually to solve a problem

‘C’ programming

- Easier part of the course
- Programs should be *grammatically* correct (easy)
- Programs should compile (easy)
- Good programming habits
- Know how to run programs
- What do we write the program for?
- Usually to solve a problem

‘C’ programming

- Easier part of the course
- Programs should be *grammatically* correct (easy)
- Programs should compile (easy)
- Good programming habits
- Know how to run programs
- What do we write the program for?
- Usually to solve a problem

Problem solving

- Harder part of the course
- Requires creative thinking
- One writes a program to make the computer carry out the steps identified to solve a problem
- The solution consists of a set of steps which must be carried out in the correct sequence – identified manually (by you)
- This is a *"programme"* for solving the problem
- Codification of this *"programme"* in a suitable computer language, such as 'C' is computer programming
- Solution to the problem *must* precede writing of the *program*

Problem solving

- Harder part of the course
- Requires creative thinking
- One writes a program to make the computer carry out the steps identified to solve a problem
- The solution consists of a set of steps which must be carried out in the correct sequence – identified manually (by you)
- This is a *"programme"* for solving the problem
- Codification of this *"programme"* in a suitable computer language, such as 'C' is computer programming
- Solution to the problem *must* precede writing of the *program*

Problem solving

- Harder part of the course
- Requires creative thinking
- One writes a program to make the computer carry out the steps identified to solve a problem
- The solution consists of a set of steps which must be carried out in the correct sequence – identified manually (by you)
- This is a *"programme"* for solving the problem
- Codification of this *"programme"* in a suitable computer language, such as 'C' is computer programming
- Solution to the problem *must* precede writing of the *program*

Section outline

2. **Simple programming exercise**
   - Sum of two numbers
   - A few shell commands

Summing two numbers

Let the two numbers be a and b

Either Assign some values to a and b

Example: $a = 6$ and $b = 14$

Or Read in values for a and b

Let the sum be $s = a + b$

How to know the value of s – display it?

Summing two numbers

Let the two numbers be a and b

Either Assign some values to a and b

Example: $a = 6$ and $b = 14$

Or Read in values for a and b

Let the sum be $s = a + b$

How to know the value of s – display it?

Summing two numbers

Let the two numbers be a and b

Either Assign some values to a and b

Example: $a = 6$ and $b = 14$

Or Read in values for a and b

Let the sum be $s = a + b$

How to know the value of s – display it?

Summing two numbers

Let the two numbers be a and b

Either Assign some values to a and b

Example: $a = 6$ and $b = 14$

Or Read in values for a and b

Let the sum be $s = a + b$

How to know the value of s – display it?

Sum program

We should do each program in a separate directory.
Open *first* terminal window and do the following:

Command shell:

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Enter the following lines in a text file `sum.c` using your preferred editor such as: vi, gvim, emacs, kwrite, etc.

Editor:

```
  a=6;
  b=14;
  s=a+b;
```

Sum program

We should do each program in a separate directory.
Open *first* terminal window and do the following:

Command shell:

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Enter the following lines in a text file **`sum.c`** using your preferred editor such as: vi, gvim, emacs, kwrite, etc.

Editor:

```
  a=6;
  b=14;
  s=a+b;
```

Sum program (contd.)

We first need to compile the program using the `cc` command

Compile it:

```
$ cc sum.c -o sum
sum.c:1: warning: data definition has no type or stora
sum.c:2: warning: data definition has no type or stora
sum.c:3: warning: data definition has no type or stora
sum.c:3: error: initializer element is not constant
make: *** [sum] Error 1
```

A few more things need to be done to have a correct ‘C’ program

Sum program (contd.)

Edit **`sum.c`** so that it as follows:

Editor:

```
int main() {
  int a=6;
  int b=14;
  int s;

  s=a+b;

  return 0;
}
```

Compile it and run it:

```
$ cc sum.c -o sum
$ $ ./sum
$
```

Sum program (contd.)

Edit **`sum.c`** so that it as follows:

Editor:

```
int main() {
  int a=6;
  int b=14;
  int s;

  s=a+b;

  return 0;
}
```

Compile it and run it:

```
$ cc sum.c -o sum
$ $ ./sum
$
```

Sum program (contd.)

There is no output!
We need to add a *statement* to print **s**
Edit **sum.c** so that it as follows:

Editor:

```
int main() {
  int a=6;
  int b=14;
  int s;

  s=a+b;
  printf ("sum=%d\n", s);

  return 0;
}
```

Sum program (contd.)

There is no output!
We need to add a *statement* to print **s**
Edit **sum.c** so that it as follows:

Editor:

```
int main() {
   int a=6;
   int b=14;
   int s;

   s=a+b;
   printf ("sum=%d\n", s);

   return 0;
}
```

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
sum.c: In function 'main':
sum.c:7: warning: incompatible implicit declaration of
```

The **`printf`** *'C'-function* is not being recognised in the correct way.

Sum program (contd.)

Edit **sum.c** so that it as follows:

Editor:

```
#include <stdio.h>
int main() {
  int a=6;
  int b=14;
  int s;

  s=a+b;
  printf ("sum=%d\n", s);
}
```

Files with suffix '.h' are meant to contain definitions, which you will see later.

A glimpse of stdio.h (contd.)

Usually located under **`/usr/include/`**

Editor:

```
// ...
#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H     1
# include <features.h>

__BEGIN_DECLS

# define __need_size_t
# define __need_NULL
# include <stddef.h>

// ...
```

A glimpse of stdio.h (contd.)

Editor:

```
// ...
/* Write formatted output to stdout.

  This function is a possible cancellation point and
therefore not
  marked with __THROW. */
extern int printf (__const char *__restrict __format, ...);
/* Write formatted output to S. */
extern int sprintf (char *__restrict __s,
        __const char *__restrict __format, ...) __THROW;
// ...
```

Sum program (contd.)

Earlier commands...

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
sum=20
```

Sum program (contd.)

Earlier commands...

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
sum=20
```

Sum program (contd.)

- This program is only good for adding 6 and 14
- Not worth the effort!
- Let it add two integer numbers
- We will have to supply the numbers.
- The program needs to read the two numbers

Sum program (contd.)

Edit **sum.c** so that it as follows:

Editor:

```
#include <stdio.h>
// program to add two numbers
int main() {
  int a, b, s;

  scanf ("%d%d", &a, &b);
  s=a+b; /* sum of a & b */
  printf ("sum=%d\n", s);

  return 0;
}
```

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
10 35
sum=45
```

- Is this programm easy to use?
- Can the programme be more interactive?

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
10 35
sum=45
```

- Is this programm easy to use?
- Can the programme be more interactive?

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
10 35
sum=45
```

- Is this programm easy to use?
- Can the programme be more interactive?

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
10 35
sum=45
```

- Is this programm easy to use?
- Can the programme be more interactive?

Sum program (contd.)

Compile it:

```
$ cc sum.c -o sum
$
```

Run it:

```
$ ./sum
10 35
sum=45
```

- Is this programm easy to use?
- Can the programme be more interactive?

Sum program (contd.)

Editor:

```
#include <stdio.h>
// program to add two numbers
int main() {
  int a, b, s;

  printf ("Enter a: "); // prompt for value of a
  scanf ("%d", &a); // read in value of a
  printf ("Enter b: "); // prompt for value of b
  scanf ("%d", &b); // read in value of b
  s=a+b; /* sum of a & b */
  printf ("sum=%d\n", s);

  return 0;
}
```

Sum program (contd.)

Earlier commands...

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Compile it and run it:

```
$ cc sum.c -o sum
$ ./sum
Enter a: 10
Enter b: 35
sum=45
```

Sum program (contd.)

Earlier commands...

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Compile it and run it:

```
$ cc sum.c -o sum
$ ./sum
Enter a: 10
Enter b: 35
sum=45
```

Sum program (contd.)

Earlier commands...

```
$ mkdir sum
$ cd sum
$ gvim sum.c &
```

Compile it and run it:

```
$ cc sum.c -o sum
$ ./sum
Enter a: 10
Enter b: 35
sum=45
```

A few shell commands

- When a new terminal window is opened, a command shell is run inside it
- This command shell usuall provides a (shell) prompt which is often a short string ending with ‘$’ or ‘$>$’
- The command shell can run shell commands, such as “ls”, “mkdir *dirName*”, “cd *targetDir*”, “cd ..”, “rm *fileName*”
- It can also run other programs, such “gvim *fileName*.c &”, “gcc *fileName*.c -o *fileName*”
- The ‘&’ at the end of the command causes the command to run in the background and the shell prompt re-appears so that a new command can be executed

Sum program (contd.)

Can this program add two real numbers?

Run it:

```
$ ./sum
Enter a: 4.5
Enter b: sum=-1077924036
```

- Representation of data in computers is an important issue.
- “Integer” numbers and “real” numbers have different (finite) representations in computers
- Different computers (computer architectures) may have incompatible representations
- It is important that programs written in high-level languages be architecture independent (as far as possible)

Sum program (contd.)

Can this program add two real numbers?

Run it:

```
$ ./sum
Enter a: 4.5
Enter b: sum=-1077924036
```

- Representation of data in computers is an important issue.
- “Integer” numbers and “real” numbers have different (finite) representations in computers
- Different computers (computer architectures) may have incompatible representations
- It is important that programs written in high-level languages be architecture independent (as far as possible)

Sum program (contd.)

Can this program add two real numbers?

Run it:

```
$ ./sum
Enter a: 4.5
Enter b: sum=-1077924036
```

- Representation of data in computers is an important issue.
- "Integer" numbers and "real" numbers have different (finite) representations in computers
- Different computers (computer architectures) may have incompatible representations
- It is important that programs written in high-level languages be architecture independent (as far as possible)

Sum program (contd.)

Can this program add two real numbers?

Run it:

```
$ ./sum
Enter a: 4.5
Enter b: sum=-1077924036
```

- Representation of data in computers is an important issue.
- "Integer" numbers and "real" numbers have different (finite) representations in computers
- Different computers (computer architectures) may have incompatible representations
- It is important that programs written in high-level languages be architecture independent (as far as possible)

Variables

- Variable names are formed out of letters: a..z, A..Z; digits: 0..9 and the underscore: '_'
- A variable name may not start with a digit
- `a a_b`, `a5`, `a_5`, `_a`
- Variable names should be sensible and intuitive – no need for excessive abbreviation – `smallest`, `largest`, `median`, `largest_2`
- Convenient to start variable names with lower case letters – easier to type
- Upper case letters or '_' may be used for multi-word names – `idxL`, `idxR`, `idx_left`, `idx_right`, `idxLeft`, `idxRight`

Variables

- Variable names are formed out of letters: a..z, A..Z; digits: 0..9 and the underscore: '_'
- A variable name may not start with a digit
- **`a a_b`**, **`a5`**, **`a_5`**, **`_a`**
- Variable names should be sensible and intuitive – no need for excessive abbreviation – **`smallest`**, **`largest`**, **`median`**, **`largest_2`**
- Convenient to start variable names with lower case letters – easier to type
- Upper case letters or '_' may be used for multi-word names – **`idxL`**, **`idxR`**, **`idx_left`**, **`idx_right`**, **`idxLeft`**, **`idxRight`**

Variables

- Variable names are formed out of letters: a..z, A..Z; digits: 0..9 and the underscore: '_'
- A variable name may not start with a digit
- **`a a_b`**, **`a5`**, **`a_5`**, **`_a`**
- Variable names should be sensible and intuitive – no need for excessive abbreviation – **`smallest`**, **`largest`**, **`median`**, **`largest_2`**
- Convenient to start variable names with lower case letters – easier to type
- Upper case letters or '_' may be used for multi-word names – **`idxL`**, **`idxR`**, **`idx_left`**, **`idx_right`**, **`idxLeft`**, **`idxRight`**

Variables

- Variable names are formed out of letters: a..z, A..Z; digits: 0..9 and the underscore: '_'
- A variable name may not start with a digit
- **`a a_b`**, **`a5`**, **`a_5`**, **`_a`**
- Variable names should be sensible and intuitive – no need for excessive abbreviation – **`smallest`**, **`largest`**, **`median`**, **`largest_2`**
- Convenient to start variable names with lower case letters – easier to type
- Upper case letters or '_' may be used for multi-word names – **`idxL`**, **`idxR`**, **`idx_left`**, **`idx_right`**, **`idxLeft`**, **`idxRight`**

Typing of variables

In ‘C’ variables hold data of a particular type, such as int.
We will see more on types later. Common base types are as follows:

int for storing “integers” – actually a small subset of integers

float for storing “real numbers” – actually a small subset thereof

char for storing characters – letters, punctuation marks, digits as “letters”, other characters

Typing of variables

In ‘C’ variables hold data of a particular type, such as int.
We will see more on types later. Common base types are as follows:

int for storing “integers” – actually a small subset of integers

float for storing “real numbers” – actually a small subset thereof

char for storing characters – letters, punctuation marks, digits as “letters”, other characters

Example of variable declarations

Editor

```
int count, idx, i=0;
float avg=0.0, root_1, root_2;
char letter='a', digit='0', punct=':';
char name[30]; // for a string of characters
```

Storage of strings require use of arrays, to be seen later
User defined are possible, also to be seen later

Section outline

Use of `printf`

```
printf ("sum=%d\n", s);
```

- It is actually a *'C'-function*, that takes a number of *parameters*
- *'C'-functions* are to be discussed later, in detail
- For now, we only learn to use `printf` and `scanf`
- The parameters taken by the above *call* to `printf` are as follows:
- `"sum=%d\n"`
- `s`

Use of `printf`

```
printf ("sum=%d\n", s);
```

- It is actually a *'C'-function*, that takes a number of *parameters*
- *'C'-functions* are to be discussed later, in detail
- For now, we only learn to use **`printf`** and **`scanf`**
- The parameters taken by the above *call* to **`printf`** are as follows:
- `"sum=%d\n"`
- `s`

Use of `printf`

```
printf ("sum=%d\n", s);
```

- It is actually a *'C'-function*, that takes a number of *parameters*
- *'C'-functions* are to be discussed later, in detail
- For now, we only learn to use **`printf`** and **`scanf`**
- The parameters taken by the above *call* to **`printf`** are as follows:
- `"sum=%d\n"`
- `s`

Use of `printf` (contd.)

- The argument `"sum=%d\n"` is the *format* argument, it says
- the string `sum=` is to be printed, then
- and integer is to be printed in place of `%d`, in *decimal* notation, and finally
- `\n` is to be printed, resulting in a *newline*
- `%d` is a place holder for an integer,
- the second argument `s` takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument `"sum=%d\n"` is the *format* argument, it says
- the string `sum=` is to be printed, then
- and integer is to be printed in place of `%d`, in *decimal* notation, and finally
- `\n` is to be printed, resulting in a *newline*
- `%d` is a place holder for an integer,
- the second argument `s` takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument `"sum=%d\n"` is the *format* argument, it says
- the string `sum=` is to be printed, then
- and integer is to be printed in place of `%d`, in *decimal* notation, and finally
- `\n` is to be printed, resulting in a *newline*
- `%d` is a place holder for an integer,
- the second argument `s` takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument `"sum=%d\n"` is the *format* argument, it says
- the string `sum=` is to be printed, then
- and integer is to be printed in place of `%d`, in *decimal* notation, and finally
- `\n` is to be printed, resulting in a *newline*
- `%d` is a place holder for an integer,
- the second argument `s` takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument **`"sum=%d\n"`** is the *format* argument, it says
- the string **`sum=`** is to be printed, then
- and integer is to be printed in place of **`%d`**, in *decimal* notation, and finally
- **`\n`** is to be printed, resulting in a *newline*
- **`%d`** is a place holder for an integer,
- the second argument **`s`** takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument `"sum=%d\n"` is the *format* argument, it says
- the string `sum=` is to be printed, then
- and integer is to be printed in place of `%d`, in *decimal* notation, and finally
- `\n` is to be printed, resulting in a *newline*
- `%d` is a place holder for an integer,
- the second argument `s` takes the place of that integer
- In the example the value of `s` was 45
- Suppose that 45 is internally represented as `0101101`
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument **`"sum=%d\n"`** is the *format* argument, it says
- the string **`sum=`** is to be printed, then
- and integer is to be printed in place of **`%d`**, in *decimal* notation, and finally
- **`\n`** is to be printed, resulting in a *newline*
- **`%d`** is a place holder for an integer,
- the second argument **`s`** takes the place of that integer
- In the example the value of **`s`** was 45
- Suppose that 45 is internally represented as **`0101101`**
- Because of the `%d`, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument **`"sum=%d\n"`** is the *format* argument, it says
- the string **`sum=`** is to be printed, then
- and integer is to be printed in place of **`%d`**, in *decimal* notation, and finally
- **`\n`** is to be printed, resulting in a *newline*
- **`%d`** is a place holder for an integer,
- the second argument **`s`** takes the place of that integer
- In the example the value of **`s`** was 45
- Suppose that 45 is internally represented as **`0101101`**
- Because of the **`%d`**, the value gets printed as 45, in decimal notation
- Other notations are also possible

Use of `printf` (contd.)

- The argument **`"sum=%d\n"`** is the *format* argument, it says
- the string **`sum=`** is to be printed, then
- and integer is to be printed in place of **`%d`**, in *decimal* notation, and finally
- **`\n`** is to be printed, resulting in a *newline*
- **`%d`** is a place holder for an integer,
- the second argument **`s`** takes the place of that integer
- In the example the value of **`s`** was 45
- Suppose that 45 is internally represented as **`0101101`**
- Because of the **`%d`**, the value gets printed as 45, in decimal notation
- Other notations are also possible

Also hexadecimal and octal

Editor: sum2.c

```
int main() {
  int a=10, b=35, s;

  s=a+b;
  printf ("sum: %d(dec), %x(hex), %X(HEX), %o(oct)\n",
    s, s, s, s);

  return 0;
}
```

Compile and run:

```
$ cc sum2.c -o sum2
$ ./sum2
sum: 45(dec), 2d(hex), 2D(HEX), 55(oct)
```

Printing real numbers

- The 'C' terminology for real numbers is **`float`**
- The *conversion specifier* for a "real" number is `f`,
- commonly used as `%f`
- The result of dividing 5345652.1 by 3.4 may be printed as:
- `printf("%f\n", 5345652.1/3.4);`
- Output: `1572250.617647`
- Number of places after the decimal point (radix character) (precision) can be changed
- `printf("%.8f\n", 5345652.1/3.4);`
- Output: `1572250.61764706`
- Length (field width) can be changed
- `printf("%14.4f\n", 5345652.1/3.4);`
- Output: `   1572250.6176`
- More details: `man 3 printf`

Printing real numbers

- The ‘C’ terminology for real numbers is **`float`**
- The *conversion specifier* for a “real” number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- `printf("%f\n", 5345652.1/3.4);`
- Output: `1572250.617647`
- Number of places after the decimal point (radix character) (precision) can be changed
- `printf("%.8f\n", 5345652.1/3.4);`
- Output: `1572250.61764706`
- Length (field width) can be changed
- `printf("%14.4f\n", 5345652.1/3.4);`
- Output: `  1572250.6176`
- More details: `man 3 printf`

Printing real numbers

- The 'C' terminology for real numbers is **`float`**
- The *conversion specifier* for a "real" number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- `printf("%f\n", 5345652.1/3.4);`
- Output: `1572250.617647`
- Number of places after the decimal point (radix character) (precision) can be changed
- `printf("%.8f\n", 5345652.1/3.4);`
- Output: `1572250.61764706`
- Length (field width) can be changed
- `printf("%14.4f\n", 5345652.1/3.4);`
- Output: `   1572250.6176`
- More details: `man 3 printf`

Printing real numbers

- The 'C' terminology for real numbers is **`float`**
- The *conversion specifier* for a "real" number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- **`printf("%f\n", 5345652.1/3.4);`**
- Output: **`1572250.617647`**
- Number of places after the decimal point (radix character) (precision) can be changed
- **`printf("%.8f\n", 5345652.1/3.4);`**
- Output: **`1572250.61764706`**
- Length (field width) can be changed
- **`printf("%14.4f\n", 5345652.1/3.4);`**
- Output: **`   1572250.6176`**
- More details: **`man 3 printf`**

Printing real numbers

- The 'C' terminology for real numbers is **`float`**
- The *conversion specifier* for a "real" number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- **`printf("%f\n", 5345652.1/3.4);`**
- Output: **`1572250.617647`**
- Number of places after the decimal point (radix character) (precision) can be changed
- **`printf("%.8f\n", 5345652.1/3.4);`**
- Output: **`1572250.61764706`**
- Length (field width) can be changed
- **`printf("%14.4f\n", 5345652.1/3.4);`**
- Output: `   1572250.6176`
- More details: **`man 3 printf`**

Printing real numbers

- The ‘C’ terminology for real numbers is **`float`**
- The *conversion specifier* for a “real” number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- **`printf("%f\n", 5345652.1/3.4);`**
- Output: **`1572250.617647`**
- Number of places after the decimal point (radix character) (precision) can be changed
- **`printf("%.8f\n", 5345652.1/3.4);`**
- Output: **`1572250.61764706`**
- Length (field width) can be changed
- **`printf("%14.4f\n", 5345652.1/3.4);`**
- Output: **`   1572250.6176`**
- More details: **`man 3 printf`**

Printing real numbers

- The ‘C’ terminology for real numbers is **`float`**
- The *conversion specifier* for a “real” number is **`f`**,
- commonly used as **`%f`**
- The result of dividing 5345652.1 by 3.4 may be printed as:
- **`printf("%f\n", 5345652.1/3.4);`**
- Output: **`1572250.617647`**
- Number of places after the decimal point (radix character) (precision) can be changed
- **`printf("%.8f\n", 5345652.1/3.4);`**
- Output: **`1572250.61764706`**
- Length (field width) can be changed
- **`printf("%14.4f\n", 5345652.1/3.4);`**
- Output: **`   1572250.6176`**
- More details: **`man 3 printf`**

More conversion specifiers (in brief)

`d,i` The **`int`** argument is converted to signed decimal notation

`o,u,x,X` The **`unsigned int`** argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation

`f,F` The **`double`** argument is rounded and converted to decimal notation in the style [-]ddd.ddd

More conversion specifiers (in brief)

d,i The **int** argument is converted to signed decimal notation

o,u,x,X The **unsigned int** argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation

f,F The **double** argument is rounded and converted to decimal notation in the style [-]ddd.ddd

More conversion specifiers (in brief)

- **`d,i`** The **`int`** argument is converted to signed decimal notation
- **`o,u,x,X`** The **`unsigned int`** argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation
- **`f,F`** The **`double`** argument is rounded and converted to decimal notation in the style [-]ddd.ddd

More conversion specifiers (contd.)

e,E The **double** argument is rounded and converted in the style [-]d.ddde$\pm$dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

More conversion specifiers (contd.)

- **c** The `int` argument is converted to an `unsigned char`, and the resulting character is written.
- **s** Characters from the array are written up to (but not including) a terminating `NUL` character. A length (precision) may also be specified.
- **p** The `void *` pointer argument is printed in hexadecimal
- **%** To output a `%`

More conversion specifiers (contd.)

- **c** The **`int`** argument is converted to an **`unsigned char`**, and the resulting character is written.
- **s** Characters from the array are written up to (but not including) a terminating **`NUL`** character. A length (precision) may also be specified.
- **p** The `void *` pointer argument is printed in hexadecimal
- **%** To output a `%`

More conversion specifiers (contd.)

- **c** The `int` argument is converted to an `unsigned char`, and the resulting character is written.
- **s** Characters from the array are written up to (but not including) a terminating `NUL` character. A length (precision) may also be specified.
- **p** The `void *` pointer argument is printed in hexadecimal
- **%** To output a %

More conversion specifiers (contd.)

- **`c`** The **`int`** argument is converted to an **`unsigned char`**, and the resulting character is written.
- **`s`** Characters from the array are written up to (but not including) a terminating **`NUL`** character. A length (precision) may also be specified.
- **`p`** The **`void *`** pointer argument is printed in hexadecimal
- **`%`** To output a `%`

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the '`&`'
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the ‘`&`’
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

Revisiting earlier call to `scanf`

- `scanf ("%d", &a);` differs from a similar call to `printf`
- `printf ("sum=%d\n", s);` – the ‘`&`’
- In case of `printf`, the decimal value contained in `s` is to be printed
- In the call `printf ("sum=%d\n", s);`, the *value* of `s` (say, 45) was passed on for printing
- In case of `scanf`, (as in the call above) there is no question of passing on the value of `a`, instead
- we want to *receive* a value of `a`
- How is that to be achieved?

An analogy to `scanf`

- Suppose that you wish to place an order to purchase a sack of rice from a shop
- You supply the shop keeper the *address* of your house for delivering (or putting) the product there
- How about supplying `scanf` the *address* of `a` so that it can put an integer there
- `&a` is simply the address of the variable `a`, which is supplied to `scanf` for reading in an integer into `a`

An analogy to `scanf`

- Suppose that you wish to place an order to purchase a sack of rice from a shop
- You supply the shop keeper the *address* of your house for delivering (or putting) the product there
- How about supplying `scanf` the *address* of `a` so that it can put an integer there
- `&a` is simply the address of the variable `a`, which is supplied to `scanf` for reading in an integer into `a`

An analogy to `scanf`

- Suppose that you wish to place an order to purchase a sack of rice from a shop
- You supply the shop keeper the *address* of your house for delivering (or putting) the product there
- How about supplying **`scanf`** the *address* of **`a`** so that it can put an integer there
- **`&a`** is simply the address of the variable **`a`**, which is supplied to **`scanf`** for reading in an integer into **`a`**

An analogy to `scanf`

- Suppose that you wish to place an order to purchase a sack of rice from a shop
- You supply the shop keeper the *address* of your house for delivering (or putting) the product there
- How about supplying **`scanf`** the *address* of **`a`** so that it can put an integer there
- **`&a`** is simply the address of the variable **`a`**, which is supplied to **`scanf`** for reading in an integer into **`a`**

An analogy to `scanf`

- Suppose that you wish to place an order to purchase a sack of rice from a shop
- You supply the shop keeper the *address* of your house for delivering (or putting) the product there
- How about supplying **`scanf`** the *address* of **`a`** so that it can put an integer there
- **`&a`** is simply the address of the variable **`a`**, which is supplied to **`scanf`** for reading in an integer into **`a`**

Simple view of (little endian) (int) data storage

...	....	....	....	....
address	....	....	....	**....**
v_1	00000000	00011110	00111000	11001011
address	3071	3070	3069	**3068**
a	00000000	00010100	00101110	11101011
address	3075	3074	3073	**3072**
...	....	....	....	....
address	....	....	....	**....**
s	00000000	00000000	00000000	00101101
address	3875	3874	3873	**3872**
...	....	....	....	....
address	....	....	....	**....**

Value of s is 45, **address** of s is 3872 and **address** of a is 3072
Garbage in a. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

`...`	`.........`	`.........`	`.........`	`.........`
address	....	....	....	....
`v_1`	`00000000`	`00011110`	`00111000`	`11001011`
address	3071	3070	3069	**3068**
`a`	`00000000`	`00010100`	`00101110`	`11101011`
address	3075	3074	3073	**3072**
`...`	`.........`	`.........`	`.........`	`.........`
address	....	....	....	....
`s`	`00000000`	`00000000`	`00000000`	`00101101`
address	3875	3874	3873	**3872**
`...`	`.........`	`.........`	`.........`	`.........`
address	....	....	....	....

Value of `s` is 45, **address** of `s` is 3872 and **address** of `a` is 3072

Garbage in `a`. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`v_1`	00000000	00011110	00111000	11001011
address	3071	3070	3069	**3068**
`a`	00000000	00010100	00101110	11101011
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`s`	00000000	00000000	00000000	00101101
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

Value of **`s`** is 45, **address** of **`s`** is 3872 and **address** of **`a`** is 3072

Garbage in **`a`**. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
v_1	00000000	00011110	00111000	11001011
address	3071	3070	3069	**3068**
a	00000000	00010100	00101110	11101011
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
s	00000000	00000000	00000000	00101101
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

Value of **s** is 45, **address** of **s** is 3872 and **address** of **a** is 3072
Garbage in **a**. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`v_1`	`00000000`	`00011110`	`00111000`	`11001011`
address	3071	3070	3069	**3068**
`a`	`00000000`	`00010100`	`00101110`	`11101011`
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`s`	`00000000`	`00000000`	`00000000`	`00101101`
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

Value of `s` is 45, **address** of `s` is 3872 and **address** of `a` is 3072

Garbage in `a`. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

...	....	....	....	....
address	....	....	....	**....**
v_1	00000000	00011110	00111000	11001011
address	3071	3070	3069	**3068**
a	00000000	00010100	00101110	11101011
address	3075	3074	3073	**3072**
...	....	....	....	....
address	....	....	....	**....**
s	00000000	00000000	00000000	00101101
address	3875	3874	3873	**3872**
...	....	....	....	....
address	....	....	....	**....**

Value of `s` is 45, **address** of `s` is 3872 and **address** of `a` is 3072
Garbage in `a`. NB: **Addresses** are divisible by 4 (why?)

Simple view of (little endian) (int) data storage

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`v_1`	00000000	00011110	00111000	11001011
address	3071	3070	3069	**3068**
`a`	00000000	00010100	00101110	11101011
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`s`	00000000	00000000	00000000	00101101
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

Value of `s` is 45, **address** of `s` is 3872 and **address** of `a` is 3072
Garbage in `a`. NB: **Addresses** are divisible by 4 (why?)

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a “real number” given in decimal form or in scientific notation to the internal “real number” representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a “real number” given in decimal form or in scientific notation to the internal “real number” representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

Simple use of `scanf`

- `scanf ("%d", &int_variable);` – to read an integer – for converting a number given in decimal notation to the internal integer representation a pointer to an `int` should be supplied
- `scanf ("%f", &float_variable);` – to read a `float` – for converting a "real number" given in decimal form or in scientific notation to the internal "real number" representation a pointer to a `float` should be supplied
- `scanf ("%c", &char_variable);` – to read a single character – for converting a character to the internal character representation
- `scanf ("%s", string_variable);` – to read a string of characters, note the missing `&`
- to be seen latter – string variables are addresses rather than values

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:
  - **WS** space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)
  - **ordinary** (not WS or `%`); which should be matched exactly (not commonly used)
  - **conversion** heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

WS	space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)
ordinary	(not WS or `%`); which should be matched exactly (not commonly used)
conversion	heavily used

- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:
  - **WS** space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)
  - **ordinary** (not WS or `%`); which should be matched exactly (not commonly used)
  - **conversion** heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

More on `scanf`

- The format string consists of a sequence of directives which describe how to handle the sequence of input characters
- If processing of a directive fails, no more input is read, and scanf returns
- A directive can be:

  WS space, tab, etc.; results in skipping any amount (0 or more) of white space (used to skip white space)

  ordinary (not WS or `%`); which should be matched exactly (not commonly used)

  conversion heavily used
- man 3 scanf for more details
- options are rich to enable reading of data from formatted outputs
- few of those options to be visited later

Illustrating `scanf`

Editor:

```
#include <stdio.h>
// program to add two numbers
int main() {
 int z; char c;
 printf("Enter an int: ");scanf("%d", &z);
 printf("You entered %d\n", z);
 printf("Enter a char: ");scanf("%c", &c);
 printf("You entered `%c'\n", c);
 printf("Enter another char: ");scanf(" %c", &c);
 printf("You entered `%c'\n", c);

  return 0;
}
```

Illustrating `scanf`

Compile and run:

```
$ cc scan.c -o scan
$ ./scan
Enter an int: 5
You entered 5
Enter a char: You entered `
'
Enter another char: w
You entered `w'
```

Section outline

4. **Preprocessor**
   - Including files
   - Macros
   - Conditional compilation

Including files

- `#include <stdio.h>`

  The `<>` braces indicate that the file must be included from the standard compiler include paths, such as `/usr/include/`
- `#include "listTyp.h"`

  Search path is expanded to include the current directory if double quotes are present
  - Error if file is absent
  - Entire text of the file replaces the `#include` directive

Macro definition and expansion

- `#define PI 3.14159`

  `...   area = PI * r * r;`

  Occurrence of `PI` is replaced by its definition, `3.14159`
- `#define RADTODEG(x) ((x) * 57.29578)`

  `deg = RADTODEG(PI);`

  This is a parameterised macro definition, expanded to `((PI) * 57.29578)`, in turn expanded to `((3.14159) * 57.29578)`
- `#define NUM1 5+5`

  `#define NUM2 (5+5)`

  What is the value of `NUM1 * NUM2` ?

Macro definition and expansion

- `#define PI 3.14159`

  `...  area = PI * r * r;`

  Occurrence of `PI` is replaced by its definition, `3.14159`
- `#define RADTODEG(x) ((x) * 57.29578)`

  `deg = RADTODEG(PI);`

  This is a parameterised macro definition, expanded to `((PI) * 57.29578)`, in turn expanded to `((3.14159) * 57.29578)`
- `#define NUM1 5+5`

  `#define NUM2 (5+5)`

  What is the value of `NUM1 * NUM2` ?

Macro definition and expansion

- `#define PI 3.14159`

  `...   area = PI * r * r;`

  Occurrence of `PI` is replaced by its definition, `3.14159`
- `#define RADTODEG(x)  ((x) * 57.29578)`

  `deg = RADTODEG(PI);`

  This is a parameterised macro definition, expanded to `((PI) * 57.29578)`, in turn expanded to `((3.14159) * 57.29578)`
- `#define NUM1 5+5`

  `#define NUM2 (5+5)`

  What is the value of `NUM1 * NUM2` ?

Conditional compilation

Generic:

```
#ifdef NAME
// program text
#else
// more program
text
#endif
```

Specific:

```
#define DEBUG 1
// above line to be
// dropped if not debugging
#ifdef DEBUG
  printf("x=%d, y=%d(dbg)\n",
    x, y); // y is extra
#else
  printf("x=%d\n", x);
  // only the essential
  // matter is printed
#endif
```

- Part between **`#ifdef DEBUG`** and **`#else`** compiled only is **`DEBUG`** is defined (as a macro)
- Otherwise part between **`#else`** and **`#endif`** is compiled

Conditional compilation (contd)

- Editing of files to supply definition of **`DEBUG`** can be avoided, but defining via the command line: **`gcc -D DEBUG ...`** to define **`DEBUG`**
- In this case compilation will happen for the situtation where **`DEBUG`** is defined
- Regular command line (without **`-D DEBUG`**) will not define **`DEBUG`** and result in compilation for the situtation where **`DEBUG`** is undefined

Syllabus (Theory)

Introduction to the Digital Computer;
Introduction to Programming – Variables, Assignment; Expressions; Input/Output;
Conditionals and Branching; Iteration;
Functions; Recursion; Arrays; Introduction to Pointers; Strings; Structures;
Introduction to Data-Procedure Encapsulation;
Dynamic allocation; Linked structures;
Introduction to Data Structure – Stacks and Queues; Searching and Sorting; Time and space requirements.

Part II

Routines and scope

Section outline

5 **Routines and functions**

- Routines
- Examples of routines
- Main routine
- Parameterised routines
- Formal and actual parameters
- Function anatomy
- Functions and macros

Routines

- An important concept – a sequence of steps to perform a specific task
- Usually part of a bigger program
- While programs are run, routines are invoked – from within the program or from other routines
- Routines are a often invoked with parameters
- Recursive routines may even invoke themselves, either directly or via other routines
- Routines often return a value after performing their task
- Routines accepting parameters and returning values are called functions in ‘C’
- In ‘C’ routines are also recursively callablrecursively callableitem In ‘C’, the program is treated as the “main” routine or function

Examples of routines

- A routine to add two numbers and return their sum
- A routine to find and return the greatest of three numbers
- A routine to reverse the digits of a number and return the result
- A routine to find and return the roots of a quadratic equation
- A routine to find a root of a function within a given interval
- A routine to find the number of ways to choose r of n distinct items
- A routine to check whether a given number is prime

Summing two numbers in the main routine

Steps placed directly in the main routine

- Read two numbers
- Add them and save result in **sum**
- Print the value of **sum**

Editor:

```
#include <stdio.h>
// program to add two numbers
int main() {
  int a, b, s;

  scanf ("%d%d", &a, &b);
  s=a+b; /* sum of a & b */
  printf ("sum=%d\n", s);

  return 0;
}
```

Parameterised routines

Consider the routine to add two given numbers

- The routine is identified by a name, say **`sum()`**, the parentheses help to distinguish it from the name of a variable
- Numbers to be added are the parameters for the summation routine, say **`x`** and **`y`**
- Parameters play a dual role:
  - at the time of developing the routine
  - at the time of invoking the routine

Summation as a parameterised routine

- The routine **sum()** takes two parameters: **int x1**, **int x2**, which are to be added
- These are *formal* parameters
- Sum **x1+x2** is saved in **s**
- Finally, **s** is returned
- **sum()** is invoked from **main()** with *actual* parameters

Editor:

```
int sum(int x1, int x2) {
  int s;
  s=x1+x2;
  return s;
}

int main() {
  int a=6;
  int b=14;
  int s;
  s=sum(a, b);
  return 0;
}
```

Formal and actual parameters

- At the time of developing a routine, the actual values to be worked upon are not known
- Routine must be developed with placeholders for the actual values
- Such placeholders are called formal parameters
- When the routine is invoked with placeholders for values to be added, say as `sum (4, 5+3)` or `sum (a, b)`, where `a` and `b` are variables used in the routine from where `sum()` is called, e.g. `main()`
- Parameters actually passed to the function at the time of invocation are called actual parameters
- For 'C' programs, values resulting from evaluation of the actual parameters (which could be expressions) are copied to the formal parameters
- This method of parameter passing is referred to as call by value

Function anatomy

Function name `main`, `sum`

Parameter list `()`, `(int x1, int x2)`

Return type int

Function body `{` *statements* `}`

Return statement `return 0;`

`main()` should return an int:

- **0** indicates regular (successful) termination of program
- **1** or any non-zero indicates faulty termination of program

Formal parameters `x1, x2`

Actual parameters `a, b+5`

Editor:

```
int sum(
  int x1, int x2) {
  int s;
  s=x1+x2;
  return s;
}

int main() {
  int a=6;
  int b=14;
  int s;
  s=sum(a, b+5);
  return 0;
}
```

About using functions

- Coding becomes more structured – separation of usage and implementation
- Repetition of similar code can be avoided
- Recursive definitions are easily accommodated
- Avoid non-essential input/output inside functions

Parameter passing

Editor:

```
int sum_fun (int a, int b) {
  return a + b;
}
...
  int x=5;
  sum_fun(x++, x++) ;
...
```

- What are the actual parameters to **`sum_fun`** ?
- If the first parameter is evaluated first, then invocation takes place as **`sum_fun(5, 6)`**
- If the second parameter is evaluated first, then invocation takes place as **`sum_fun(6, 5)`**
- The language standard does not specify the order of parameter evaluation
- Bad practice to use function calls that are sensitive to the order of parameter evaluation

Parameter passing

Editor:

```
int sum_fun (int a, int b) {
  return a + b;
}
...
  int x=5;
  sum_fun(x++, x++) ;
...
```

- What are the actual parameters to **sum_fun** ?
- If the first parameter is evaluated first, then invocation takes place as **sum_fun(5, 6)**
- If the second parameter is evaluated first, then invocation takes place as **sum_fun(6, 5)**
- The language standard does not specify the order of parameter evaluation
- Bad practice to use function calls that are sensitive to the order of parameter evaluation

Parameter passing

Editor:

```
int sum_fun (int a, int b) {
  return a + b;
}
...
  int x=5;
  sum_fun(x++, x++) ;
...
```

- What are the actual parameters to **sum_fun** ?
- If the first parameter is evaluated first, then invocation takes place as **sum_fun(5, 6)**
- If the second parameter is evaluated first, then invocation takes place as **sum_fun(6, 5)**
- The language standard does not specify the order of parameter evaluation
- Bad practice to use function calls that are sensitive to the order of parameter evaluation

Parameter passing

Editor:

```
int sum_fun (int a, int b) {
  return a + b;
}
...
  int x=5;
  sum_fun(x++, x++) ;
...
```

- What are the actual parameters to **`sum_fun`** ?
- If the first parameter is evaluated first, then invocation takes place as **`sum_fun(5, 6)`**
- If the second parameter is evaluated first, then invocation takes place as **`sum_fun(6, 5)`**
- The language standard does not specify the order of parameter evaluation
- Bad practice to use function calls that are sensitive to the order of parameter evaluation

Parameter passing

Editor:

```
int sum_fun (int a, int b) {
  return a + b;
}
...
  int x=5;
  sum_fun(x++, x++) ;
...
```

- What are the actual parameters to **`sum_fun`** ?
- If the first parameter is evaluated first, then invocation takes place as **`sum_fun(5, 6)`**
- If the second parameter is evaluated first, then invocation takes place as **`sum_fun(6, 5)`**
- The language standard does not specify the order of parameter evaluation
- Bad practice to use function calls that are sensitive to the order of parameter evaluation

Functions and macros

Example

```
#define isZero(x) (x < EPSILON && x > -EPSILON)
int isZero(x) {
  return (x < EPSILON && x > -EPSILON) ;
}
```

- A function is called, as already explained
- A macro is expanded where it is used,
  - the call is replaced by its definition
  - text of the parameters, if any, gets copied wherever they are used

Example

```
isZero(2+3) → (2+3 < EPSILON && 2+3 > -EPSILON)
```

Section outline

6. **Scope**
   - Function scope
   - Block scope
   - Global variables
   - Static variables

Function scope

Editor:

```
#include <stdio.h>
float sq_x_plus2 (float x) {
  x += 2; // increment x by 2
  x *= x; // now square
  return x;
}
main() {   float x=5.0;
  printf("sq_x_plus2(%f)=%f\n",
      x, sq_x_plus2(x));
  printf("x=%f\n", x);
}
```

Compile and run:

```
$cc sq_x_plus2.c -o sq_x_plus2
$ ./sq_x_plus2
sq_x_plus2(5.000000)=49.000000
x=5.000000
```

- Scope of a declaration is the part of the program to which it is applicable
- The variables named **`x`** in **`sq_x_plus2()`** and **`main()`** are independent
- Scope of a variable is restricted to within the function where it is declared
- Scope of a function parameter extends to all parts within the function where it is declared

Block scope

Simple example

```
#include <stdio.h>
float sq_x_plus2 (float x) {
  x += 2; // increment x by 2
  x *= x; // now square
  return x;
}
main() {
  float x=5.0;
  printf("sq_x_plus2(%f)=%f\n",
      x, sq_x_plus2(x));
  printf("x=%f\n", x);
  { // new sub-block
    int x;
    // scope of x
  }
}
```

Scope in blocks

```
fun(int test) {
  int test; // invalid
  // clash with test
}
main() {
  int test;
  // scope of test
  { // new sub-block
    int test;
    // scope of test
  }
  { // another sub-block
    int test;
    // scope of test
  }
}
```

Global variables

File 1

```
int varA; // global
// scope, normal memory
// allocation is done
```

File 2

```
extern int varA;
// no allocation
// of memory
```

- Scope of variable declaration outside a function is global to all functions
- Declaration is overridden by a variable of the same name in a function or a block therein
- A global variable in one file can be linked to the declaration of the same variable (matching in type) in another file via the **`extern`** keyword
- Declaration with extern does not lead to memory allocation for declared item – instead linked to original declaration

Static variables

File 1

```
static int varA; // global
// but only in this file
void funA() {
  static int callCntA;
// local to this function, value
// retained across function calls
callCntA++; // keeps count of
// calls to funA()
}
void funB() {
  int varD;
// local and value not retained
// across function calls
}
```

- `static` variables have linkage restricted to declarations and definitions within local file
- `static` variables declared wthin functions retain value across function calls
- Conflicts with *re-entrant* nature of functions

Usage of `static`

- Except for special applications, where **`static`** is convenient, it should not be used
- Unlike “normal” variables within functions, which are allocated fresh with every function call, **`static`** variables are not
- **`extern`** and **`static`** do not mix (oxymoron)
- Non-re-entrant nature of **`static`** can be a problem if used carelessly in functions

Part III

Operators and expression evaluation

Section outline

7. **Operators and expression evaluation**
   - Operators
   - Associativity and Precedence Relationships

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary − Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary − Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float - int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float - int` is `float`

Binary ⋆ Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float - int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary ∗ Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add **int** and **float**
int + int is **int**
any other combination, eg **int + float** is **float**

Binary – Subtract **int** and **float**
int – int is **int**
any other combination, eg **float - int** is **float**

Binary * Multiply **int** and **float**
int * int is **int**
any other combination is **float**

Binary / Divide **int** and **float**
int / int is **int** (quotient)
any other combination, eg **float * float** is **float**
(result is as for "real division")

Binary % Remainder of dividing **int** by **int**

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary ⋆ Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for “real division”)

Binary % Remainder of dividing `int` by `int`

No exponentiation ‘C’ does not provide an exponentiation operation

Arithmetic operators

Binary + Add **int** and **float**
int + int is **int**
any other combination, eg **int + float** is **float**

Binary – Subtract **int** and **float**
int – int is **int**
any other combination, eg **float - int** is **float**

Binary * Multiply **int** and **float**
int * int is **int**
any other combination is **float**

Binary / Divide **int** and **float**
int / int is **int** (quotient)
any other combination, eg **float * float** is **float**
(result is as for "real division")

Binary % Remainder of dividing **int** by **int**

No exponentiation 'C' does not provide an exponentiation operation

Arithmetic operators

Binary + Add `int` and `float`
`int + int` is `int`
any other combination, eg `int + float` is `float`

Binary – Subtract `int` and `float`
`int - int` is `int`
any other combination, eg `float` - `int` is `float`

Binary * Multiply `int` and `float`
`int * int` is `int`
any other combination is `float`

Binary / Divide `int` and `float`
`int / int` is `int` (quotient)
any other combination, eg `float * float` is `float`
(result is as for "real division")

Binary % Remainder of dividing `int` by `int`

No exponentiation 'C' does not provide an exponentiation operation

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of `a` ? 1; integer division
- `float x ; x = 10 / 6 ;` value of `x` ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of `x` ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of `b` ? 1; still “real division” but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of `a` ? 1; integer division
- `float x ; x = 10 / 6 ;` value of `x` ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of `x` ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of `b` ? 1; still "real division" but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of `a` ? 1; integer division
- `float x ; x = 10 / 6 ;` value of `x` ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of `x` ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of `b` ? 1; still “real division” but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of `a` ? 1; integer division
- `float x ; x = 10 / 6 ;` value of `x` ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of `x` ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of `b` ? 1; still “real division” but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;` the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;` type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of `a` ? 1; integer division
- `float x ; x = 10 / 6 ;` value of `x` ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of `x` ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of `b` ? 1; still "real division" but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still "real division" but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a **float**
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still "real division" but result is assigned to **int** after truncation
- `int a ; float x ; a = (int) x ;`
  the **float** value is *cast* into an **int** and then that value is assigned to **a**
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- **`int a ; a = 10 / 6 ;`** value of **`a`** ? 1; integer division
- **`float x ; x = 10 / 6 ;`** value of **`x`** ? 1.0; division is still integer only value is stored in a **`float`**
- **`float x ; x = 10.0 / 6.0 ;`** value of **`x`** ? 1.666666; "real division"
- **`int b ; x = 10.0 / 6.0 ;`** value of **`b`** ? 1; still "real division" but result is assigned to **`int`** after truncation
- **`int a ; float x ; a = (int) x ;`**
  the **`float`** value is *cast* into an **`int`** and then that value is assigned to **`a`**
- **`int a ; float x ; x = a ;`**
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a **float**
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still "real division" but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a **float**
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still “real division” but result is assigned to **int** after truncation
- `int a ; float x ; a = (int) x ;` the **float** value is *cast* into an **int** and then that value is assigned to **a**
- `int a ; float x ; x = a ;` type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a **float**
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still "real division" but result is assigned to **int** after truncation
- `int a ; float x ; a = (int) x ;` the **float** value is *cast* into an **int** and then that value is assigned to **a**
- `int a ; float x ; x = a ;` type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still “real division” but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;` the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;` type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a `float`
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still “real division” but result is assigned to `int` after truncation
- `int a ; float x ; a = (int) x ;`
  the `float` value is *cast* into an `int` and then that value is assigned to `a`
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- **`int a ; a = 10 / 6 ;`** value of **`a`** ? 1; integer division
- **`float x ; x = 10 / 6 ;`** value of **`x`** ? 1.0; division is still integer only value is stored in a **`float`**
- **`float x ; x = 10.0 / 6.0 ;`** value of **`x`** ? 1.666666; “real division”
- **`int b ; x = 10.0 / 6.0 ;`** value of **`b`** ? 1; still “real division” but result is assigned to **`int`** after truncation
- **`int a ; float x ; a = (int) x ;`**
  the **`float`** value is *cast* into an **`int`** and then that value is assigned to **`a`**
- **`int a ; float x ; x = a ;`**
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **`a`** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **`x`** ? 1.0; division is still integer only value is stored in a **`float`**
- `float x ; x = 10.0 / 6.0 ;` value of **`x`** ? 1.666666; “real division”
- `int b ; x = 10.0 / 6.0 ;` value of **`b`** ? 1; still “real division” but result is assigned to **`int`** after truncation
- `int a ; float x ; a = (int) x ;`
  the **`float`** value is *cast* into an **`int`** and then that value is assigned to **`a`**
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Assignments

- *variable* = *expression*
- `int a ; a = 10 / 6 ;` value of **a** ? 1; integer division
- `float x ; x = 10 / 6 ;` value of **x** ? 1.0; division is still integer only value is stored in a **float**
- `float x ; x = 10.0 / 6.0 ;` value of **x** ? 1.666666; "real division"
- `int b ; x = 10.0 / 6.0 ;` value of **b** ? 1; still "real division" but result is assigned to **int** after truncation
- `int a ; float x ; a = (int) x ;`
  the **float** value is *cast* into an **int** and then that value is assigned to **a**
- `int a ; float x ; x = a ;`
  type casting still happens, but is done automatically

Short hands

- *variable = variable operator expression* may be written as
- *variable op= expression*
- `a = a + 14.3 ;` is equivalent to `a += 14.3 ;`
- Distraction for new programmers, better avoid (for now), but
- Need to know to understand programs written by others

Short hands

- *variable* = *variable operator expression* may be written as
- *variable op= expression*
- `a = a + 14.3 ;` is equivalent to `a += 14.3 ;`
- Distraction for new programmers, better avoid (for now), but
- Need to know to understand programs written by others

Short hands

- *variable = variable operator expression* may be written as
- *variable op= expression*
- `a = a + 14.3 ;` is equivalent to `a += 14.3 ;`
- Distraction for new programmers, better avoid (for now), but
- Need to know to understand programs written by others

Short hands (contd.)

- `a += b ;` /* equivalent to a = a + b */
- `a -= b ;` /* equivalent to a = a - b */
- `a *= b ;` /* equivalent to a = a * b */
- `a /= b ;` /* equivalent to a = a / b */
- `a &= b ;` /* equivalent to a = a & b (bit wise AND) */
- `a |= b ;` /* equivalent to a = a | b (bit) wise OR */
- `a ^= b ;` /* equivalent to a = a ˆ b (bit) wise XOR */

A useful syntax for small if constructions is the expression

b ? c : d / evaluates to c if b is true, and d otherwise */*

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

`++ --`

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

`++ --`

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

++ --

- `int a, b ;`
- `a = b ; b = b + 1 ;` may be written as
- `a = b++ ;` *post-increment* ; know, but avoid (for now)
- `b = b + 1; a = b ;` may be written as
- `a = ++b ;` *pre-increment*
- `a = b ; b = b - 1 ;` may be written as
- `a = b-- ;` *post-decrement*
- `b = b - 1; a = b ;` may be written as
- `a = --b ;` *pre-decrement*
- Not an aid to problem solving!

Side effects

- Consider the two statements: `x=a+1;` and `y=a++;`
- Both `x` and `y` have the same value
- Now consider the statements: `a+1; x=a+1; a++; y=a++;`
- `x` and `y` now have different values
- This is because the `++` (every pre/post – increment/decrement operator) changes the value of their operand
- This is called a side effect
- Thus these operators should be used only when this side effect is desired

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in ‘C’ is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in ‘C’ is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in ‘C’ is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in ‘C’ is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Associativity

- `1 + 2 + 3 = (1 + 2) + 3 = 5`
- `1 - 2 - 3 = (1 - 2) - 3 = -4`
- `1 - (2 - 3) = 2` (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Associativity

- **`1 + 2 + 3 = (1 + 2) + 3 = 5`**
- **`1 - 2 - 3 = (1 - 2) - 3 = -4`**
- **`1 - (2 - 3) = 2`** (not -4), associativity matters!

When $\oplus$ is left associative:

$a \oplus b \oplus c = (a \oplus b) \oplus c$

When $\oplus$ is right associative:

$a \oplus b \oplus c = a \oplus (b \oplus c)$

`2+3-4*5/6` ? 2 or 5, result is 2, BODMAS applies, but set of operators in 'C' is richer

Bit operators (to be covered later)

- **~** complement
- **«** *variable* « *n*, left shift *n* bits
- **»** *variable* » *n*, right shift *n* bits
- **&** bit wise AND
- **|** bit wise OR

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- `<< >>` binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- `<< >>` binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- `«  »` binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- `<< >>` binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- `« »` binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Precedence

- `() [] -> .` left to right
- `! ~ (bit) ++ -- - (unary) * (indirection) & (address-of) sizeof casts` right to left
- `* / %` binary, left to right
- `+ - (subtraction)` binary, left to right
- « » binary (bit), left to right
- `< <= >= >` binary, left to right
- `== !=` binary, left to right
- `& (bit)` binary, left to right
- `^ (bit)` binary, left to right
- `| (bit)` binary, left to right
- `&&` binary, left to right
- `||` binary, left to right
- `?:` binary, right to left
- `= += -= *=`, etc. binary, right to left
- `,` binary, left to right

Section outline

8 **Examples**

- Digits of a Number
- Area computations
- More straight line coding

Extracting units and tens values from a decimal number

- Let the number be n
- Units: $n \bmod 10$
- Hundreds: $(n/10) \bmod 10$

Extracting units and tens values from a decimal number

- Let the number be n
- Units: $n \bmod 10$
- Hundreds: $(n/10) \bmod 10$

Extracting units and tens values from a decimal number

- Let the number be n
- Units: $n \mod 10$
- Hundreds: $(n/10) \mod 10$

Program

Editor:

```
#include <stdio.h>

main() {
   int n, units, tens;

   printf ("enter an integer: ");
   scanf ("%d", &n);
   units = n % 10;
   tens = (n/10) % 10;
   printf ("number=%d, tens=%d, units=%d\n",
    n, tens, units);
}
```

Results

Compile and run:

```
$ cc digits.c -o digits
$ ./digits
enter an integer: 3453
number=3453, tens=5, units=3
```

Computing the area of a circle

- Let the radius be n
- Area: πr^2

Computing the area of a circle

- Let the radius be n
- Area: πr^2

Program

Editor:

```
#include <stdio.h>
#include <math.h>

main() {
   float r, area;

   printf ("enter the radius: ");
   scanf ("%f", &r);
   area = M_PI * r * r;
   printf ("radius=%f, area=%f\n", r, area);
}
```

Results

Compile and run:

```
$ cc circle.c -o circle
$ ./circle
enter the radius: 3.6
radius=3.600000, area=40.715038
```

Computing the area of an equilateral triangle

- Let the side be s
- Area: $\frac{s^2 sin(\pi/3)}{2}$

Computing the area of an equilateral triangle

- Let the side be s
- Area: $\frac{s^2 sin(\pi/3)}{2}$

Program

Editor:

```
#include <stdio.h>
#include <math.h>

main() {
    float s, area;

    printf ("enter the side: ");
    scanf ("%f", &s);
    area = 1.0/2.0 * s * s * sin(M_PI/3);
    printf ("side=%f, area=%f\n", s, area);
}
```

Results

Compile and run:

```
$ cc eqTri.c -o eqTri -lm
$ ./eqTri
enter the side: 10.0
side=10.000000, area=43.301270
```

More straight line coding

- Simple interest
- Compound interest
- Mortgage computation
- Solving a pair of linear simultaneous equations
- Finding the largest positive integer representable in the CPU

Syllabus (Theory)

Introduction to the Digital Computer;
Introduction to Programming – Variables, Assignment; Expressions; Input/Output;
Conditionals and Branching; Iteration;
Functions; Recursion; Arrays; Introduction to Pointers; Strings; Structures;
Introduction to Data-Procedure Encapsulation;
Dynamic allocation; Linked structures;
Introduction to Data Structure – Stacks and Queues; Searching and Sorting; Time and space requirements.

Part IV

CPU

Section outline

9. **Programmer's view of CPU**
   - Programming
   - ISA
   - Storage
   - Assembly
   - CPU operation
   - Instruction sequencing
   - Around the CPU

High-level versus low-level languages

- We have mentioned that ‘C’ is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: ‘C’ – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- cc: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that ‘C’ is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0’s and 1’s
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- cc: ‘C’ – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer’s view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- **cc**: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

High-level versus low-level languages

- We have mentioned that 'C' is a high-level programming language, also Java, C++, FORTRAN, and others
- High-level because they keep us away from then nitty-gritty details of programming the computer (its central processing unit)
- Computer has its own set of instructions that it understands – machine language – just a sequence of 0's and 1's
- Compiler translates high-level language programs to machine language, usually via the corresponding assembly language – little better for us
- `cc`: 'C' – compile → assembly language – assemble → machine language
- One-to-one correspondence (nearly) between assembly language of the machine (CPU) and the machine language of the CPU
- To understand, how a computer (CPU) works, we shall try to understand its working at the assembly language level
- The programmer's view of the CPU with its registers, memory and register addressing schemes and its instructions make up its Instruction Set Architecture (ISA)

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

Instruction set architecture (ISA) – Not for exams

The Instruction Set Architecture (ISA) is the part of the processor that is visible to a programmer – an abstract view of it

- *Registers* What registers are available for keeping data in the CPU (apart from the main memory, outside the CPU)?
- Can store integers, floating point numbers (usually) and other simple types of data
- How can data be *addressed*?
- We can usually refer to the registers as R1, R2, etc.
- We can usually refer to memory locations directly (such as 3072)
- Can we store an addresses in a register and then use it "indirectly" – put 3072 in R1 and use it via R1? – and so on
- *What can be done* within the CPU (by way of *CPU instructions*) – add data, move data between places, make decisions, jump to some instruction, etc

[Storage of variables]

Sum of two numbers revisited

Editor:

```
main() {
   int a=6;
   int b=14;
   int s;
   s=a+b;
}
```

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`a (=6)`	00000000	00000000	00000000	00000110
address	3075	3074	3073	**3072**
`b (=14)`	00000000	00000000	00000000	00001110
address	3079	3078	3077	**3076**
`s`	01010011	11001010	10101111	11010010
address	3083	3082	3081	**3080**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- Usual for declared to be allocated space in the (main) memory
- Allocated memory locations for `a`, `b` and `s` are depicted
- Locations for `a` and `b` are shown to contain their initial values
- Location for `s` is shown to contain a "garbage" value

What is there in the variables?

...	....	....	....	....
address	....	....	....	**....**
a (=6)	00000000	00000000	00000000	00000110
address	3075	3074	3073	**3072**
b (=14)	00000000	00000000	00000000	00001110
address	3079	3078	3077	**3076**
s	01010011	11001010	10101111	11010010
address	3083	3082	3081	**3080**
...	....	....	....	....
address	....	....	....	**....**

- Usual for declared to be allocated space in the (main) memory
- Allocated memory locations for `a`, `b` and `s` are depicted
- Locations for `a` and `b` are shown to contain their initial values
- Location for `s` is shown to contain a "garbage" value

What is there in the variables?

...	....	....	....	....
address	....	....	....	**....**
`a (=6)`	00000000	00000000	00000000	00000110
address	3075	3074	3073	**3072**
`b (=14)`	00000000	00000000	00000000	00001110
address	3079	3078	3077	**3076**
`s`	01010011	11001010	10101111	11010010
address	3083	3082	3081	**3080**
...	....	....	....	....
address	....	....	....	**....**

- Usual for declared to be allocated space in the (main) memory
- Allocated memory locations for `a`, `b` and `s` are depicted
- Locations for `a` and `b` are shown to contain their initial values
- Location for `s` is shown to contain a "garbage" value

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`a (=6)`	`00000000`	`00000000`	`00000000`	`00000110`
address	3075	3074	3073	**3072**
`b (=14)`	`00000000`	`00000000`	`00000000`	`00001110`
address	3079	3078	3077	**3076**
`s`	`01010011`	`11001010`	`10101111`	`11010010`
address	3083	3082	3081	**3080**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- Usual for declared to be allocated space in the (main) memory
- Allocated memory locations for `a`, `b` and `s` are depicted
- Locations for `a` and `b` are shown to contain their initial values
- Location for `s` is shown to contain a "garbage" value

Translated to assembly language

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;
  int b=14;
  int s;
  s=a+b;

}
```

- Suppose `a`, `b` and `s` are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;
  int b=14;
  int s;
  s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
   int a=6;          // LDI R1, 06;
   int b=14;
   int s;
   s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;        // LDI R1, 06; STM R1, 3072;
  int b=14;
  int s;
  s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
   int a=6;        // LDI R1, 06; STM R1, 3072;
   int b=14;       // LDI R1, 14;
   int s;
   s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;        // LDI R1, 06; STM R1, 3072;
  int b=14;       // LDI R1, 14; STM R1, 3076;
  int s;
  s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
   int a=6;          // LDI R1, 06; STM R1, 3072;
   int b=14;         // LDI R1, 14; STM R1, 3076;
   int s;            // Nothing to do
   s=a+b;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;          // LDI R1, 06; STM R1, 3072;
  int b=14;         // LDI R1, 14; STM R1, 3076;
  int s;            // Nothing to do
  s=a+b;            // LDM R1, 3072;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
   int a=6;          // LDI R1, 06; STM R1, 3072;
   int b=14;         // LDI R1, 14; STM R1, 3076;
   int s;            // Nothing to do
   s=a+b;            // LDM R1, 3072; LDM R2, 3076;

}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
  int a=6;          // LDI R1, 06; STM R1, 3072;
  int b=14;         // LDI R1, 14; STM R1, 3076;
  int s;            // Nothing to do
  s=a+b;            // LDM R1, 3072; LDM R2, 3076;
                    // ADD R3, R1, R2;
}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

Sum of two numbers revisited (contd.)

Editor:

```
main() {
   int a=6;        // LDI R1, 06; STM R1, 3072;
   int b=14;       // LDI R1, 14; STM R1, 3076;
   int s;          // Nothing to do
   s=a+b;          // LDM R1, 3072; LDM R2, 3076;
                   // ADD R3, R1, R2; STM R3, 3080;
}
```

- Suppose **a**, **b** and **s** are located in the main memory at addresses 3072, 3076 and 3080, respectively.
- LDI: LoaD Immediate operand
- STM: STore operand in Memory
- LDM: LoaD operand from Memory
- ADD: ADD last two registers and store in first

[Working of the ADD instruction]

How was the ADD done?

- The CPU has a component (Arithmetic Logic Unit (ALU)) that can perform arithmetic operations such as: addition, subtraction, multiplication and division
- Multiplication and division are more complex than addition and subtraction
- Not all CPUs have ALUs capable of multiplication and division
- ALU can also perform logical operations such a comparing two numbers and also performing bit wise operations on them
- Bit wise operations will be considered later

How was the ADD done?

- The CPU has a component (Arithmetic Logic Unit (ALU)) that can perform arithmetic operations such as: addition, subtraction, multiplication and division
- Multiplication and division are more complex than addition and subtraction
- Not all CPUs have ALUs capable of multiplication and division
- ALU can also perform logical operations such a comparing two numbers and also performing bit wise operations on them
- Bit wise operations will be considered later

How was the ADD done?

- The CPU has a component (Arithmetic Logic Unit (ALU)) that can perform arithmetic operations such as: addition, subtraction, multiplication and division
- Multiplication and division are more complex than addition and subtraction
- Not all CPUs have ALUs capable of multiplication and division
- ALU can also perform logical operations such a comparing two numbers and also performing bit wise operations on them
- Bit wise operations will be considered later

How was the ADD done?

- The CPU has a component (Arithmetic Logic Unit (ALU)) that can perform arithmetic operations such as: addition, subtraction, multiplication and division
- Multiplication and division are more complex than addition and subtraction
- Not all CPUs have ALUs capable of multiplication and division
- ALU can also perform logical operations such a comparing two numbers and also performing bit wise operations on them
- Bit wise operations will be considered later

How was the ADD done?

- The CPU has a component (Arithmetic Logic Unit (ALU)) that can perform arithmetic operations such as: addition, subtraction, multiplication and division
- Multiplication and division are more complex than addition and subtraction
- Not all CPUs have ALUs capable of multiplication and division
- ALU can also perform logical operations such a comparing two numbers and also performing bit wise operations on them
- Bit wise operations will be considered later

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Which instruction to execute?

- We knew which instruction was to be executed, but how does the CPU know?
- Instructions are also stored in memory in sequence – each instruction has an address
- A special CPU register, the program counter (PC) keeps tract of the instruction to be executed
- After an instruction at the memory location pointed to by the PC is fetched, the PC value is incremented properly to point to the next instruction
- JMP instructions cause new values to be loaded into the PC

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Test yourselves – Not for exams

- 06, 14 ? immediate operands
- R1, R2, R3 ? CPU registers
- 3072, 3076, 3080 ? addresses of memory locations (for `a`, `b` and `s`)
- LDI ? LoaD Immediate operand – CPU instruction
- STM ? STore operand in Memory – CPU instruction
- LDM ? LoaD operand from Memory – CPU instruction
- ADD ? ADD last two registers and store in first – CPU instruction
- LDI, STM, LDM, ADD – instruction pnemonic codes (instruction short forms)
- Contemporary CPUs have lots of instructions
- PC ? Program counter

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional “helper hardware” is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional “helper hardware” is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Beyond the main memory

- Program was magically there in the main memory
- How does it get there?
- How does the program receive user inputs?– those are not available in the main memory
- How does data appear on the screen? – not enough to store data in the main memory
- Additional "helper hardware" is needed – peripheral devices, which help the CPU to do input/output (i/o)
- Important i/o operations: reading and writing from the hard disk, receiving keystrokes from the keyboard, displaying characters on the terminal and others

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that “hand shaking” with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

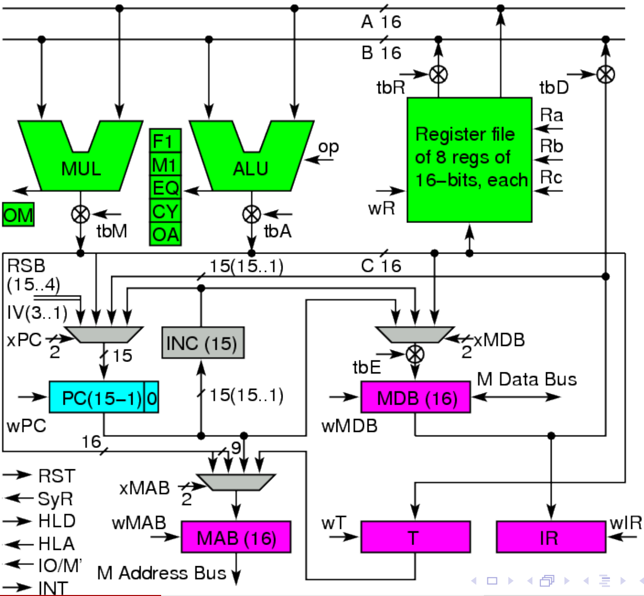
Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

Peripheral devices – Not for exams

- But how does the CPU communicate with peripheral devices?
- Special memory locations reserved to work with peripheral devices
- These locations are outside the main memory but are accessed by memory operations!
- These locations have special meaning associated with them
- For example, to print a character, the CPU could
  - check a specially designated memory location (1) to know that the device is ready to receive a character
  - then write the character to be output to another specially designated memory location (2)
  - Write a special code at the specially designated location (1) to indicate that there is new data to be output
  - The device would then know that it should now output the character and do its job
  - Note that "hand shaking" with the peripheral device is involved in this case
- I/O operations are involved, but this is the basic principle
- Efficient mechanisms have been evolved to conduct i/o operations

A classroom CPU design – Not for exams



Section outline

10 **Integer representation**

- Valuation scheme
- Decimal to binary
- Negative numbers
- Summary of NS
- Hexadecimal and octal

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- $0 \times 127 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- $0 \times 127 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- $0 \times 127 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+1\times2^1+0\times2^0$
- $0\times127+1\times64+1\times32+0\times16+1\times8+0\times4+1\times2+0\times1=106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+1\times2^1+0\times2^0$
- $0\times127+1\times64+1\times32+0\times16+1\times8+0\times4+1\times2+0\times1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$
- Its value will be defined as $b_{n-1} 2^{n-1} + b_{n-2} 2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- $0 \times 127 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Representation of Integers

- Mathematically, an integer can have an arbitrarily large value
- Representation on a computer is inherently finite
- Only a subset of integers can be directly represented
- We shall consider binary representation, using 0's and 1's
- A sequence of n binary bits will be numbered as $b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- Its value will be defined as $b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$
- Value of `0 1 1 0 1 0 1 0` ?
- $0\times 2^7 + 1\times 2^6 + 1\times 2^5 + 0\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0$
- $0\times 127 + 1\times 64 + 1\times 32 + 0\times 16 + 1\times 8 + 0\times 4 + 1\times 2 + 0\times 1 = 106$
- Binary number system is of base 2 or radix 2
- Bit position i has a weight of 2^i

Decimal to binary

- Binary of 106?
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106?
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106?
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits (0..($k-1$))
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits (0..($k-1$))
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Decimal to binary

- Binary of 106? 0 1 1 0 1 0 1 0
- By repeated division

	106	Remainder
After division by 2	53	0 (b_0)
After division by 2	26	1 (b_1)
After division by 2	13	0 (b_2)
After division by 2	6	1 (b_3)
After division by 2	3	0 (b_4)
After division by 2	1	1 (b_5)
After division by 2	0	1 (b_6)
After division by 2	0	0 (b_7)

- Divide k times for a binary representation in k-bits $(0..(k-1))$
- Maximum value of a binary number of k-bits: $2^k - 1$ (255, if $k = 8$)
- What if original number is larger than $2^k - 1$ (say 1000, for $k = 8$)?
- Coverted value of binary number = (Original number) modulo 2^k

Simple view of modulo 2^k

- $N \equiv b_{n-1} b_{n-2} \ldots b_k \ldots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k[b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \ldots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \ldots)$
- Only consider the bits at positions $0..(k-1)$

Simple view of modulo 2^k

- $N \equiv b_{n-1}b_{n-2}\ldots b_k \ldots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k[b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \ldots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \ldots)$
- Only consider the bits at positions $0..(k-1)$

Simple view of modulo 2^k

- $N \equiv b_{n-1}b_{n-2}\dots b_k \dots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \dots + b_k 2^k + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k[b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \dots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \dots)$
- Only consider the bits at positions $0..(k-1)$

Simple view of modulo 2^k

- $N \equiv b_{n-1}b_{n-2}\ldots b_k \ldots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k[b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \ldots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \ldots)$
- Only consider the bits at positions $0..(k-1)$

Simple view of modulo 2^k

- $N \equiv b_{n-1} b_{n-2} \ldots b_k \ldots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k [b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \ldots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \ldots)$
- Only consider the bits at positions $0..(k-1)$

Simple view of modulo 2^k

- $N \equiv b_{n-1} b_{n-2} \dots b_k \dots b_2 b_1 b_0$ has value

$$
\begin{aligned}
N &= b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \dots + b_k 2^k + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
&= 2^k[b_{n-1}2^{n-1-k} + b_{n-2}2^{n-1-k} + \dots + b_k] + b_2 2^2 + b_1 2^1 + b_0 2^0 \\
N \bmod 2^k &= b_{k-1}2^{k-1} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0
\end{aligned}
$$

- Simple view: just disregard all bits from position k and beyond $(k, k+1, k+2, \dots)$
- Only consider the bits at positions $0..(k-1)$

Decimal to binary (contd.)

- Binary of 1000?
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000?
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000?
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 0 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 1 0 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 0 1 0 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 1 0 1 0 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 1 1 0 1 0 0 0
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 1 1 1 0 1 0 0 0 $\equiv$ 232
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Decimal to binary (contd.)

- Binary of 1000? 1 1 1 0 1 0 0 0 $\equiv$ 232
- By repeated division

	1000	Remainder
After division by 2	500	0 (b_0)
After division by 2	250	0 (b_1)
After division by 2	125	0 (b_2)
After division by 2	62	1 (b_3)
After division by 2	31	0 (b_4)
After division by 2	15	1 (b_5)
After division by 2	7	1 (b_6)
After division by 2	3	1 (b_7)

- 1000 modulo 2^8 (remainder of dividing 1000 by 256) = 232

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- “Eats up” one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k - 1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

Negative numbers

- Only positive numbers represented, so far
- Possible to designate one bit to represent sign
  `0 1 1 0 1 0 1 0` $\equiv$ +106, `1 1 1 0 1 0 1 0` $\equiv$ -106 – intuitive!
- Sign bit does not contribute to the value of the number
- "Eats up" one bit, out of the k bits for representing the sign, only the remaining $k-1$ bits contribute to the value of the number
- Binary arithmetic on signed-magnitude numbers more complex
- How many distinct values can be represented in the signed-magnitude of k-bits? $2^k - 1$ (why?)
- Because zero has two representations

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2b'_1b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1}+(1-b_{n-2})2^{n-1}+\ldots+(1-b_2)2^2+(1-b_1)2^1+(1-b_0)2^0$
- $2^{n-1}+2^{n-1}+\ldots+2^2+2^1+2^0-(b_{n-1}2^{n-1}+b_{n-2}2^{n-1}+\ldots+b_22^2+b_12^1+b_02^0)$
- 2^k-1-N
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in 2^k-1-N?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2 b'_1 b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1}+(1-b_{n-2})2^{n-1}+\ldots+(1-b_2)2^2+(1-b_1)2^1+(1-b_0)2^0$
- $2^{n-1}+2^{n-1}+\ldots+2^2+2^1+2^0-(b_{n-1}2^{n-1}+b_{n-2}2^{n-1}+\ldots+b_2 2^2+b_1 2^1+b_0 2^0)$
- 2^k-1-N
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in 2^k-1-N?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2b'_1b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1}+(1-b_{n-2})2^{n-1}+\ldots+(1-b_2)2^2+(1-b_1)2^1+(1-b_0)2^0$
- $2^{n-1}+2^{n-1}+\ldots+2^2+2^1+2^0-(b_{n-1}2^{n-1}+b_{n-2}2^{n-1}+\ldots+b_22^2+b_12^1+b_02^0)$
- 2^k-1-N
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in 2^k-1-N?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2 b'_1 b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1} + (1-b_{n-2})2^{n-1} + \ldots + (1-b_2)2^2 + (1-b_1)2^1 + (1-b_0)2^0$
- $2^{n-1} + 2^{n-1} + \ldots + 2^2 + 2^1 + 2^0 - (b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0)$
- $2^k - 1 - N$
- $106 \equiv$ 0 1 1 0 1 0 1 0
- 1's complement of $106 \equiv$ 1 0 0 1 0 1 0 1
- Possible to get rid of the (-1) in $2^k - 1 - N$?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$
- 1's complement: $b'_{n-1} b'_{n-2} \ldots b'_2 b'_1 b'_0$
  $(1 - b_{n-1})(1 - b_{n-2}) \ldots (1 - b_2)(1 - b_1)(1 - b_0)$
- Its value will be: $(1 - b_{n-1})2^{n-1} + (1 - b_{n-2})2^{n-1} + \ldots + (1 - b_2)2^2 + (1 - b_1)2^1 + (1 - b_0)2^0$
- $2^{n-1} + 2^{n-1} + \ldots + 2^2 + 2^1 + 2^0 - (b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0)$
- $2^k - 1 - N$
- $106 \equiv$ **0 1 1 0 1 0 1 0**
- 1's complement of $106 \equiv$ **1 0 0 1 0 1 0 1**
- Possible to get rid of the (-1) in $2^k - 1 - N$?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2b'_1b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1}+(1-b_{n-2})2^{n-1}+\ldots+(1-b_2)2^2+(1-b_1)2^1+(1-b_0)2^0$
- $2^{n-1}+2^{n-1}+\ldots+2^2+2^1+2^0-(b_{n-1}2^{n-1}+b_{n-2}2^{n-1}+\ldots+b_22^2+b_12^1+b_02^0)$
- 2^k-1-N
- $106 \equiv$ 0 1 1 0 1 0 1 0
- 1's complement of $106 \equiv$ 1 0 0 1 0 1 0 1
- Possible to get rid of the (-1) in 2^k-1-N?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1} b_{n-2} \dots b_2 b_1 b_0$
- 1's complement: $b'_{n-1} b'_{n-2} \dots b'_2 b'_1 b'_0$
  $(1-b_{n-1})(1-b_{n-2})\dots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1} + (1-b_{n-2})2^{n-1} + \dots + (1-b_2)2^2 + (1-b_1)2^1 + (1-b_0)2^0$
- $2^{n-1} + 2^{n-1} + \dots + 2^2 + 2^1 + 2^0 - (b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0)$
- $2^k - 1 - N$
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in $2^k - 1 - N$?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2b_1b_0$
- 1's complement: $b'_{n-1}b'_{n-2}\ldots b'_2b'_1b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1}+(1-b_{n-2})2^{n-1}+\ldots+(1-b_2)2^2+(1-b_1)2^1+(1-b_0)2^0$
- $2^{n-1}+2^{n-1}+\ldots+2^2+2^1+2^0-(b_{n-1}2^{n-1}+b_{n-2}2^{n-1}+\ldots+b_22^2+b_12^1+b_02^0)$
- 2^k-1-N
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in 2^k-1-N?

1's complement operation

- Definition is as follows:
- Given number: $N \equiv b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$
- 1's complement: $b'_{n-1} b'_{n-2} \ldots b'_2 b'_1 b'_0$
  $(1-b_{n-1})(1-b_{n-2})\ldots(1-b_2)(1-b_1)(1-b_0)$
- Its value will be: $(1-b_{n-1})2^{n-1} + (1-b_{n-2})2^{n-1} + \ldots + (1-b_2)2^2 + (1-b_1)2^1 + (1-b_0)2^0$
- $2^{n-1} + 2^{n-1} + \ldots + 2^2 + 2^1 + 2^0 - (b_{n-1}2^{n-1} + b_{n-2}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0)$
- $2^k - 1 - N$
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 1's complement of $106 \equiv$ `1 0 0 1 0 1 0 1`
- Possible to get rid of the (-1) in $2^k - 1 - N$?

2's complement operation

Definition (2's complement)

The two's complement of a binary number is defined as the value obtained by subtracting that number from a large power of two (specifically, from 2^n for an n-bit two's complement)

- Given number: $N \equiv b_{n-1}b_{n-2}\dots b_2 b_1 b_0$
- 2's complement: 1's complement, then increment
- $b'_{n-1}b'_{n-2}\dots b'_2 b'_1 b'_0 + 1$
- $2^n - 1 - N + 1 = 2^n - N$
- $106 \equiv$ 0 1 1 0 1 0 1 0
- 2's complement of $106 \equiv$ 1 0 0 1 0 1 1 0
- The MSB indicates the sign, anyway

2's complement operation

Definition (2's complement)

The two's complement of a binary number is defined as the value obtained by subtracting that number from a large power of two (specifically, from 2^n for an n-bit two's complement)

- Given number: $N \equiv b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$
- 2's complement: 1's complement, then increment
- $b'_{n-1} b'_{n-2} \ldots b'_2 b'_1 b'_0 + 1$
- $2^n - 1 - N + 1 = 2^n - N$
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 2's complement of $106 \equiv$ `1 0 0 1 0 1 1 0`
- The MSB indicates the sign, anyway

2's complement operation

Definition (2's complement)

The two's complement of a binary number is defined as the value obtained by subtracting that number from a large power of two (specifically, from 2^n for an n-bit two's complement)

- Given number: $N \equiv b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$
- 2's complement: 1's complement, then increment
- $b'_{n-1}b'_{n-2}\ldots b'_2 b'_1 b'_0 + 1$
- $2^n - 1 - N + 1 = 2^n - N$
- $106 \equiv$ `0 1 1 0 1 0 1 0`
- 2's complement of $106 \equiv$ `1 0 0 1 0 1 1 0`
- The MSB indicates the sign, anyway

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \bmod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 = $\begin{array}{cccccccccl} & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{(modulo } 2^8\text{)} \end{array}$

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \mod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 =

	0	1	1	0	1	0	1	0	
+	1	0	0	1	0	1	1	0	
	0	0	0	0	0	0	0	0	(modulo 2^8)

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \bmod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 =

	0	1	1	0	1	0	1	0	
+	1	0	0	1	0	1	1	0	
	0	0	0	0	0	0	0	0	(modulo 2^8)

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \bmod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 =

	0	1	1	0	1	0	1	0	
+	1	0	0	1	0	1	1	0	
	0	0	0	0	0	0	0	0	(modulo 2^8)

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \bmod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 =

$$
\begin{array}{cccccccccl}
 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{(modulo } 2^8\text{)}
\end{array}
$$

Subtraction of numbers

- Let the numbers be M and N (represented in k-bits), $M - N$=?
- Let's add 2's complemnent of N to M: $M + 2^k - N$
- Since the representation is in k-bits, the result is inherently modulo 2^k
- Hence, $M + 2^k - N \equiv M - N \bmod 2^k$ (why?)
- Subtraction is achieved by adding the 2's complement of the subtrahend (N) to the minuend (M)
- 106 - 106 =

$$\begin{array}{rcccccccc} & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \text{ (modulo } 2^8\text{)}$$

[Summary of number systems]

Comparison of the representations (8-bit)

Dec	s/m	1's cmp	2's cmp
+127	01111111	01111111	01111111
...	. . .	. . .	. . .
+1	00000001	00000001	00000001
0	00000000	00000000	00000000
0	10000000	11111111	00000000
-1	10000001	11111110	11111111
...	. . .	. . .	. . .
-127	01111111	10000000	10000001
-128	---	---	10000000
	$2^k - 1$	$2^k - 1$	2^k

Example of subtraction

106-11 (in 8-bits)

Binary of 11:	0 0 0 0 1 0 1 1
2's complement of 11:	1 1 1 1 0 1 0 0 + 1
2's complement representation of -11:	1 1 1 1 0 1 0 1
Binary of 106:	0 1 1 0 1 0 1 0
+ 2's complement of 11:	1 1 1 1 0 1 0 1
106 - 11 = 95:	0 1 0 1 1 1 1 1

NB

- 2's complement represenation: It is scheme for representing 0, +ve and -ve numbers
- 2's complement of a given number: It is an operation (bitwise complementation followed by addition of 1 (increment)) defined on a given number represented in 2's complement form

Example of adding two 2's complement numbers

(-106) + (-11) (in 8-bits)

Binary of 106:	0 1 1 0 1 0 1 0
2's complement of 106:	1 0 0 1 0 1 0 1 + 1
2's complement representation of -106:	1 0 0 1 0 1 1 0
Binary of 11:	0 0 0 0 1 0 1 1
2's complement of 11:	1 1 1 1 0 1 0 0 + 1
2's complement representation of -11:	1 1 1 1 0 1 0 1
2's complement of 106:	1 0 0 1 0 1 1 0
+ 2's complement of 11:	1 1 1 1 0 1 0 1
(-106) + (-11) = -117:	1 0 0 0 1 0 1 1

Check the result:

2's complement of -117:	0 1 1 1 0 1 0 0 + 1
2's complement representation of 117:	0 1 1 1 0 1 0 1 (okay)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` ≡ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111 + 1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Problems with Represenation

- 8-bit 2's complement representation of -128? `10000000`
- 2's complement of -128 (8-bit representation)?
- `01111111` + `1` = ? `10000000` (inconsistent)
- 256 - 128 = 128
- (256 - 128) % 256 = 128
- 8-bit 2's complement representation of 127? `01111111`
- 127 + 1 (in 8-bits) ?
- `10000000` $\equiv$ -128
- Addition of positive and negative numbers never result in a wrong answer
- If sum of two positive numbers is less than zero, then there is an error (overflow)
- If sum of two negative numbers is greater than zero, then also there is an error (overflow)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106?
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106?
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106?
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106? A(1010)
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106? 6(0110) A(1010)
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to hexadecimal (base 16)

- Hexadecimal of 106? 0x6A: 6(0110) A(1010)
- By repeated division

	106	Remainder
After division by 2^4	6	10 (A/1010)
After division by 2^4	0	6 (6/0110)

- Relationship between binary and hexadecimal (hex): just group four binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106?
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106?
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106?
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106? 2(010)
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106? 5(101) 2(010)
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106? 1(001) 5(101) 2(010)
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Decimal to octal (base 8)

- Octal of 106? 0152: 1(001) 5(101) 2(010)
- By repeated division

	106	Remainder
After division by 2^3	13	2 (2/010)
After division by 2^3	1	5 (5/101)
After division by 2^3	0	1 (1/001)

- Relationship between binary and octal (oct): just group three binary bits from the right (least significant bit position – LSB)

Sum program revisited

Edit **sum.c** so that it as follows:

Editor: Dangers of a leading 0

```
#include <stdio.h>
main() {
  int a=006; // octal of 6
  int b=014; // octal of 12
  int s;

  s=a+b;
  printf ("sum=%d\n", s);
}
```

Compile and run:

```
$ cc sum.c -o sum
$ ./sum
sum=18
```

Section outline

11 **Real number representation**
- Valuation
- Converting fractions
- IEEE 754
- Non-associative addition
- Special IEEE754 numbers

(Approximate) representation of real numbers

- Suppose we have: `01101010.110101`
- `01101010` $\equiv 106$
- `.110101` $\equiv 1 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} + 0 \times \frac{1}{2^3} + 1 \times \frac{1}{2^4} + 0 \times \frac{1}{2^5} + 1 \times \frac{1}{2^6} = .828125$
- `01101010.110101` $\equiv 106.828125$

(Approximate) representation of real numbers

- Suppose we have: `01101010.110101`
- `01101010` $\equiv$ 106
- `.110101`
  $\equiv 1 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} + 0 \times \frac{1}{2^3} + 1 \times \frac{1}{2^4} + 0 \times \frac{1}{2^5} + 1 \times \frac{1}{2^6} = .828125$
- `01101010.110101` $\equiv$ 106.828125

(Approximate) representation of real numbers

- Suppose we have: `01101010.110101`
- `01101010` $\equiv 106$
- `.110101`
  $\equiv 1 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} + 0 \times \frac{1}{2^3} + 1 \times \frac{1}{2^4} + 0 \times \frac{1}{2^5} + 1 \times \frac{1}{2^6} = .828125$
- `01101010.110101` $\equiv 106.828125$

(Approximate) representation of real numbers

- Suppose we have: `01101010.110101`
- `01101010` $\equiv 106$
- `.110101`
  $\equiv 1 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} + 0 \times \frac{1}{2^3} + 1 \times \frac{1}{2^4} + 0 \times \frac{1}{2^5} + 1 \times \frac{1}{2^6} = .828125$
- `01101010.110101` $\equiv 106.828125$

(Approximate) representation of real numbers (contd.)

- Binary of 0.2?
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2?
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2?
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0 0
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0 0 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0 0 1 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0 0 1 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

(Approximate) representation of real numbers (contd.)

- Binary of 0.2? 0.0 0 1 1 0 0 1 1
- By repeated multiplication

	fractional part	integral part
	0.2	
After multiplication by 2	0.4	0 (b_{-1})
After multiplication by 2	0.8	0 (b_{-2})
After multiplication by 2	0.6	1 (b_{-3})
After multiplication by 2	0.2	1 (b_{-4})
After multiplication by 2	0.4	0 (b_{-5})
After multiplication by 2	0.8	0 (b_{-6})
After multiplication by 2	0.6	1 (b_{-7})
After multiplication by 2	0.2	1 (b_{-8})

- Representation of 0.2 is non-terminating
- Several representation options, normalised representation required

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- `01101010.110101` $= 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- **01101010.110101** $\equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- `01101010.110101` $\equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are 0's, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- `01101010.110101` $\equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- $\mathtt{01101010.110101} \equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- **`01101010.110101`** $\equiv 1.101010110101 \times 2^6$
- Since a **`1`** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are 0's, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- **01101010.110101** $\equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are 0's, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- **01101010.110101** $\equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

IEEE 754

- $106.828125 = 1.06828125 \times 10^2$
- $\mathtt{01101010.110101} \equiv 1.101010110101 \times 2^6$
- Since a **1** is always present in the normalised form, it need not be represented explicitly – it is implicitly present
- A standardised approximate 32-bit representation of real numbers is the IEEE754 standard
- $s\ e_7 e_6 \ldots e_1 e_0\ m_{22} m_{21} \ldots m_1 m_0$
- Its value is: $(1 - 2 \times s) \times (1.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
- Exponent is in *excess 127* form, exponent of 0 is represented as 127 (in binary)
- Storing a biased exponent before a normalized mantissa means we can compare IEEE values as if they were signed integers.
- When all the exponent bits are **0**'s, the numbers are no longer normalized
- Denormal value: $(1 - 2 \times s) \times (0.m_{22} m_{21} \ldots m_1 m_0)_2 \times 2^{-126}$

A Sample Conversion

- What is the decimal value of the following IEEE number?
  10111110011000000000000000000000
- Work on the fields individually
  - The sign bit s is 1.
  - The e field contains 01111100 = 124.
  - The *mantissa* is 0.11000... = 0.75.
- Plug these values of s, e and f into our formula:
  $(1 - 2 \times s) \times (1.m_{22}m_{21}\ldots m_1 m_0)_2 \times 2^{[(e_7 e_6 \ldots e_1 e_0)_2 - 127]}$
  This gives us
  $(1-2) * (1 + 0.75) * 2^{124-127} = (-1.75 \times 2^{-3}) = -0.21875.$

A Pitfall: Addition is not Associative

$$x = -2.5 \times 10^{40}$$
$$y = 2.5 \times 10^{40}$$
$$z = 1.0$$

$$\begin{aligned} x + (y + z) &= -2.5 \times 10^{40} + (2.5 \times 10^{40} + 1.0) \\ &= -2.5 \times 10^{40} + 2.5 \times 10^{40} \\ &= 0 \end{aligned}$$

$$\begin{aligned} (x + y) + z &= (-2.5 \times 10^{40} + 2.5 \times 10^{40}) + 1.0 \\ &= 0 + 1.0 \\ &= 1.0 \end{aligned}$$

Requires extreme alertness of the programmer

Special IEEE754 numbers

+ infinity `0 11111111 000 0000 00000000 00000000` +Inf

- infinity `1 11111111 000 0000 00000000 00000000` -Inf

Not a number `? 11111111 nnn nnnn nnnnnnnn nnnnnnnn` NaN

`nnn nnnn nnnnnnnn nnnnnnnn` is any non-zero sequence of bits

Syllabus Details of IEEE754, excess 127 exponent, implicit 1 in mantissa

Special IEEE754 numbers should be known

Advanced Denormal forms

Special IEEE754 numbers

+ infinity `0 11111111 000 0000 00000000 00000000` +Inf

- infinity `1 11111111 000 0000 00000000 00000000` -Inf

Not a number `? 11111111 nnn nnnn nnnnnnnn nnnnnnnn` NaN

`nnn nnnn nnnnnnnn nnnnnnnn` is any non-zero sequence of bits

Syllabus Details of IEEE754, excess 127 exponent, implicit 1 in mantissa

Special IEEE754 numbers should be known

Advanced Denormal forms

Special IEEE754 numbers

+ infinity `0 11111111 000 0000 00000000 00000000` +Inf

- infinity `1 11111111 000 0000 00000000 00000000` -Inf

Not a number `? 11111111 nnn nnnn nnnnnnnn nnnnnnnn` NaN

`nnn nnnn nnnnnnnn nnnnnnnn` is any non-zero sequence of bits

Syllabus Details of IEEE754, excess 127 exponent, implicit 1 in mantissa

Special IEEE754 numbers should be known

Advanced Denormal forms

Special IEEE754 numbers

+ infinity `0 11111111 000 0000 00000000 00000000` +Inf

- infinity `1 11111111 000 0000 00000000 00000000` -Inf

Not a number `? 11111111 nnn nnnn nnnnnnnn nnnnnnnn` NaN

`nnn nnnn nnnnnnnn nnnnnnnn` is any non-zero sequence of bits

Syllabus Details of IEEE754, excess 127 exponent, implicit 1 in mantissa

Special IEEE754 numbers should be known

Advanced Denormal forms

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Comparison of real numbers

- Real numbers, as they are represented, often have errors in them
- Equality test of real numbers is risky – we had done it while making decisions on the sign of the discriminant, earlier
- Better way: Define a suitably small constant with a sensible name (say EPSILON) and then carry out the check
- `#define EPSILON 1.0E-8`
- Faulty: `if (d==0) { ... }`
- Better: `if (d<EPSILON && d>-EPSILON) { ... }`
- Likely to make mistakes on repeated use, better define a macro
- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- With macro: `if (isZR(d)) { ... }`

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- ```
  #define isZR(x) {int _y=x; \
        (_y<EPSILON && _y>-EPSILON)}
  ```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!
  ```

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- ```
  #define isZR(x) {int _y=x; \
       (_y<EPSILON && _y>-EPSILON)}
  ```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!
  ```

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- `#define isZR(x) {int _y=x; \`
  `(_y<EPSILON && _y>-EPSILON)}`
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- ```
  #define isZR(x) {int _y=x; \
      (_y<EPSILON && _y>-EPSILON)}
  ```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!
  ```

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- ```
  #define isZR(x) {int _y=x; \
       (_y<EPSILON && _y>-EPSILON)}
  ```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!
  ```

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
- ```
  #define isZR(x) {int _y=x; \
       (_y<EPSILON && _y>-EPSILON)}
  ```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!
  ```

Caution with macros

- `#define isZR(x) (x)<EPSILON && (x)>-EPSILON`
- What will be the expansion of `isZR(y++)` ?
- `(y++)<EPSILON && (y++)>-EPSILON`
- `y` is incremented twice
- A safer version of the `isZR` macro?
-
```
#define isZR(x) {int _y=x; \
     (_y<EPSILON && _y>-EPSILON)}
```
- Scope of `_y` is restricted to the block
- What will be the expansion of `isZR(y++)` now?
- Try it out to check if it works!

Section outline

12 **Elementary data types**

- Integer variants
- Size of datatypes
- Portability

Elementary data types

- Integers in 32-bits or four bytes:`int`
- Reals in 32-bit or four bytes: `float`
- Characters in 8-bits or one byte:`char`
- Real variants: `float`, `double`, `long double`
- precision(`long double`) $\geq$ precision(`double`) $\geq$ precision(`float`)
- Printing: `float`, `double`: `%f`; `long double`: `%lf`

Elementary data types

- Integers in 32-bits or four bytes:`int`
- Reals in 32-bit or four bytes: `float`
- Characters in 8-bits or one byte:`char`
- Real variants: `float`, `double`, `long double`
- precision(`long double`) $\geq$ precision(`double`) $\geq$ precision(`float`)
- Printing: `float`, `double`: `%f`; `long double`: `%lf`

Elementary data types

- Integers in 32-bits or four bytes:`int`
- Reals in 32-bit or four bytes: `float`
- Characters in 8-bits or one byte:`char`
- Real variants: `float`, `double`, `long double`
- precision(`long double`) $\geq$ precision(`double`) $\geq$ precision(`float`)
- Printing: `float`, `double`: `%f`; `long double`: `%lf`

Elementary data types

- Integers in 32-bits or four bytes:**`int`**
- Reals in 32-bit or four bytes: **`float`**
- Characters in 8-bits or one byte:**`char`**
- Real variants: **`float`**, **`double`**, **`long double`**
- precision(**`long double`**) $\geq$ precision(**`double`**) $\geq$ precision(**`float`**)
- Printing: **`float`**, **`double`**: `%f`; **`long double`**: `%lf`

Elementary data types

- Integers in 32-bits or four bytes:**`int`**
- Reals in 32-bit or four bytes: **`float`**
- Characters in 8-bits or one byte:**`char`**
- Real variants: **`float`**, **`double`**, **`long double`**
- precision(**`long double`**) $\geq$ precision(**`double`**) $\geq$ precision(**`float`**)
- Printing: **`float`**, **`double`**: **`%f`**; **`long double`**: **`%lf`**

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword `signed` is redundant and can be dropped
- Printing: `signed int`, `short`, `char`: `%d`
- `unsigned int`, `unsigned short`, `unsigned char`: `%u`
- `int`, `short`, `char`: `%x` or `%o`
- `signed long int`: `%d`
- `unsigned long int`: `%lu`
- `long int`: `%lx` or `%lo`

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- **`signed long int`**: **`%d`**
- **`unsigned long int`**: **`%lu`**
- **`long int`**: **`%lx`** or **`%lo`**

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- **`signed long int`**: **`%d`**
- **`unsigned long int`**: **`%lu`**
- **`long int`**: **`%lx`** or **`%lo`**

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- `signed long int`: `%d`
- `unsigned long int`: `%lu`
- `long int`: `%lx` or `%lo`

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- **`signed long int`**: **`%d`**
- **`unsigned long int`**: **`%lu`**
- **`long int`**: **`%lx`** or **`%lo`**

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- **`signed long int`**: **`%d`**
- **`unsigned long int`**: **`%lu`**
- **`long int`**: **`%lx`** or **`%lo`**

Integer variants

- Integer variants: **`unsigned short int`**, **`unsigned int`**, **`unsigned long int`**, **`signed short int`**, **`signed int`**, **`signed long int`**
- The keyword **`signed`** is redundant and can be dropped
- Printing: **`signed int`**, **`short`**, **`char`**: **`%d`**
- **`unsigned int`**, **`unsigned short`**, **`unsigned char`**: **`%u`**
- **`int`**, **`short`**, **`char`**: **`%x`** or **`%o`**
- **`signed long int`**: **`%d`**
- **`unsigned long int`**: **`%lu`**
- **`long int`**: **`%lx`** or **`%lo`**

`sizeof`

- `sizeof(`*typeName*`)`
- `sizeof(`*varName*`)`
- Not exactly a function call – handled by compiler to substitute correct value
- `int s;`
- `sizeof(int)` is 4
- `sizeof(s)` is 4

Portability

- High-level languages are meant to be portable – should compile and run on any platform
- Strong and machine independent datatypes help to attain program portability
- Unfortunately, the ‘C’ language is not the best example of a portable high-level programming language
- Functional programming languages such as SML have better features, but these are not commercially successful

Syllabus (Theory)

Introduction to the Digital Computer;
Introduction to Programming – Variables, Assignment; Expressions; Input/Output;
Conditionals and Branching; Iteration;
Functions; Recursion; Arrays; Introduction to Pointers; Strings; Structures;
Introduction to Data-Procedure Encapsulation;
Dynamic allocation; Linked structures;
Introduction to Data Structure – Stacks and Queues; Searching and Sorting; Time and space requirements.

Part V

Branching and looping

Section outline

13 Decision Making

- Conditionals
- Dangling else
- Condition evaluation
- Comma operator
- Switching
- Simple RDs

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0$, $a \neq 0$, a, b, c are real

Formula for roots: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Roots of a quadratic equation

Equation: $ax^2 + bx + c = 0, a \neq 0, a, b, c$ are real

Formula for roots: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Discriminant: $b^2 - 4ac$

The roots are classified as one of the following three cases, depending on the value of the discriminant:

zero Roots are equal

positive Roots are distinct and real

negative Roots are complex conjugates

Depending on the particular condition, (slightly) different computations need to be performed

Program

Editor:

```
#include <stdio.h>
#include <math.h>
main() {
float a, b, c, d;
printf ("enter a, b, c: "); scanf("%f%f%f", &a, &b, &c);
d = b*b - 4*a*c ; // the discriminant
 if (d == 0) { // roots are equal
  float r = -b/(2*a) ;
  printf ("equal roots: %e\n", r);
 } else if (d > 0) { // roots are real
  float d_root = sqrt(d);
  float r_1 = (-b + d_root) / (2*a) ;
  float r_2 = (-b - d_root) / (2*a) ;
  printf ("real roots: %e and %e\n", r_1, r_2);
 } else { // roots are complex
  float d_root = sqrt(-d);
  float r = -b / (2*a) ;
  float c = d_root / (2*a) ;
  printf ("complex roots:\n %e+i%e and\n %e-i%e\n", r, c, r, c);
 }
}
```

Results

Compile and run:

```
$ cc quadratic.c -o quadratic -lm
$ ./quadratic
enter a, b, c: 1 2 1
equal roots: -1.000000e+00
$ ./quadratic
enter a, b, c: 1 2 0
real roots: 0.000000e+00 and -2.000000e+00
$ ./quadratic
enter a, b, c: 1 1 1
complex roots:
  -5.000000e-01+i8.660254e-01 and
  -5.000000e-01-i8.660254e-01
```

Greater of two numbers

- Numbers are: a and b
- Let m be $\max(a, b)$ (in a mathematical sense)

Computation of $m = \max(a, b)$

```
if (a >= b) { // a is greater (or equal to)
  m = a ;
} else { // b is greater
  m = b ;
}
```

Shorthand for $m = \max(a, b)$

```
m = (a>=b) ? a : b ;
```

Greater of two numbers

- Numbers are: a and b
- Let m be $\max(a, b)$ (in a mathematical sense)

Computation of $m = \max(a, b)$

```
if (a >= b) { // a is greater (or equal to)
 m = a ;
} else { // b is greater
 m = b ;
}
```

Shorthand for $m = \max(a, b)$

```
m = (a>=b) ? a : b ;
```

Greater of two numbers

- Numbers are: a and b
- Let m be $\max(a, b)$ (in a mathematical sense)

Computation of $m = \max(a, b)$

```
if (a >= b) { // a is greater (or equal to)
 m = a ;
} else { // b is greater
 m = b ;
}
```

Shorthand for $m = \max(a, b)$

```
m = (a>=b) ? a : b ;
```

Greater of two numbers

- Numbers are: a and b
- Let m be $\max(a, b)$ (in a mathematical sense)

Computation of $m = \max(a, b)$

```
if (a >= b) { // a is greater (or equal to)
 m = a ;
} else { // b is greater
 m = b ;
}
```

Shorthand for $m = \max(a, b)$

```
 m = (a>=b) ? a : b ;
```

Greatest of three numbers

- Numbers are: a, b and c
- Let m be $\max(a, b)$ (in a mathematical sense) ,
- then $\max(m, c)$ will be the greatest of the three numbers

Greatest of three numbers

- Numbers are: a, b and c
- Let m be $\max(a, b)$ (in a mathematical sense) ,
- then $\max(m, c)$ will be the greatest of the three numbers

Greatest of three numbers

- Numbers are: a, b and c
- Let m be $\max(a, b)$ (in a mathematical sense) ,
- then $\max(m, c)$ will be the greatest of the three numbers

Program

Editor:

```
#include <stdio.h>
main() {
  int a, b, c, max_now;
  printf("enter a, b and c: ");
  scanf ("%d%d%d", &a, &b, &c);
  max_now = a >= b ? a : b ; // greater of a and b
  max_now = c >= max_now ? c : max_now ; // it is now max
  printf ("greatest of a, b, c: %d\n", max_now);
}
```

Results

Compile and run:

```
$ ./greatest
enter a, b and c: 32 -45 36
greatest of a, b, c: 36
```

Syntax – if

If-statement

statement ::= `if ( `*expression*` ) `*statement*
| `if ( `*expression*` ) `*statement* `else` *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): `a` or `1`, true if non-zero, otherwise false
- An expression `a+b` or `5+3`, true if non-zero, otherwise false
- A comparison `a==5`, true if, comparison is true, otherwise false
- An assignment `a=b`, true if non-zero, otherwise false
- Repeated assignments `a=b=c`, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= `if ( `*expression*` ) `*statement*
| `if ( `*expression*` ) `*statement* `else` *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): `a` or `1`, true if non-zero, otherwise false
- An expression `a+b` or `5+3`, true if non-zero, otherwise false
- A comparison `a==5`, true if, comparison is true, otherwise false
- An assignment `a=b`, true if non-zero, otherwise false
- Repeated assignments `a=b=c`, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= `if ( `*expression*` ) `*statement*
| `if ( `*expression*` ) `*statement* `else` *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): `a` or `1`, true if non-zero, otherwise false
- An expression `a+b` or `5+3`, true if non-zero, otherwise false
- A comparison `a==5`, true if, comparison is true, otherwise false
- An assignment `a=b`, true if non-zero, otherwise false
- Repeated assignments `a=b=c`, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= **`if`** **`(`** *expression* **`)`** *statement*
| **`if`** **`(`** *expression* **`)`** *statement* **`else`** *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): **`a`** or **`1`**, true if non-zero, otherwise false
- An expression **`a+b`** or **`5+3`**, true if non-zero, otherwise false
- A comparison **`a==5`**, true if, comparison is true, otherwise false
- An assignment **`a=b`**, true if non-zero, otherwise false
- Repeated assignments **`a=b=c`**, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= `if ( `*expression*` ) `*statement*
| `if ( `*expression*` ) `*statement* `else` *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): `a` or `1`, true if non-zero, otherwise false
- An expression `a+b` or `5+3`, true if non-zero, otherwise false
- A comparison `a==5`, true if, comparison is true, otherwise false
- An assignment `a=b`, true if non-zero, otherwise false
- Repeated assignments `a=b=c`, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= **`if ( `** *expression* **`)`** *statement*
| **`if ( `** *expression* **`)`** *statement* **`else`** *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): **`a`** or **`1`**, true if non-zero, otherwise false
- An expression **`a+b`** or **`5+3`**, true if non-zero, otherwise false
- A comparison **`a==5`**, true if, comparison is true, otherwise false
- An assignment **`a=b`**, true if non-zero, otherwise false
- Repeated assignments **`a=b=c`**, true if non-zero, otherwise false

Syntax – if

If-statement

statement ::= `if ( ` *expression* `)` *statement*
| `if (` *expression* `)` *statement* `else` *statement*

Expression

expression ::= *[prefix_operators] term [postfix_operators]*
| *term infix_operator expression*

Expressions

- A variable (or constant): `a` or `1`, true if non-zero, otherwise false
- An expression `a+b` or `5+3`, true if non-zero, otherwise false
- A comparison `a==5`, true if, comparison is true, otherwise false
- An assignment `a=b`, true if non-zero, otherwise false
- Repeated assignments `a=b=c`, true if non-zero, otherwise false

Smallest of three numbers

Classroom assignment

- Numbers are: a, b and c
- Let m be $\min(a, b)$ (in a mathematical sense) ,
- then $\min(m, c)$ will be the smallest of the three numbers

Short hand code for $\min(a, b)$?

Smallest of three numbers

Classroom assignment

- Numbers are: a, b and c
- Let m be $\min(a, b)$ (in a mathematical sense) ,
- then $\min(m, c)$ will be the smallest of the three numbers

Short hand code for $\min(a, b)$?

Smallest of three numbers

Classroom assignment

- Numbers are: a, b and c
- Let m be $\min(a, b)$ (in a mathematical sense) ,
- then $\min(m, c)$ will be the smallest of the three numbers

Short hand code for $\min(a, b)$?

Smallest of three numbers

Classroom assignment

- Numbers are: a, b and c
- Let m be $\min(a, b)$ (in a mathematical sense) ,
- then $\min(m, c)$ will be the smallest of the three numbers

Short hand code for $\min(a, b)$?

Smallest of three numbers

Classroom assignment

- Numbers are: a, b and c
- Let m be $\min(a, b)$ (in a mathematical sense) ,
- then $\min(m, c)$ will be the smallest of the three numbers

Short hand code for $\min(a, b)$?

Quadratic revisited

Editor: Note the different branching structure

```
...
 if (d >= 0) { // roots are real
  float r_1, r_2; // the roots
  if (d==0) { // roots are identical
   r_1 = r_2 = -b/(2*a) ;
   printf ("equal roots: ");
  } else { // roots are real
   float d_root = sqrt(d);
   r_1 = (-b + d_root) / (2*a) ;
   r_2 = (-b - d_root) / (2*a) ;
   printf ("real distinct roots: \n");
  }   printf ("%e and %e\n", r_1, r_2);
 } else { // roots are complex
  float d_root = sqrt(-d);
  float r = -b / (2*a) ;
  float c = d_root / (2*a) ;
  printf ("complex roots:\n %e+i%e and\n %e-i%e\n", r, c, r, c);
 }
...
```

Dangling else

- An `else` clause binds to the nearest preceeding `if` clause
- Consider: `if (C1) if (C2) S2 else S3`
- This is equivalent to: `if (C1) {if (C2) S2 else S3}` because `else S3` must bind to `if (C2) S2`, as that is the nearest preceeding `if` clause
- Using this rule, `if (C1) if (C2) S2 else S3 else S4` works out as: `if (C1) {if (C2) S2 else S3} else S4`, which is what we would intuitively expect

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- **`(a+b) * (c+d)`**
- Either **`(a+b)`** or **`(c+d)`** may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Condition evaluation

- Expressions are often evaluated from left to right
- `(a+b) * (c+d)`
- Either `(a+b)` or `(c+d)` may be evaluated first
- Does not conflict with associativity
- That is not a requirement by the language standard
- In some cases the evaluation order matters
- `if (a!=0 && b/a>1)`
- `if (a && c/b>1)`
- `if (a==0 || b/a>1)`

Comma operator

- A comma separated list of expresions, evaluated from left to right
- *expression-1* , *expression-2* , *expression-3*
- *expression-1*, then *expression-2* and finally *expression-3* gets evaluated
- Value of a comma separated list of expresions is the value of the last (rightmost) expression

Branching on multiple case values

Editor:

```
printf ("enter choice (1..3): "); scanf("%d", &choice);
if (choice==1) {
  // do something for choice==1
} else if (choice==2) {
  // do something for choice==2
} else if (choice==3) {
  // do something for choice==3
} else {
  // do something default
}
```

`switch` statement

Editor:

```
printf ("enter choice (1..3): "); scanf("%d", &choice);
switch (choice) {
  case 1: // do something for choice==1
     break ; // will go to next case if break is missing
  case 2: // do something for choice==2
     break ; // will go to next case if break is missing
  case 3: // do something for choice==3
     break ; // will go to next case if break is missing
  default: // do something default
     break ; // recommended to put this break also
}
```

Syntax of `switch` statement

statement ::= **`switch`** **`(`** *expression* **`)`** **`{`**
{ **`case`** *integer_constant_expression* **`:`** *statement* [**`break`** **`;`**] }
[**`default`** **`:`** *statement*]
`}`

Class room assignment

- Initialize `a` (used as an *accumulator*) to zero
- Initialize `r` (used as a working area – a *register*) to zero
- Read `choice`
  - If `choice==1` Read a new number into the accumulator
  - If `choice==2` Add the register value to the accumulator
  - If `choice==3` Subtract the register value to the accumulator
  - If `choice==4` Multiply the accumulator with the value of the register
  - If `choice==5` Divide the accumulator with the value of the register
- Print the value in the accumulator and the register

Recursive definitions

Recursive definitions (RD) are a powerful mechanism to describe objects or a procedure elegantly.

An RD has three types of clauses:

- Basis clauses (or simply basis) indicates the starting items/steps
- Inductive clauses establishes the ways by which elements/steps identified so far can be combined to produce new elements/steps
- An extremal clause (may be implicit) rules out any item/step not derived via the recursive defintion (either as a basis case or via induction)

RDs can often be stated only using conditionals

Examples of recursive definitions

Example (Day-to-day use)

John's ancestors

Basis John's parents are ancestors of John

Induction Any parent of an ancestor of John is an ancestor of John

Extremality No one else is an ancestor of John

Identification of royalty

Basis A monarch is a royal

Induction A descendent of a royal is a royal

Extremality No one else is a royal

Examples of recursive definitions (contd)

Example (Mathematical examples)

Factorial
- **Basis** factorial(0) = 1
- **Induction** factorial(N) = N $\times$ factorial($N - 1$), if ($N > 0$)

Fibonacci
- **Basis** fib(0) = 0
- **Basis** fib(1) = 1
- **Induction** fib(N) = fib($N - 1$) + fib($N - 1$), if ($N > 1$)

Modular exponention (slow) $a^n \mod m$
- **Basis** $a^1 \mod m = a \mod m$
- **Induction** $a^{p+1} \mod m = (q * a \mod m)$, where $q = a^p \mod m$

Greatest common divisor gcd(a, b), $0 < a < b$

Let $r = b \mod a$
- **Basis** gcd(a, b) = a, if $r = 0$
- **Induction** gcd(a, b) = gcd(r, a), if $r \neq 0$

Examples of recursive definitions (contd)

Example (Mathematical examples)

Factorial
- **Basis** factorial(0) = 1
- **Induction** factorial(N) = N $\times$ factorial($N - 1$), if ($N > 0$)

Fibonacci
- **Basis** fib(0) = 0
- **Basis** fib(1) = 1
- **Induction** fib(N) = fib($N - 1$) + fib($N - 1$), if ($N > 1$)

Modular exponention (slow) $a^n \mod m$
- **Basis** $a^1 \mod m = a \mod m$
- **Induction** $a^{p+1} \mod m = (q * a \mod m)$, where $q = a^p \mod m$

Greatest common divisor gcd(a, b), $0 < a < b$

Let $r = b \mod a$
- **Basis** gcd(a, b) = a, if $r = 0$
- **Induction** gcd(a, b) = gcd(r, a), if $r \neq 0$

Examples of recursive definitions (contd)

Example (Mathematical examples)

Factorial
- **Basis** factorial(0) = 1
- **Induction** factorial(N) = N $\times$ factorial($N - 1$), if ($N > 0$)

Fibonacci
- **Basis** fib(0) = 0
- **Basis** fib(1) = 1
- **Induction** fib(N) = fib($N - 1$) + fib($N - 1$), if ($N > 1$)

Modular exponention (slow) $a^n \mod m$
- **Basis** $a^1 \mod m = a \mod m$
- **Induction** $a^{p+1} \mod m = (q * a \mod m)$, where $q = a^p \mod m$

Greatest common divisor gcd(a, b), $0 < a < b$

Let $r = b \mod a$
- **Basis** gcd(a, b) = a, if $r = 0$
- **Induction** gcd(a, b) = gcd(r, a), if $r \neq 0$

Examples of recursive definitions (contd)

Example (Mathematical examples)

Factorial
- **Basis** factorial(0) = 1
- **Induction** factorial(N) = N $\times$ factorial($N-1$), if ($N>0$)

Fibonacci
- **Basis** fib(0) = 0
- **Basis** fib(1) = 1
- **Induction** fib(N) = fib($N-1$) + fib($N-1$), if ($N>1$)

Modular exponention (slow) $a^n \mod m$
- **Basis** $a^1 \mod m = a \mod m$
- **Induction** $a^{p+1} \mod m = (q * a \mod m)$, where $q = a^p \mod m$

Greatest common divisor gcd(a, b), $0 < a < b$

Let $r = b \mod a$
- **Basis** gcd(a, b) = a, if $r = 0$
- **Induction** gcd(a, b) = gcd(r, a), if $r \neq 0$

Divide and conquer done recursively

This is a very important problem solving scheme stated as follows:

You are given a problem P

1. Divide P into several smaller subproblems, $P_1, P_2, \ldots, P_n$
   In many cases the number of such problems is small, say two
2. Somehow (may be recursively – in the same way) solve (or conquer), each of the subproblems to get solutions $S_1, S_2, \ldots, S_n$
3. Use $S_1, S_2, \ldots, S_n$ to construct a solution to the original problem, P (to complete the conquer phase)

Examples of divide and conquer

Example (Fast modular exponention to compute $a^n \mod m$)

Basis $a^1 \mod m = a \mod m$

Induction $a^{2p} \mod m = (q * q \mod m)$, where $q = a^p \mod m$

Divide Original problem ($a^{2p} \mod m$) divided into two identical sub-problems ($q = a^p \mod m$)

Conquer
1. Recursively solving ($q = a^p \mod m$)
2. Using the result to compute $a^{2p} \mod m = (q * q \mod m)$

Induction $a^{2p+1} \mod m = ((q * q \mod m) * a \mod m)$, where $q = a^p \mod m$

Divide and conquer Similar to above case, with the additional multiplication by a, resulting from $n = 2p + 1$

Examples of divide and conquer

Example (Fast modular exponention to compute $a^n \mod m$)

Basis $a^1 \mod m = a \mod m$

Induction $a^{2p} \mod m = (q * q \mod m)$, where $q = a^p \mod m$

Divide Original problem ($a^{2p} \mod m$) divided into two identical sub-problems ($q = a^p \mod m$)

Conquer
1. Recursively solving ($q = a^p \mod m$)
2. Using the result to compute $a^{2p} \mod m = (q * q \mod m)$

Induction $a^{2p+1} \mod m = ((q * q \mod m) * a \mod m)$, where $q = a^p \mod m$

Divide and conquer Similar to above case, with the additional multiplication by a, resulting from $n = 2p + 1$

Examples of divide and conquer

Example (Fast modular exponention to compute $a^n \mod m$)

Basis $a^1 \mod m = a \mod m$

Induction $a^{2p} \mod m = (q * q \mod m)$, where $q = a^p \mod m$

Divide Original problem ($a^{2p} \mod m$) divided into two identical sub-problems ($q = a^p \mod m$)

Conquer

1. Recursively solving ($q = a^p \mod m$)
2. Using the result to compute $a^{2p} \mod m = (q * q \mod m)$

Induction $a^{2p+1} \mod m = ((q * q \mod m) * a \mod m)$, where $q = a^p \mod m$

Divide and conquer Similar to above case, with the additional multiplication by a, resulting from $n = 2p + 1$

Examples of divide and conquer (contd)

Example (Choose r items from n items: ${}^nC_r$)

Basis When $r = 0$: ${}^nC_0 = 1$

Basis When $r = n$: ${}^nC_n = {}^nC_{n-n} = {}^nC_0 = 1$

Induction When $r > 0$:

- let a particular item be chosen
  $n - 1$ items left, $r - 1$ items to be chosen, i.e. ${}^{n-1}C_{r-1}$
  - this is an inductive step
- let a particular item not be chosen
  $n - 1$ items left, r items to be chosen, i.e. ${}^{n-1}C_r$
  - this is another inductive step
  - total ways: ${}^{n-1}C_{r-1} + {}^{n-1}C_r$

Divide The sub-problems: ${}^{n-1}C_{r-1}$ and ${}^{n-1}C_r$

Conquer
1. Solving these two sub-problems recursively
2. Adding the results to get the value of ${}^nC_r$

Examples of divide and conquer (contd)

Example (Choose r items from n items: ${}^nC_r$)

Basis When $r = 0$: ${}^nC_0 = 1$

Basis When $r = n$: ${}^nC_n = {}^nC_{n-n} = {}^nC_0 = 1$

Induction When $r > 0$:

- let a particular item be chosen
  $n - 1$ items left, $r - 1$ items to be chosen, i.e. ${}^{n-1}C_{r-1}$
- this is an inductive step
- let a particular item not be chosen
  $n - 1$ items left, r items to be chosen, i.e. ${}^{n-1}C_r$
- this is another inductive step
- total ways: ${}^{n-1}C_{r-1} + {}^{n-1}C_r$

Divide The sub-problems: ${}^{n-1}C_{r-1}$ and ${}^{n-1}C_r$

Conquer
1. Solving these two sub-problems recursively
2. Adding the results to get the value of ${}^nC_r$

Examples of divide and conquer (contd)

Example (Choose r items from n items: ${}^nC_r$)

Basis When $r = 0$: ${}^nC_0 = 1$

Basis When $r = n$: ${}^nC_n = {}^nC_{n-n} = {}^nC_0 = 1$

Induction When $r > 0$:

- let a particular item be chosen
  $n-1$ items left, $r-1$ items to be chosen, i.e. ${}^{n-1}C_{r-1}$
- this is an inductive step
- let a particular item not be chosen
  $n-1$ items left, r items to be chosen, i.e. ${}^{n-1}C_r$
- this is another inductive step
- total ways: ${}^{n-1}C_{r-1} + {}^{n-1}C_r$

Divide The sub-problems: ${}^{n-1}C_{r-1}$ and ${}^{n-1}C_r$

Conquer
1. Solving these two sub-problems recursively
2. Adding the results to get the value of ${}^nC_r$

Examples of divide and conquer (contd)

Example (Choose r items from n items: ${}^{n}C_{r}$)

Basis When $r = 0$: ${}^{n}C_{0} = 1$

Basis When $r = n$: ${}^{n}C_{n} = {}^{n}C_{n-n} = {}^{n}C_{0} = 1$

Induction When $r > 0$:

- let a particular item be chosen
  $n - 1$ items left, $r - 1$ items to be chosen, i.e. ${}^{n-1}C_{r-1}$
- this is an inductive step
- let a particular item not be chosen
  $n - 1$ items left, r items to be chosen, i.e. ${}^{n-1}C_{r}$
- this is another inductive step
- total ways: ${}^{n-1}C_{r-1} + {}^{n-1}C_{r}$

Divide The sub-problems: ${}^{n-1}C_{r-1}$ and ${}^{n-1}C_{r}$

Conquer

1. Solving these two sub-problems recursively
2. Adding the results to get the value of ${}^{n}C_{r}$

Section outline

Average of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let s be the sum of the n numbers:

$$s = \sum_{i=0}^{i=n-1} x_i$$

- Computation of s:
  1. Initialise **s=0**
  2. Looping n times, add x_i to **s** each time
- Average is $\frac{s}{n}$
- Key programming feature needed: a way to do some computations in a loop n times
- More generally, do some computations in a loop while or until some condition is satisfied
- 'C' provides several looping constructs

Average of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let s be the sum of the n numbers:

$$s = \sum_{i=0}^{i=n-1} x_i$$

- Computation of s:
  1. Initialise **s=0**
  2. Looping n times, add x_i to **s** each time
- Average is $\frac{s}{n}$
- Key programming feature needed: a way to do some computations in a loop n times
- More generally, do some computations in a loop while or until some condition is satisfied
- 'C' provides several looping constructs

Average of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let s be the sum of the n numbers:

$$s = \sum_{i=0}^{i=n-1} x_i$$

- Computation of s:
  1. Initialise **s=0**
  2. Looping n times, add x_i to **s** each time
- Average is $\frac{s}{n}$
- Key programming feature needed: a way to do some computations in a loop n times
- More generally, do some computations in a loop while or until some condition is satisfied
- 'C' provides several looping constructs

Average of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let s be the sum of the n numbers:

$$s = \sum_{i=0}^{i=n-1} x_i$$

- Computation of s:
  1. Initialise **s=0**
  2. Looping n times, add x_i to **s** each time
- Average is $\frac{s}{n}$
- Key programming feature needed: a way to do some computations in a loop n times
- More generally, do some computations in a loop while or until some condition is satisfied
- 'C' provides several looping constructs

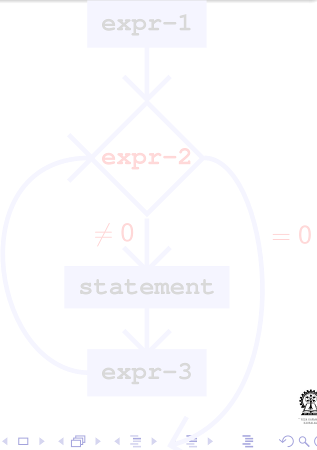
Syntax/grammar – for

for

statement ::= **`for`** `(` *expression-1* `;` *expression-2* `;` *expression-3* `)`
statement

Meaning

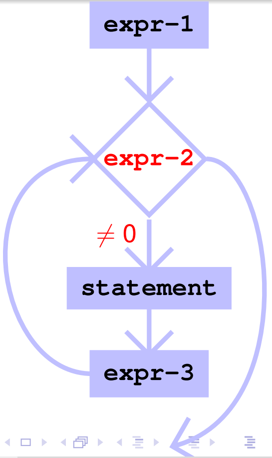
```
expression-1 ;
FTEST: if ( expression-2 ) {
  statement
  expression-3 ;
  goto FTEST ;
}
```



Syntax/grammar – for

for

statement ::= **for** (*expression-1* ; *expression-2* ; *expression-3*)
statement



Meaning

```
expression-1 ;
FTEST: if ( expression-2 ) {
  statement
  expression-3 ;
  goto FTEST ;
}
```

Examples – 'for'

Editor:

```
#include <stdio.h>
main() {
 float s=0, x, avg;
 int i, n;
 printf ("enter n: ");
 scanf ("%d", &n);
 for (i=0; i<n; i++) {
  // note: i starts at 0 and leaves after reaching n
  printf ("enter x: ");
  scanf("%f", &x);
  s = s + x;
 }
 avg=s/n;
 printf("average of the given %d numbers is %f\n",
  n, avg);
}
```

Results

Compile and run:

```
$ cc average.c -o average
$ ./average
enter n: 5
enter x: 2
enter x: 3
enter x: 4
enter x: 5
enter x: 6
average of the given 5 numbers is 4.000000
```

Standard deviation of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let their average be $\bar{x}$
- The variance

$$\sigma^2 = \frac{1}{n}\left(\sum_i (x_i - \bar{x})^2\right)$$
$$= \frac{1}{n}\sum_i (x_i^2) - \bar{x}^2$$

- The standard deviation is σ
- Need to compute both $\sum_i x_i$ and $\sum_i x_i^2$

Standard deviation of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let their average be $\bar{x}$
- The variance

$$\sigma^2 = \frac{1}{n}\left(\sum_i (x_i - \bar{x})^2\right)$$
$$= \frac{1}{n}\sum_i (x_i^2) - \bar{x}^2$$

- The standard deviation is σ
- Need to compute both $\sum_i x_i$ and $\sum_i x_i^2$

Standard deviation of some numbers

- Let there be n numbers: $x_i, i = 0..(n-1)$
- Let their average be $\bar{x}$
- The variance

$$\sigma^2 = \frac{1}{n}\left(\sum_i (x_i - \bar{x})^2\right)$$
$$= \frac{1}{n}\sum_i (x_i^2) - \bar{x}^2$$

- The standard deviation is σ
- Need to compute both $\sum_i x_i$ and $\sum_i x_i^2$

Program

Editor: Compilation should be with -lm

```
#include <stdio.h>
#include <math.h>
main() {
 float s=0, sq=0, x, avg, var, std;
 int i, n;
 printf ("enter n: "); scanf ("%d", &n);
 for (i=0; i<n; i++) {
  printf ("enter x: ");  scanf("%f", &x);
  s = s + x; sq = sq + x*x;
 }
 avg=s/n;
 var = sq/n - avg*avg ; std = sqrt(var) ;
 printf("avg. & st. dev. of the %d numbers: %f, %f\n",
  n, avg, std);
}
```

Computation of e^x

- $e^x = \sum_{i>=0} T_i$, where $T_i = \dfrac{x^i}{i!}$
- T_i may be recursively defined as:
  - $T_0 = 1$
  - $T_j = \dfrac{x}{j} T_{j-1}$, if $j > 0$

Program

Editor:

```
#include <stdio.h>
main() {
 int n, i;
 float x, T=1.0, S=0.0;
 printf ("enter number of terms to add: ");
 scanf ("%d", &n);
 printf ("enter value of x: ");
 scanf ("%f", &x);
 for (i=1; i<n ; i++) {
  S = S + T; // add current term to sum
  T = T*x/i; // Compute T(i+1)
 }
 printf ("x=%f, e**x=%f\n", x, S);
}
```

Computation of e^x accurate to some value

- $e^x = \sum_{i>=0} \frac{x^i}{i!}$
- $e^x = \sum_{i>=0} T_i$, where
-

$$\begin{aligned} T_i &= 1 \text{ if } (i = 0) \\ &= \frac{x}{i} T_{i-1} \text{ otherwise} \end{aligned}$$

- How long should we keep adding terms?
- Let the acceptable error be r
- We can stop when the contribution of the current term is less than r

Computation of e^x accurate to some value

- $e^x = \sum_{i>=0} \frac{x^i}{i!}$
- $e^x = \sum_{i>=0} T_i$, where
- $$T_i = 1 \text{ if } (i = 0)$$
  $$= \frac{x}{i} T_{i-1} \text{ otherwise}$$
- How long should we keep adding terms?
- Let the acceptable error be r
- We can stop when the contribution of the current term is less than r

Computation of e^x accurate to some value

- $e^x = \sum_{i>=0} \frac{x^i}{i!}$
- $e^x = \sum_{i>=0} T_i$, where
- $$T_i = 1 \text{ if } (i = 0)$$
  $$= \frac{x}{i} T_{i-1} \text{ otherwise}$$
- How long should we keep adding terms?
- Let the acceptable error be r
- We can stop when the contribution of the current term is less than r

Computation of e^x accurate to some value

- $e^x = \sum_{i>=0} \frac{x^i}{i!}$
- $e^x = \sum_{i>=0} T_i$, where
- $$\begin{aligned} T_i &= 1 \text{ if } (i = 0) \\ &= \frac{x}{i} T_{i-1} \text{ otherwise} \end{aligned}$$
- How long should we keep adding terms?
- Let the acceptable error be r
- We can stop when the contribution of the current term is less than r

Program

Editor:

```
#include <stdio.h>
main() {
 int i=0;
 float x, r, T=1.0, S=0.0;
 printf ("enter value of x: ");
 scanf ("%f", &x);
 printf ("enter value of error: ");
 scanf ("%f", &r);
 while (T>r) { // while loop
  S = S + T; // add current term to sum
  i++; // increment i within the loop body
  T = T*x/i; // Compute T(i+1)
 }
 printf ("x=%f, e**x=%f\n", x, S);
}
```

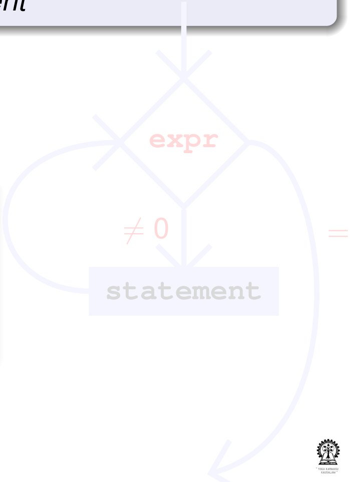
Syntax/grammar – while

while

statement ::= **`while`** **`(`** *expression* **`)`** *statement*

Meaning

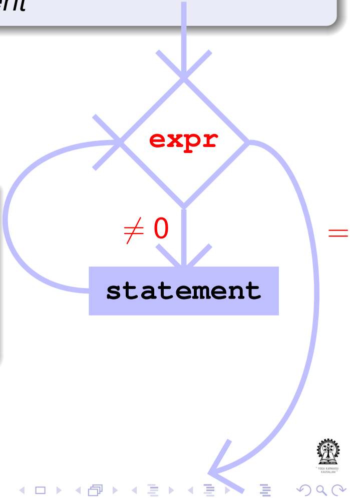
```
WTEST: if ( expression ) {
   statement
   goto WTEST ;
}
```



Syntax/grammar – while

while

statement ::= **`while`** **`(`** *expression* **`)`** *statement*



Meaning

```
WTEST: if ( expression ) {
   statement
   goto WTEST ;
}
```

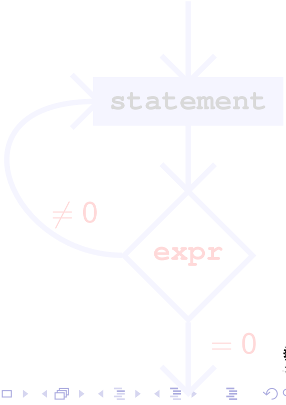
Syntax/grammar – do-while

while

statement ::= **`do`** *statement* **`while`** **(** *expression* **)** **`;`**

Meaning

```
DWTEST: {
   statement
} if ( expression )  goto
DWTEST ;
```



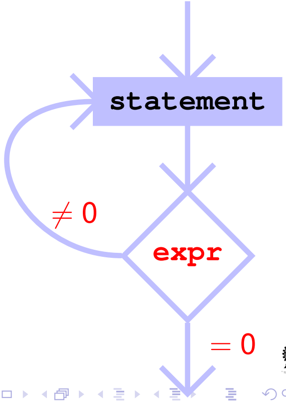
Syntax/grammar – do-while

while

statement ::= **`do`** *statement* **`while`** **`(`** *expression* **`)`** **`;`**

Meaning

```
DWTEST: {
  statement
} if ( expression )  goto
DWTEST ;
```



An alternate program for e^x

Editor:

```
#include <stdio.h>
main() {
 int i=0;
 float x, r, T=1.0, S=0.0;
 printf ("enter value of x: ");
 scanf ("%f", &x);
 printf ("enter value of error: ");
 scanf ("%f", &r);
 do { // do-while loop
  S = S + T; // add current term to sum
  i++; // increment i within the loop body
  T = T*x/i; // Compute T(i+1)
 } while (T>r)
 printf ("x=%f, e**x=%f\n", x, S);
}
```

Section outline

15 **More on loops**
- Breaking out
- Continue

Average, when size is not known in advance

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x, $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Average, when size is not known in advance

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x, $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Average, when size is not known in advance

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x, $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Average, when size is not known in advance

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x, $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Infinite *for*, *while* and *do-while* loops

- `for (expr-1 ; ; expr-3) ;`
- `for (expr-1 ; ; expr-3) { statements }`
- `while (1) { statements }`
- `do { statements } while (1) ;`

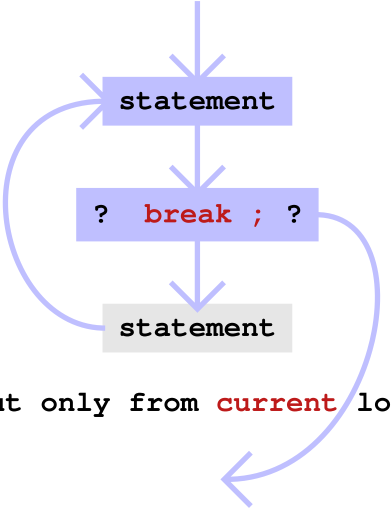
Caution

```
for (expr-1;;expr-3) ;
{ statements }
```

Unwanted infinite loop

```
for (expr-1;;expr-3) ;
{ statements }
```

Diagrammatic view of infinite loop with break



Breaks out only from current loop

Program

Editor:

```
#include <stdio.h>
main() {
 float s=0, x, avg;
 int i, n;
 for (n=0 ; ; s=s+x, n++) {
  printf ("enter x: ");
  scanf("%f", &x);
  // how to detect end of input ?
  if (feof(stdin)) break; // details of feof, stdin,
later
 }
 if (n>0) { // avoid division by 0!
  avg=s/n;
  printf("average of the given %d numbers is %f\n",
   n, avg);
 }
}
```

Program for e^x using break

Editor:

```
#include <stdio.h>
#define ERROR 1.0e-8
main() {
 int n, i;
 float x, T=1.0, S=0.0;
 printf ("enter value of x: ");
 scanf ("%f", &x);
 for (i=1; ; i++) {
  S = S + T; // add current term to sum
  T = T*x/i; // Compute T(i+1)
  if (T < ERROR) break;
 }
 printf ("x=%f, e**x=%f\n", x, S);
}
```

Average, dropping -ve numbers, also unknown input size

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
    - Try to read a number
    - If end of input is detected, then quit the loop
    - After reading each number x,
    - if x is negative, then skip to next iteration
    - $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

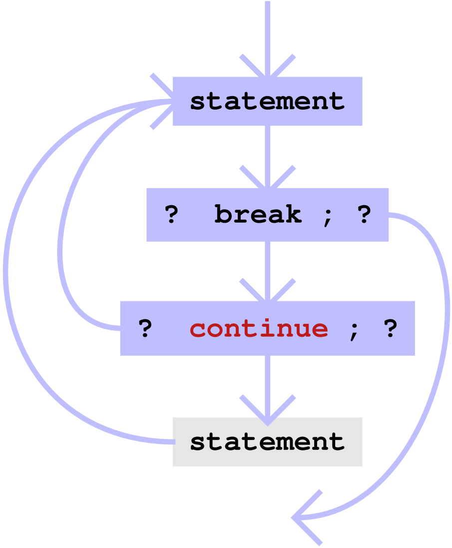
Average, dropping -ve numbers, also unknown input size

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x,
  - if x is negative, then skip to next iteration
  - $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Average, dropping -ve numbers, also unknown input size

- Let s be the sum of the numbers, initially, $s = 0$
- Let n be the numbers seen so far, initially, $n = 0$
- Loop as follows:
  - Try to read a number
  - If end of input is detected, then quit the loop
  - After reading each number x,
  - if x is negative, then skip to next iteration
  - $s = s + x$, $n = n + 1$
- if $n > 0$, then average is $\frac{s}{n}$

Diagrammatic view of (infinite) loop with continue



Program

Editor:

```
#include <stdio.h>
main() {
 float s=0, x, avg; int i, n;
 for (n=0 ; ; ) {
  printf ("enter x: "); scanf("%f", &x);
  // how to detect end of input ?
  if (feof(stdin)) break; // feof, stdin, later
  if (x<0) continue; // skip the rest of the processing
  s=s+x ; n++ ; // skipped if x is negative
 }
 if (n>0) { // avoid division by 0!
  avg=s/n;
  printf("average of the %d numbers: %f\n", n, avg);
 } else printf ("too few numbers!\n");
}
```

Cautionary points on controls

- An expression with non-zero value is treated as **`true`**, otherwise **`false`**
- Thus **`while (1);`** is an infinite loop
- Similarly **`do while (0);`** is an infinite loop
- **`for (;1;);`** is and infinite loop
- Also, a dropped condition in the **`for`** loop is treated as **`true`**, thus **`for (;;);`** is an infinite loop

Syllabus (Theory)

Introduction to the Digital Computer;
Introduction to Programming – Variables, Assignment; Expressions; Input/Output;
Conditionals and Branching; Iteration;
Functions; Recursion; Arrays; Introduction to Pointers; Strings; Structures;
Introduction to Data-Procedure Encapsulation;
Dynamic allocation; Linked structures;
Introduction to Data Structure – Stacks and Queues; Searching and Sorting; Time and space requirements.

Part VI

1D Arrays

Section outline

16 **Arrays**

- Need for arrays
- Sample definitions
- Array initialisation
- Memory snapshots

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in ‘C’ using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in ‘C’ using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in 'C' using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in ‘C’ using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in ‘C’ using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in 'C' using arrays
- The size of the array is usually fixed

Need for arrays

- Vectors and matrices have long been used to represent information – well before the advent of computers
- Dot products, cross products, vector triple products, solution to systems of linear equations, eigen vector computation and many more mathematical operations defined using vectors and matrices
- Support for these in a high-level programming language is only expected
- Two important characteristics: all elements are of the same type and elements are indexed by integers
- Vectors and matrices are representable in ‘C’ using arrays
- The size of the array is usually fixed

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: `int A[5]`
  – first element: `A[0]`, last elementL `A[4]`
- Array of ten reals: `float B[10]`
  – first element: `B[0]`, last elementL `B[9]`
- Array of eleven characters: `char C[11]`
  – first element: `C[0]`, last elementL `C[10]`
- In `int z`, `z` represents the value of the integer – what does the `A` in `int A[5]` represent?
- There is no single value to represent
- The `A` in `int A[5]` represents the starting address of the array – address of the first element of `A`
- For `int A[5]`, `A` $\equiv$ `&(A[0])`
- Same for any array declaration/definition

Sample definitions

- Array of five integers: **`int A[5]`**
  – first element: **`A[0]`**, last elementL **`A[4]`**
- Array of ten reals: **`float B[10]`**
  – first element: **`B[0]`**, last elementL **`B[9]`**
- Array of eleven characters: **`char C[11]`**
  – first element: **`C[0]`**, last elementL **`C[10]`**
- In **`int z`**, **`z`** represents the value of the integer – what does the **`A`** in **`int A[5]`** represent?
- There is no single value to represent
- The **`A`** in **`int A[5]`** represents the starting address of the array – address of the first element of **`A`**
- For **`int A[5]`**, **`A`** $\equiv$ **`&(A[0])`**
- Same for any array declaration/definition

Array initialisation

- `int A[5] = { 1, 2, 4, 8, 16};` – equivalent to `A[0] = 1; A[1] = 2; A[2] = 4; A[3] = 8; A[4] = 16;`
- `int A[5] = { 1, 2};`
- `A[0] = 1; A[1] = 2;`
- "Default-initialisation" (usually zeroes) for the the remaining elements – `A[2] = A[3] = A[4] = 0`, by default
- `char C[5] = "Yes";`

Integer and Character arrays in memory

`A[0]`	00000000	00000000	00000000	00000001
address	3075	3074	3073	**3072**
`A[1]`	00000000	00000000	00000000	00000010
address	3079	3078	3077	**3076**
`A[2]`	00000000	00000000	00000000	00000100
address	3083	3082	3081	**3080**
`A[3]`	00000000	00000000	00000000	00001000
address	3087	3086	3085	**3084**
`A[4]`	00000000	00000000	00000000	00010000
address	3091	3090	3089	**3088**
`C[3]..C[0]`	00000000	01110011	01100101	01011001
address	3095	3094	3093	**3092**
`...C[4]`	10100011	00001101	01110010	10110110
address	3099	3098	3097	**3096**

`A` has address 3072 and its elements are initialised

`C` has address 3088 and its elements are partially initialised

Section outline

17 Working with arrays

- Address arithmetic
- Array declaration
- Passing 1D Arrays

Address arithmetic

- Integer and character array elements have different sizes
- `&A[0]`, `&A[4]`, `&C[3]` gives us addresses (references) of the desired array elements – '`&`' is the reference operator
- `*A`, `*C` yields the value at the addresses of `A` and `C`, resp. – '`*`' is the de-reference operator
- Can we compute on our own? – often needed
- Clever address arithmetic in 'C'
- `A+0` $\equiv$ `&A[0]`, `A[0]` $\equiv$ `*(A+0)`
- `A+4` $\equiv$ `&A[4]`, `A[4]` $\equiv$ `*(A+4)`
- `&A[i]` $\equiv$ `A+i` Implicitly: addr. of A $+$ $i\times$size of an integer – done internally by compiler, never multiply yourself
- `C+3` $\equiv$ `&C[3]`, `C[3]` $\equiv$ `*(C+3)`
- `&C[i]` $\equiv$ `C+i` Implicitly: addr. of C $+$ $i\times$size of an character

Address arithmetic

- Integer and character array elements have different sizes
- `&A[0]`, `&A[4]`, `&C[3]` gives us addresses (references) of the desired array elements – '`&`' is the reference operator
- `*A`, `*C` yields the value at the addresses of `A` and `C`, resp. – '`*`' is the de-reference operator
- Can we compute on our own? – often needed
- Clever address arithmetic in 'C'
- `A+0` $\equiv$ `&A[0]`, `A[0]` $\equiv$ `*(A+0)`
- `A+4` $\equiv$ `&A[4]`, `A[4]` $\equiv$ `*(A+4)`
- `&A[i]` $\equiv$ `A+i` Implicitly: addr. of A $+$ $i\times$size of an integer – done internally by compiler, never multiply yourself
- `C+3` $\equiv$ `&C[3]`, `C[3]` $\equiv$ `*(C+3)`
- `&C[i]` $\equiv$ `C+i` Implicitly: addr. of C $+$ $i\times$size of an character

Address arithmetic

- Integer and character array elements have different sizes
- `&A[0]`, `&A[4]`, `&C[3]` gives us addresses (references) of the desired array elements – '`&`' is the reference operator
- `*A`, `*C` yields the value at the addresses of `A` and `C`, resp. – '`*`' is the de-reference operator
- Can we compute on our own? – often needed
- Clever address arithmetic in 'C'
- `A+0` $\equiv$ `&A[0]`, `A[0]` $\equiv$ `*(A+0)`
- `A+4` $\equiv$ `&A[4]`, `A[4]` $\equiv$ `*(A+4)`
- `&A[i]` $\equiv$ `A+i` Implicitly: addr. of A $+$ $i \times$size of an integer – done internally by compiler, never multiply yourself
- `C+3` $\equiv$ `&C[3]`, `C[3]` $\equiv$ `*(C+3)`
- `&C[i]` $\equiv$ `C+i` Implicitly: addr. of C $+$ $i \times$size of an character

Address arithmetic

- Integer and character array elements have different sizes
- `&A[0]`, `&A[4]`, `&C[3]` gives us addresses (references) of the desired array elements – '`&`' is the reference operator
- `*A`, `*C` yields the value at the addresses of `A` and `C`, resp. – '`*`' is the de-reference operator
- Can we compute on our own? – often needed
- Clever address arithmetic in 'C'
- `A+0` $\equiv$ `&A[0]`, `A[0]` $\equiv$ `*(A+0)`
- `A+4` $\equiv$ `&A[4]`, `A[4]` $\equiv$ `*(A+4)`
- `&A[i]` $\equiv$ `A+i` Implicitly: addr. of A $+$ $i\times$size of an integer – done internally by compiler, never multiply yourself
- `C+3` $\equiv$ `&C[3]`, `C[3]` $\equiv$ `*(C+3)`
- `&C[i]` $\equiv$ `C+i` Implicitly: addr. of C $+$ $i\times$size of an character

Address arithmetic

- Integer and character array elements have different sizes
- `&A[0]`, `&A[4]`, `&C[3]` gives us addresses (references) of the desired array elements – '`&`' is the reference operator
- `*A`, `*C` yields the value at the addresses of `A` and `C`, resp. – '`*`' is the de-reference operator
- Can we compute on our own? – often needed
- Clever address arithmetic in 'C'
- `A+0` $\equiv$ `&A[0]`, `A[0]` $\equiv$ `*(A+0)`
- `A+4` $\equiv$ `&A[4]`, `A[4]` $\equiv$ `*(A+4)`
- `&A[i]` $\equiv$ `A+i` Implicitly: addr. of A $+$ $i \times$size of an integer – done internally by compiler, never multiply yourself
- `C+3` $\equiv$ `&C[3]`, `C[3]` $\equiv$ `*(C+3)`
- `&C[i]` $\equiv$ `C+i` Implicitly: addr. of C $+$ $i \times$size of an character

Reading integers into an array

Editor:

```
#include <stdio.h>
#define SIZE 5
int main() {
 int A[SIZE], B[SIZE], i;
 for (i=0; i<SIZE; i++) {
  printf("Enter A[%d]: ", i);
  scanf("%d", &(A[i])); // using address operator
 }
 for (i=0; i<SIZE; i++) {
  printf("Enter B[%d]: ", i);
  scanf("%d", B+i); // using address arithmetic
  // &B[i] ≡ B+i
 }
return 0; }
```

Populating an array manually is not convenient

Reading integers into an array

Editor:

```
#include <stdio.h>
#define SIZE 5
int main() {
 int A[SIZE], B[SIZE], i;
 for (i=0; i<SIZE; i++) {
  printf("Enter A[%d]: ", i);
  scanf("%d", &(A[i])); // using address operator
 }
 for (i=0; i<SIZE; i++) {
  printf("Enter B[%d]: ", i);
  scanf("%d", B+i); // using address arithmetic
  // &B[i] ≡ B+i
 }
return 0; }
```

Populating an array manually is not convenient

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

Array declaration

- `int A[5]` is a definition of an array, because storage space gets allocated
- `int aD[]` is a declaration that `aD` represents a single dimensional array of integers – `aD` can store a reference (pointer) to an `int` array – no storage space gets allocated for the array elements
- `aD` is essentially an un-initialised address of an integer array
- It should be used only after initialisation (say `aD = A`)
- NB. The size of the declared array `aD` is not specified
- Not needed for a single dimensional array

View in memory

`A[0]`	`00000000`	`00000000`	`00000000`	`00000001`
address	3075	3074	3073	**3072**
`A[1]`	`00000000`	`00000000`	`00000000`	`00000010`
address	3079	3078	3077	**3076**
`A[2]`	`00000000`	`00000000`	`00000000`	`00000100`
address	3083	3082	3081	**3080**
`A[3]`	`00000000`	`00000000`	`00000000`	`00001000`
address	3087	3086	3085	**3084**
`A[4]`	`00000000`	`00000000`	`00000000`	`00010000`
address	3091	3090	3089	**3088**
`aD`	`01101101`	`01110011`	`01110101`	`11011001`
address	3095	3094	3093	**3092**

`int A[5], aD[];` location of **`aD`** initially has garbage

`aD=A;` Now `aD` and `A`, both refer to 3072

There is no location for `A` containing 3072, compiler knows that 3072 should be used for `A`, where appropriate

View in memory

`A[0]`	00000000	00000000	00000000	00000001
address	3075	3074	3073	**3072**
`A[1]`	00000000	00000000	00000000	00000010
address	3079	3078	3077	**3076**
`A[2]`	00000000	00000000	00000000	00000100
address	3083	3082	3081	**3080**
`A[3]`	00000000	00000000	00000000	00001000
address	3087	3086	3085	**3084**
`A[4]`	00000000	00000000	00000000	00010000
address	3091	3090	3089	**3088**
`aD`	00000000	01110011	00001100	00000000
address	3095	3094	3093	**3092**

`int A[5], aD[];` location of **`aD`** initially has garbage

`aD=A;` Now **`aD`** and **`A`**, both refer to 3072

There is no location for **`A`** containing 3072, compiler knows that 3072 should be used for **`A`**, where appropriate

View in memory

A[0]	00000000	00000000	00000000	00000001
address	3075	3074	3073	**3072**
A[1]	00000000	00000000	00000000	00000010
address	3079	3078	3077	**3076**
A[2]	00000000	00000000	00000000	00000100
address	3083	3082	3081	**3080**
A[3]	00000000	00000000	00000000	00001000
address	3087	3086	3085	**3084**
A[4]	00000000	00000000	00000000	00010000
address	3091	3090	3089	**3088**
aD	00000000	01110011	00001100	00000000
address	3095	3094	3093	**3092**

`int A[5], aD[];` location of `aD` initially has garbage

`aD=A;` Now `aD` and `A`, both refer to 3072

There is no location for `A` containing 3072, compiler knows that 3072 should be used for `A`, where appropriate

Initialise an array with integers

Editor:

```
#include <stdlib.h>
#include <time.h>
#define SIZE 50
populateRand(int Z[], int sz) {
// array Z of type int is declared
 int i;
 for (i=0; i<sz ; i++) Z[i]=mrand48();
} // ``man mrand48'' for details
int main() {
 int A[SIZE]; // array A of SIZE ints is defined
 srand48(time(NULL));
 // to get fresh random numbers on each run
 populateRand(A, SIZE); // call to populate A randomly
 return 0; }
```

Z=A (Z gets defined to A) via `populateRand(A, SIZE)`

Passing 1D Arrays to functions

- 1D arrays are passed to functions with or without their dimensions, as `int A[10]` or `int A[]`
- Only the address of the array, as available in the calling function (caller) is passed
- There is no new allocation of memory to store arrays passed as formal parameters
- `A[i]` is obtained as `*(A+1)`, where the dimension does not play any role
- Formal parameters of functions declared as arrays are always arrays declarations

Part VII

More on functions

Section outline

Finding average of two numbers

Editor: Simple program that does not compile

```
#include <stdio.h>
main() {
 float x, y, avg;  printf ("enter two numbers: ");
 scanf ("%f%f", &x, &y);
 avg = avg_fun(x, y);
 printf("average of the given numbers is %f\n", avg);
}

float avg_fun (float a, float b) {
 return (a + b)/2;
}
```

Compile:

```
$ cc avg2.c -o avg2
avg2.c:8: error: conflicting types for ‘avg_fun’
avg2.c:5: error: previous implicit declaration of ‘avg_fun’ was here
make: *** [avg2] Error 1
```

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Explanation of compilation failure

- If a function is used before it is defined, the compiler cannot handle the function call properly (its return type may be defaulted to int)
- Solution:
  - Define the functions before they are used – not always possible (why?)
  - Function may be recursive – to be seen soon
  - Use forward declarations, using function prototypes
- Presence of a prototype enables automatic type casting, if necessary
- Functions taking no arguments should have a prototype with (void) as the argument specification

Use case of prototypes

Editor:

```
#include <stdio.h>

float avg_fun (float , float ) ;

main() {
 float x, y, avg;  printf ("enter two numbers: ");
 scanf ("%f%f", &x, &y);  avg = avg_fun(x, y);
 printf("average of the given numbers is %f\n", avg);
}
float avg_fun (float a, float b) {
 return (a + b)/2;
}
```

Compile:

```
$ cc avg2.c -o avg2
$
```

Function prototype – example (contd.)

Editor:

```
#include <stdio.h>

float avg_fun (float x , float y ) ;

main() {
 float x, y, avg;  printf ("enter two numbers: ");
 scanf ("%f%f", &x, &y);  avg = avg_fun(x, y);
 printf("average of the given numbers is %f\n", avg);
}
float avg_fun (float a, float b) {
 return (a + b)/2;
}
```

Compile:

```
$ cc avg2.c -o avg2
$
```

Points to note

- Prototypes are an advance declaration (but not definition) of the function
- Prototypes indicate the type and number of arguments taken by the functions
- Prototypes also indicate the return type of the function
- Parameter names are not needed in a prototype declaration
- If parameter names are used, then they are ignored
- However, it is sometimes easier to indicate the type of the parameter by declaring it in the regular manner, using a parameter name

Evaluation version of Fibonacci

Editor: Counting using a global variable

```
#include <stdio.h>
int count;
// scope of global variable count covers whole file
int fib_rec_Eval (int n) {
  count++;
  if (n < 2) return 1 ;
  return fib_rec_Eval (n-1) + fib_rec_Eval (n-2) ;
}
main() {
  count=0;
  printf ("fib_rec_Eval(5)=%d\n", fib_rec_Eval(5));
  printf ("count=%d\n", count);
}
```

Evaluation version of Fibonacci

Editor: Counting using a global variable

```
#include <stdio.h>
int count;
// scope of global variable count covers whole file
int fib_rec_Eval (int n) {
  count++;
  if (n < 2) return 1 ;
  return fib_rec_Eval (n-1) + fib_rec_Eval (n-2) ;
}
main() {
  count=0;
  printf ("fib_rec_Eval(5)=%d\n", fib_rec_Eval(5));
  printf ("count=%d\n", count);
}
```

Evaluation version of Fibonacci (contd.)

Editor: Counting using a static variable

```
#include <stdio.h>
int fib_rec_Eval (int n, int flag) {
static int count; // automatically initialize to 0
// count has usual scope -- within this function
  if (flag) { // flag=1 for printing count
    printf ("fib_rec_Eval called %d times\n", count);
    count=0; // reset count for the next round of
counting
  } else { // flag=0 for normal usage
    count++; // value is remembered across calls!
  }
  if (n < 2) return 1 ;
  return fib_rec_Eval (n-1, 0) + fib_rec_Eval (n-2, 0) ;
}
main() {
  printf ("fib_rec_Eval(5, 0)=%d\n", fib_rec_Eval(5, 0));
  fib_rec_Eval(0, 1); // for printing statistics
}
```

Evaluation version of Fibonacci (contd.)

Editor: Counting using a static variable

```
#include <stdio.h>
int fib_rec_Eval (int n, int flag) {
static int count; // automatically initialize to 0
// count has usual scope -- within this function
  if (flag) { // flag=1 for printing count
    printf ("fib_rec_Eval called %d times\n", count);
    count=0; // reset count for the next round of
counting
  } else { // flag=0 for normal usage
    count++; // value is remembered across calls!
  }
  if (n < 2) return 1 ;
  return fib_rec_Eval (n-1, 0) + fib_rec_Eval (n-2, 0) ;
}
main() {
  printf ("fib_rec_Eval(5, 0)=%d\n", fib_rec_Eval(5, 0));
  fib_rec_Eval(0, 1); // for printing statistics
}
```

Evaluation version of Fibonacci (contd.)

Compile and run:

```
$ cc fib_rec_Eval.c -o fib_rec_Eval
$ ./fib_rec_Eval
fib_rec_Eval(5)=8
fib_rec_Eval called 15 times
```

Evaluation version of Fibonacci (contd.)

Editor: Counting using a global variable

```
#include <stdio.h>
int fibT_Eval(int n, int c_1, int c_2, int flag) {
static int count; // automatically initialize to 0
  if (flag) { // flag=1 for printing count
    printf ("fibT_Eval called %d times\n", count);
  } else { // flag=0 for normal usage
    count++; // value is remembered across calls!
  }
  if (n==0 || n==1) return 1;
  else if (n==2) return c_1 + c_2;
  else return fibT_Eval(n-1, c_1 + c_2, c_1, 0) ;
}
main() {
  printf ("fibT_Eval(5, 1, 1, 0)=%d\n", fibT_Eval(5, 0));
  fibT_Eval(0, 1); // for printing statistics
}
```

Evaluation version of Fibonacci (contd.)

Editor: Counting using a global variable

```
#include <stdio.h>
int fibT_Eval(int n, int c_1, int c_2, int flag) {
static int count; // automatically initialize to 0
  if (flag) { // flag=1 for printing count
    printf ("fibT_Eval called %d times\n", count);
  } else { // flag=0 for normal usage
    count++; // value is remembered across calls!
  }
  if (n==0 || n==1) return 1;
  else if (n==2) return c_1 + c_2;
  else return fibT_Eval(n-1, c_1 + c_2, c_1, 0) ;
}
main() {
  printf ("fibT_Eval(5, 1, 1, 0)=%d\n", fibT_Eval(5, 0));
  fibT_Eval(0, 1); // for printing statistics
}
```

Evaluation version of Fibonacci (contd.)

Compile and run:

```
$ cc fibT_Eval.c -o fibT_Eval
$ ./fibT_Eval
fibT_Eval(5, 1, 1, 0)=8
fibT_Eval called 4 times
```

Observation

The **`fibT()`** implementation of Fibonacci is better than **`fib_rec()`**.

Counting calls to Fibonacci

$$\text{fib}(n) = \text{if } (n \notin \{0, 1\}) \text{ then } \text{fib}(n-1) + \text{fib}(n-2) \quad (1)$$
$$= \text{otherwise } 1 \quad (2)$$

How many times is fib called for $n = 8$?

n	0	1	2
calls	1	1	$1+1+1=3$

n	3	4	5
calls	$1+1+3=5$	$1+3+5=9$	$1+5+9=15$

n	6	7	8
calls	$1+9+15=25$	$1+15+25=41$	$1+25+41=67$

Classroom assignment

- The function **`fib_rec()`** may be called several times.
- Using **`static`** variables within functions develop a way to limit the number of recursive calls made to **`fib_rec()`**.

Classroom assignment

Write a function to check if a positive integer (provided as parameter) is prime.

Classroom assignment

What does it do?

```
unsigned int foo1 ( unsigned int n ) {
  unsigned int t = 0;

    while (n > 0) {
      if (n % 2 == 1) ++t;
        n = n / 2;
    }
    return t;
}
```

Try out the function on a few numbers and also examine the code carefully

Classroom assignment

- The Towers of Hanoi (ToH) problem is as follows.
- You are given three pins (f, t and u).
- Initially, the ‘f’ pin has *n* disks stacked on it such that no disk has a disk of larger radius stacked on it.
- You are required to transfer the *n* disks from the ‘f’ pin to the ‘t’ pin using the ‘u’ pin, so that, it is never the case that a disk has a disk of larger radius stacked on it.
- You need to write a function that can generate (print) the sequence of individual disk transfers so that the overall transfer is achieved.

Classroom assignment

Catalan numbers are defined as follows:

$$C_0 = 1$$
$$C_1 = 1$$
$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \ldots + C_{n-2} C_1 + C_{n-1} C_0 \text{ for } n >= 2$$

Write a function to compute C_n

Scope rules

- Declarations in a parameter list of a function extend over the entire function, overridding is not permitted
- Scope declaration of a variable in a block extends to contained sub-blocks
- Declaration of a variable in a block overrides any earlier declaration of that variable (unless it is a function parameter)

Section outline

19 **References**

- Need to pass addresses
- Storage snapshots
- Swapping two variable
- Summary

Possible to increment x using a function?

Editor: Does it increment ?

```
#include <stdio.h>
int increment (int x) {
  x += 1; // increment x by 1
  return x;
}
main() {
  int x=5;
  printf("increment(%d)=%d\n", x, increment(x));
  printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
increment(5)=6
x=5
```

Possible to increment x using a function?

Editor: Does it increment ?

```
#include <stdio.h>
int increment (int x) {
  x += 1; // increment x by 1
  return x;
}
main() {
  int x=5;
  printf("increment(%d)=%d\n", x, increment(x));
  printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
increment(5)=6
x=5
```

Possible to increment x using a function?

Editor: Does it increment ?

```c
#include <stdio.h>
int increment (int x) {
  x += 1; // increment x by 1
  return x;
}
main() {
  int x=5;
  printf("increment(%d)=%d\n", x, increment(x));
  printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
increment(5)=6
x=5
```

Incrementing x using a function

Editor: Sending and using address of x (as with scanf)

```
#include <stdio.h>
void increment (int *xA)
{// xA is a pointer to an integer
  *xA += 1; // increment contents of location xA by 1
// return x; // Not needed!
}
main() {
  int x=5;
  increment(&x); // passing address of (reference to) x
  printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
x=6
```

Incrementing x using a function

Editor: Sending and using address of x (as with scanf)

```
#include <stdio.h>
void increment (int *xA)
{// xA is a pointer to an integer
  *xA += 1; // increment contents of location xA by 1
// return x; // Not needed!
}
main() {
  int x=5;
  increment(&x); // passing address of (reference to) x
  printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
x=6
```

Incrementing `x` using a function

Editor: Sending and using address of `x` (as with `scanf`)

```
#include <stdio.h>
void increment (int *xA)
{// xA is a pointer to an integer
   *xA += 1; // increment contents of location xA by 1
// return x; // Not needed!
}
main() {
   int x=5;
   increment(&x); // passing address of (reference to) x
   printf("x=%d\n", x);
}
```

Compile and run:

```
$ cc increment.c -o increment
$ ./increment
x=6
```

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`x(=5)`	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`xA(=3072)`	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- `xA` has the address of `x` [as a result of binding of actual value `&x`(=3072) to formal parameter `xA`]
- `xA` is a reference to `x`
- `xA` is dereferenced by the `*` operator to get the value of `x`
- `*` *`reference_to_variable`* ≡ `variable`
- `*` `xA` ≡ `x`

What is there in the variables?

...	....	....	....	....
address	....	....	....	**....**
`x(=5)`	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
...	....	....	....	....
address	....	....	....	**....**
`xA(=3072)`	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
...	....	....	....	....
address	....	....	....	**....**

- `xA` has the address of `x` [as a result of binding of actual value `&x`(=3072) to formal parameter `xA`]
- `xA` is a reference to `x`
- `xA` is dereferenced by the `*` operator to get the value of `x`
- `*` *`reference_to_variable`* ≡ `variable`
- `*` `xA` ≡ `x`

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`x(=5)`	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`xA(=3072)`	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- **`xA`** has the address of **`x`** [as a result of binding of actual value **`&x`**(=3072) to formal parameter **`xA`**]
- **`xA`** is a reference to **`x`**
- **`xA`** is dereferenced by the `*` operator to get the value of **`x`**
- `*` *`reference_to_variable`* ≡ `variable`
- `*` `xA` ≡ `x`

What is there in the variables?

...	....	....	....	....
address	....	....	....	**....**
`x(=5)`	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
...	....	....	....	....
address	....	....	....	**....**
`xA(=3072)`	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
...	....	....	....	....
address	....	....	....	**....**

- **`xA`** has the address of **`x`** [as a result of binding of actual value **`&x`**(=3072) to formal parameter **`xA`**]
- `xA` is a reference to `x`
- `xA` is dereferenced by the `*` operator to get the value of `x`
- `*` *`reference_to_variable`* ≡ `variable`
- `*` `xA` ≡ `x`

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`x(=5)`	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
`xA(=3072)`	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- **`xA`** has the address of **`x`** [as a result of binding of actual value **`&x`**(=3072) to formal parameter **`xA`**]
- **`xA`** is a reference to **`x`**
- **`xA`** is dereferenced by the `*` operator to get the value of **`x`**
- `*` *`reference_to_variable`* $\equiv$ `variable`
- `*` `xA` $\equiv$ `x`

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
x(=5)	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
xA(=3072)	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- **xA** has the address of **x** [as a result of binding of actual value **&x**(=3072) to formal parameter **xA**]
- **xA** is a reference to **x**
- **xA** is dereferenced by the * operator to get the value of **x**
- $*$ *reference_to_variable* $\equiv$ *variable*
- $*$ **xA** $\equiv$ **x**

What is there in the variables?

...	....	....	....	....
address	....	....	....	**....**
x (=5)	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
...	....	....	....	....
address	....	....	....	**....**
xA (=3072)	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
...	....	....	....	....
address	....	....	....	**....**

- **xA** has the address of **x** [as a result of binding of actual value **&x**(=3072) to formal parameter **xA**]
- **xA** is a reference to **x**
- **xA** is dereferenced by the * operator to get the value of **x**
- * ***reference_to_variable*** ≡ **variable**
- * **xA** ≡ **x**

What is there in the variables?

. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
x(=5)	00000000	00000000	00000000	00000101
address	3075	3074	3073	**3072**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**
xA(=3072)	00000000	00000000	00001111	00100000
address	3875	3874	3873	**3872**
. . .	. . . .	. . . .	. . . .	. . . .
address	....	....	....	**....**

- **xA** has the address of **x** [as a result of binding of actual value **&x**(=3072) to formal parameter **xA**]
- **xA** is a reference to **x**
- **xA** is dereferenced by the * operator to get the value of **x**
- * ***reference_to_variable*** ≡ **variable**
- * **xA** ≡ **x**

Swap `x` and `y` (very common problem)

Editor: By passing addresses (references) to `x` and `y`

```
#include <stdio.h>
void swap (int *xA, int *yA) { // note the references
  int temp; // temporary storage
  temp = *xA; // save x in temp
  *xA = *yA; // now copy y to x
  *yA = temp; // saved value of x is finally copied to y
}
main() {
  int x=5, y=9;
  swap (&x, &y);
  printf("x=%d, y=%d\n", x, y);
}
```

Compile and run:

```
$ cc swap.c -o swap
$ ./swap
x=9, y=5
```

Swap x and y (very common problem)

Editor: By passing addresses (references) to x and y

```
#include <stdio.h>
void swap (int *xA, int *yA) { // note the references
  int temp; // temporary storage
  temp = *xA; // save x in temp
  *xA = *yA; // now copy y to x
  *yA = temp; // saved value of x is finally copied to y
}
main() {
  int x=5, y=9;
  swap (&x, &y);
  printf("x=%d, y=%d\n", x, y);
}
```

Compile and run:

```
$ cc swap.c -o swap
$ ./swap
x=9, y=5
```

Swap x and y (very common problem)

Editor: By passing addresses (references) to x and y

```
#include <stdio.h>
void swap (int *xA, int *yA) { // note the references
  int temp; // temporary storage
  temp = *xA; // save x in temp
  *xA = *yA; // now copy y to x
  *yA = temp; // saved value of x is finally copied to y
}
main() {
  int x=5, y=9;
  swap (&x, &y);
  printf("x=%d, y=%d\n", x, y);
}
```

Compile and run:

```
$ cc swap.c -o swap
$ ./swap
x=9, y=5
```

Summary

In the context of the two examples, discussed so far,

- `increment()` could have returned `x+1`
- `x = increment(x)` could have been done
- Same could not be done for `swap()`
- Both `increment()` and `swap()` using references have a sense of simplicity of usage
- Just the call `increment (&x)` or `swap (&x, &y)` is enough – no need for an additional assignment statement
- Pointers (references) also have their problems – to be discovered soon
- Java has done away with pointers

Summary

In the context of the two examples, discussed so far,

- **`increment()`** could have returned **`x+1`**
- **`x = increment(x)`** could have been done
- Same could not be done for **`swap()`**
- Both **`increment()`** and **`swap()`** using references have a sense of simplicity of usage
- Just the call **`increment (&x)`** or **`swap (&x, &y)`** is enough – no need for an additional assignment statement
- Pointers (references) also have their problems – to be discovered soon
- Java has done away with pointers

Summary

In the context of the two examples, discussed so far,

- `increment()` could have returned `x+1`
- `x = increment(x)` could have been done
- Same could not be done for `swap()`
- Both `increment()` and `swap()` using references have a sense of simplicity of usage
- Just the call `increment (&x)` or `swap (&x, &y)` is enough – no need for an additional assignment statement
- Pointers (references) also have their problems – to be discovered soon
- Java has done away with pointers

Section outline

20 **Recursive functions**
- Considerations
- Activation records

Considerations

- A function is said to be recursive when it is permissible to invoke it before its earlier invocation has been completed
- Modern programming languages support recursion
- Earlier versions of FORTRAN did not support recursion
- Recursively defined routines often cannot be implemented in an iterative manner
- In such cases use of recursive functions becomes essential for the problem under consideration
- An important question is what happens to the contents of the variables when the function is called again
- Instead of allocating a fixed space for the variables of a function, fresh space (activation record) is allocated for each invocaton, so that variables do not get overwritten

Recursive and iterative factorial functions

Example

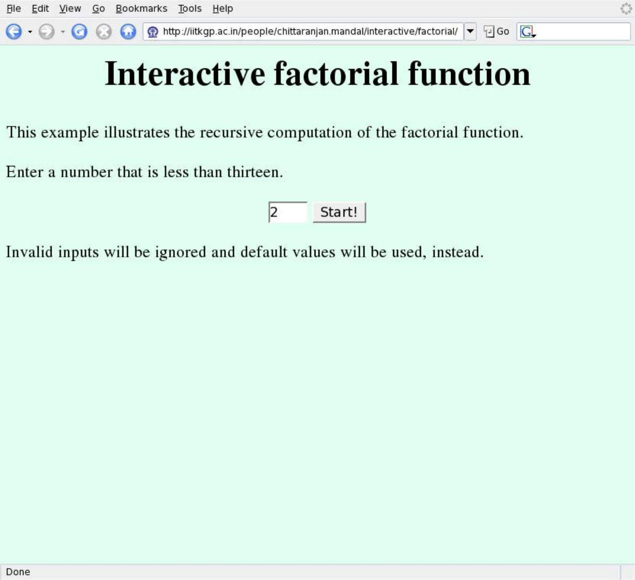
Editor:

```
int fact_iter (int n) {
int i, f;
  for (f=1,i=n;i>0;i--)
    f = f * i ;
  return f;
}
```

Editor:

```
factorial (int n) {
  int f_n_less_1;
  if (n==0) {
    return 1;
  } else {
    f_n_less_1 =
       factorial (n-1);
    return n * f_n_less_1;
  }
}
```

Trace of Recursive Factorial



Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

n=2

Step Run Reset

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

n=2, about to call factorial with 2

Step | Run | Reset

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

factorial called with n=2

Step Run Reset

Call Stack

factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

Call Stack

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

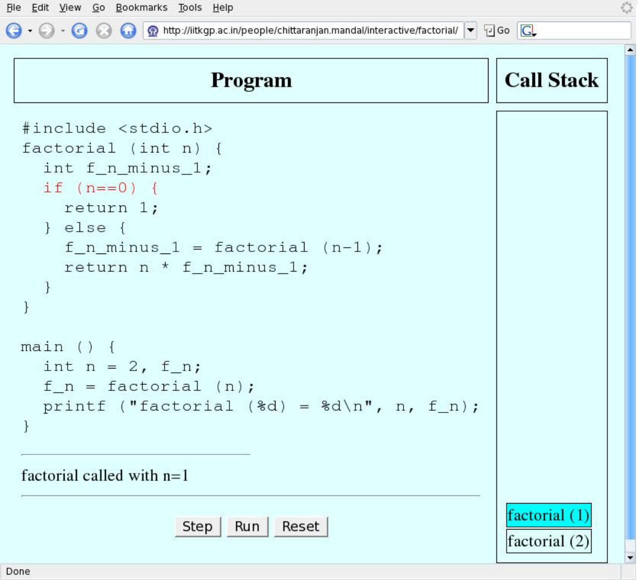
calling factorial recursively with 1

Step Run Reset

factorial (2)

Done

Trace of Recursive Factorial



- **`factorial(1)`** invoked from within invocation of **`factorial(1)`**
- Note the creation of activation records for each invocation of **`factorial()`**
- Fresh set of variables per call through activation record

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

calling factorial recursively with 0

Step Run Reset

Call Stack

factorial (1)

factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

factorial called with n=0

Step | Run | Reset

Call Stack

factorial (0)
factorial (1)
factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

base case, returning 1

Step Run Reset

Call Stack

factorial (0)
factorial (1)
factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

n=1, f_n_minus_1=1, about to return 1

Step Run Reset

Call Stack

factorial (1)

factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

Program

Call Stack

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

n=2, f_n_minus_1=1, about to return 2

Step Run Reset

factorial (2)

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

f_n=2

Step Run Reset

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

---

factorial (2) = 2

Step Run Reset

Done

Trace of Recursive Factorial

File Edit View Go Bookmarks Tools Help

http://iitkgp.ac.in/people/chittaranjan.mandal/interactive/factorial/ Go

```
#include <stdio.h>
factorial (int n) {
  int f_n_minus_1;
  if (n==0) {
    return 1;
  } else {
    f_n_minus_1 = factorial (n-1);
    return n * f_n_minus_1;
  }
}

main () {
  int n = 2, f_n;
  f_n = factorial (n);
  printf ("factorial (%d) = %d\n", n, f_n);
}
```

---

factorial (2) = 2

Over!

Done

Section outline

21 **Recursion with arrays**

- Simple search
- Combinations
- Permuations of n items

Searching (slowly) for a key in an array

- Say we have an array **`A`** of integers and another number – a key
- We want to check whether the key is present in the array or not
  - If there are no elements in the array, then fail
  - Compare the key to the first element in the array,
  - If matched, then done, otherwise search in the rest of the array
- Worst case runtime (counted as number of steps) of described procedure is proportional to number of elements in array

Searching (slowly) for a key in an array

- Say we have an array **A** of integers and another number – a key
- We want to check whether the key is present in the array or not
  - If there are no elements in the array, then fail
  - Compare the key to the first element in the array,
  - If matched, then done, otherwise search in the rest of the array
- Worst case runtime (counted as number of steps) of described procedure is proportional to number of elements in array

Searching (slowly) for a key in an array

- Say we have an array `A` of integers and another number – a key
- We want to check whether the key is present in the array or not
  - If there are no elements in the array, then fail
  - Compare the key to the first element in the array,
  - If matched, then done, otherwise search in the rest of the array
- Worst case runtime (counted as number of steps) of described procedure is proportional to number of elements in array

Searching (slowly) for a key in an array

- Say we have an array `A` of integers and another number – a key
- We want to check whether the key is present in the array or not
  - If there are no elements in the array, then fail
  - Compare the key to the first element in the array,
  - If matched, then done, otherwise search in the rest of the array
- Worst case runtime (counted as number of steps) of described procedure is proportional to number of elements in array

Searching (slowly) for a key in an array

- Say we have an array **A** of integers and another number – a key
- We want to check whether the key is present in the array or not
  - If there are no elements in the array, then fail
  - Compare the key to the first element in the array,
  - If matched, then done, otherwise search in the rest of the array
- Worst case runtime (counted as number of steps) of described procedure is proportional to number of elements in array

Recursive definition for sequential search

searchSeq(A, n, k)

Inductive/recursive case

CI1 [$n > 0$ and k does not match first element of A]

AI1 return searchSeq (rest of A (leaving out the first element), n-1, k)

Base case

CB2 [$n > 0$ and k matches first element of A]

AB2 return success

Base case

CB1 [$n = 0$] (array empty)

AB1 return failure

Searching slowly in an array

Editor: Recursive, ranges by address arithmetic

```
int searchSeqRA(int Z[], int ky, int sz, int pos) {
// sample invocation: searchSeqRA(A, ky, SIZE, 0)
 if (sz==0) return -1; // CB1 ⇒ AB1; failed
 if (Z[0]==ky) return pos; // CB2 ⇒ AB2; matched
 return searchSeqRA(Z+1, ky, sz-1, pos+1); // recursion
} // CI1 ⇒ AI1; finally
```

Editor: Recursive, ranges by array index

```
int searchSeqRI(int Z[], int ky, int sz, int pos) {
// sample invocation: searchSeqRI(A, ky, SIZE, 0)
 if (pos>=sz) return -1; // CB1 ⇒ AB1; failed
 if (Z[pos]==ky) return pos; // CB2 ⇒ AB2; matched
 return searchSeqRI(Z, ky, sz, pos+1); // recursion
} // CI1 ⇒ AI1; finally
```

Searching slowly in an array

Editor: Recursive, ranges by address arithmetic

```
int searchSeqRA(int Z[], int ky, int sz, int pos) {
// sample invocation: searchSeqRA(A, ky, SIZE, 0)
 if (sz==0) return -1; // CB1 ⇒ AB1; failed
 if (Z[0]==ky) return pos; // CB2 ⇒ AB2; matched
 return searchSeqRA(Z+1, ky, sz-1, pos+1); // recursion
} // CI1 ⇒ AI1; finally
```

Editor: Recursive, ranges by array index

```
int searchSeqRI(int Z[], int ky, int sz, int pos) {
// sample invocation: searchSeqRI(A, ky, SIZE, 0)
 if (pos>=sz) return -1; // CB1 ⇒ AB1; failed
 if (Z[pos]==ky) return pos; // CB2 ⇒ AB2; matched
 return searchSeqRI(Z, ky, sz, pos+1); // recursion
} // CI1 ⇒ AI1; finally
```

Searching slowly in an array (contd.)

Editor: Iterative, ranges by array index

```
int searchSeqII(int Z[], int ky, int sz) {  int i;
// sample invocation: searchSeqIR(A, SIZE, 5)
 for (i=0; i<sz ; i++) { CB1 is false within for loop
  if (Z[i]==ky) return i; // CB2 ⇒ AB2; matched
 } // CI1 ⇒ AI1; searching reduced to (i+1) to end of Z
 return -1; // CB1 ⇒ AB1; failed
}
```

Editor: Iterative, ranges by address arithmetic

```
int searchSeqIA(int Z[], int ky, int sz) {
// sample invocation: searchSeqIA(A, SIZE, 5)
 for (; n; n--, Z++) { CB1 is false within for loop
  if (*Z==ky) return i; // CB2 ⇒ AB2; matched
 } // CI1 ⇒ AI1; Z++ advances array head to next element
 return -1; // CB1 ⇒ AB1; failed
}
```

Combinations

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

1. the first item is not taken, so r items must be selected from the remaining $n-1$ items
2. the first item is taken, so $r-1$ items must be selected from the remaining $n-1$ items
3. nothing to do when 0 items are to be selected, report what items were chosen earlier
4. if exactly n of n items are to be chosen, then choose all of them, report what items were chosen earlier and these items

Editor: Combinations of r of n items using array indices

```
void nCrShow (int selVec[], int n, int r, int itemIdx) {
// usage: nCrShow (selVec, n, r, 0), n+itemIdx=totItems
int total, i;
 if (r == 0) { // nothing more to choose, print pattern
  for (total = n + itemIdx, i = 0; i < itemIdx; i++)
   printf ("%d ", selVec[i]);
  for (; i < total; i++) printf ("0 "); printf ("\n");
 } else if (r == n) { // take all n items, print pattern
  for (total = n + itemIdx, i = 0; i < itemIdx; i++)
   printf ("%d ", selVec[i]);
  for (; i < total; i++) printf ("1 "); printf ("\n");
 } else { // induction: either take or drop item itemIdx
  selVec[itemIdx] = 1; gen patterns when item is taken
  nCrShow (selVec, n - 1, r - 1, itemIdx + 1);
  selVec[itemIdx] = 0; gen patterns when item is dropped
  nCrShow (selVec, n - 1, r, itemIdx + 1);
 } // decisions from item itemIdx+1 onwards taken
} // printing of patterns is a required functionality!
```

Permuations of n items

$$P(n) = n \times P(n-1)$$
$$P(0) = 1$$

1. choose the first item in n ways and then take the permuation of the remaining $n-1$ items
2. nothing to do for 0 items

Permuations of n items

Editor: Swap elements in array

```
void swapArr (int arr[], int i, int j) {
// interchange elements at positions i and j of arr[]
  int t;

  t = arr[i];
  arr[i] = arr[j];
  arr[j] = t;
}
```

Permuations of n items (Contd.)

Editor: `nPnShow (pattern, n, 0)`

```
void nPnShow (int pattern[], int n, int nowPos) {
 int i, total;
 if (n <= 1) { // done, now show the pattern
   for (total = n + nowPos, i = 0; i < total; i++)
     printf ("%d ", pattern[i]);
   printf ("\n");
 } else
   for (total = n + nowPos, i = 0; i < n; i++) {
    swapArr (pattern, nowPos, nowPos + i);
    // start with the i-th item
    nPnShow (pattern, n - 1, nowPos + 1);
    // generate permutation of all remaining items
    swapArr (pattern, nowPos, nowPos + i);
    // restore the i-th item at its original position so
    // that the remaining items can be treated consistently
   }
}
```

Section outline

22. **Efficient recursion**
    - Factorial again
    - Tail recursion
    - Handling TR

Factorial – iteratively from recursive definition

$$\text{fact}(n) = \text{if}(n \neq 0) \text{ then } n\,\text{fact}(n-1)$$
$$\text{fact}(0) = 1$$

By repeated substitution,

$$\text{fact}(n) = n\,\text{fact}(n-1) = n(n-1)\,\text{fact}(n-2) = n(n-1)(n-2)\,\text{fact}(n-3)$$

$$\text{fact}(n) = n(n-1)(n-2)\ldots 1\,\text{fact}(0) = n(n-1)(n-2)\ldots 1$$

Thus, $\text{fact}(n)$ may be computed as the product $n(n-1)(n-2)\ldots 1$ – this can be done in a loop

1. Initilise `p = 1`
2. Looping while `n > 0`,
   a. multiply `n` to `p` (`p` = `p` × `n`)
   b. decrement `n` (`n = n-1`)

Factorial – iteratively from recursive definition

$$\mathrm{fact}(n) = \mathrm{if}(n \neq 0) \text{ then } n\, \mathrm{fact}(n-1)$$
$$\mathrm{fact}(0) = 1$$

By repeated substitution,

$\mathrm{fact}(n) = n\,\mathrm{fact}(n-1) = n(n-1)\,\mathrm{fact}(n-2) = n(n-1)(n-2)\,\mathrm{fact}(n-3)$

$$\mathrm{fact}(n) = n(n-1)(n-2)\ldots 1\ \mathrm{fact}(0) = n(n-1)(n-2)\ldots 1$$

Thus, $\mathrm{fact}(n)$ may be computed as the product $n(n-1)(n-2)\ldots 1$ – this can be done in a loop

1. Initilise `p = 1`
2. Looping while `n > 0`,
   a. multiply `n` to `p` (`p` = `p` × `n`)
   b. decrement `n` (`n = n-1`)

Factorial – iteratively from recursive definition

$$\text{fact}(n) = \text{if}(n \neq 0) \text{ then } n \, \text{fact}(n-1)$$
$$\text{fact}(0) = 1$$

By repeated substitution,

$\text{fact}(n) = n \, \text{fact}(n-1) = n(n-1) \, \text{fact}(n-2) = n(n-1)(n-2) \, \text{fact}(n-3)$

$$\text{fact}(n) = n(n-1)(n-2)\ldots 1 \, \text{fact}(0) = n(n-1)(n-2)\ldots 1$$

Thus, $\text{fact}(n)$ may be computed as the product $n(n-1)(n-2)\ldots 1$ – this can be done in a loop

1. Initilise `p = 1`
2. Looping while `n > 0`,
   a. multiply `n` to `p` (`p` = `p` × `n`)
   b. decrement `n` (`n = n-1`)

Factorial – iteratively from recursive definition

$$\text{fact}(n) = \text{if}(n \neq 0) \text{ then } n\,\text{fact}(n-1)$$
$$\text{fact}(0) = 1$$

By repeated substitution,

$$\text{fact}(n) = n\,\text{fact}(n-1) = n(n-1)\,\text{fact}(n-2) = n(n-1)(n-2)\,\text{fact}(n-3)$$

$$\text{fact}(n) = n(n-1)(n-2)\ldots 1\,\text{fact}(0) = n(n-1)(n-2)\ldots 1$$

Thus, $\text{fact}(n)$ may be computed as the product $n(n-1)(n-2)\ldots 1$ – this can be done in a loop

1. Initilise `p = 1`
2. Looping while `n > 0`,
   - a. multiply `n` to `p` (`p` = `p` × `n`)
   - b. decrement `n` (`n = n-1`)

Factorial – iteratively from recursive definition

$$\text{fact}(n) = \text{if}(n \neq 0) \text{ then } n\,\text{fact}(n-1)$$
$$\text{fact}(0) = 1$$

By repeated substitution,

$$\text{fact}(n) = n\,\text{fact}(n-1) = n(n-1)\,\text{fact}(n-2) = n(n-1)(n-2)\,\text{fact}(n-3)$$

$$\text{fact}(n) = n(n-1)(n-2)\ldots 1\ \text{fact}(0) = n(n-1)(n-2)\ldots 1$$

Thus, $\text{fact}(n)$ may be computed as the product $n(n-1)(n-2)\ldots 1$ – this can be done in a loop

1. Initilise `p = 1`
2. Looping while `n > 0`,
   a. multiply **n** to **p** (**p** = **p** × **n**)
   b. decrement **n** (`n = n-1`)

Program, results and discussions

Editor:

```
#include <stdio.h>
main() {
 int i, n, f=1;
 printf ("enter n: ");
 scanf ("%d", &n);
 for (i=n; i>0 ;i--)
  f = f * i ;
 printf ("factorial(%d)=%d\n",
          n, f);
}
```

Compile and run:

```
$ cc factR.c -o factR
$ ./factR
enter n: 5
factorial(5)=120
```

- fact(n) was expanded to the product: $n(n-1)\ldots 1$
- Such simple expansions not always possible
- Simpler options need to be considered
- For $n > 0$, reformulate fact(n) = $n\times$fact($n-1$) as facT(n, p) = facT($n-1, p \times n$)
- Second parameter carries the evolving product
- Let facT($0, p$) = p and
- fact(n) = facT($n, 1$), so that facT() starts with $p = 1$

Program, results and discussions

Editor:

```
#include <stdio.h>
main() {
 int i, n, f=1;
 printf ("enter n: ");
 scanf ("%d", &n);
 for (i=n; i>0 ;i--)
  f = f * i ;
 printf ("factorial(%d)=%d\n",
          n, f);
}
```

Compile and run:

```
$ cc factR.c -o factR
$ ./factR
enter n: 5
factorial(5)=120
```

- fact(n) was expanded to the product: $n(n-1)\ldots 1$
- Such simple expansions not always possible
- Simpler options need to be considered
- For $n > 0$, reformulate fact(n) = $n\times$fact($n-1$) as facT(n, p) = facT($n-1, p \times n$)
- Second parameter carries the evolving product
- Let facT($0, p$) = p and
- fact(n) = facT(n, 1), so that facT() starts with $p = 1$

Program, results and discussions

Editor:

```
#include <stdio.h>
main() {
 int i, n, f=1;
 printf ("enter n: ");
 scanf ("%d", &n);
 for (i=n; i>0 ;i--)
  f = f * i ;
 printf ("factorial(%d)=%d\n",
          n, f);
}
```

Compile and run:

```
$ cc factR.c -o factR
$ ./factR
enter n: 5
factorial(5)=120
```

- fact(n) was expanded to the product: $n(n-1)\ldots 1$
- Such simple expansions not always possible
- Simpler options need to be considered
- For $n > 0$, reformulate fact(n) = $n\times$fact($n-1$) as facT(n, p) = facT($n-1, p \times n$)
- Second parameter carries the evolving product
- Let facT($0, p$) = p and
- fact(n) = facT(n, 1), so that facT() starts with $p = 1$

Program, results and discussions

Editor:

```
#include <stdio.h>
main() {
 int i, n, f=1;
 printf ("enter n: ");
 scanf ("%d", &n);
 for (i=n; i>0 ;i--)
  f = f * i ;
 printf ("factorial(%d)=%d\n",
         n, f);
}
```

Compile and run:

```
$ cc factR.c -o factR
$ ./factR
enter n: 5
factorial(5)=120
```

- fact(n) was expanded to the product: $n(n-1)\ldots 1$
- Such simple expansions not always possible
- Simpler options need to be considered
- For $n > 0$, reformulate fact(n) = $n \times$fact($n-1$) as facT(n, p) = facT($n-1, p \times n$)
- Second parameter carries the evolving product
- Let facT($0, p$) = p and
- fact(n) = facT($n, 1$), so that facT() starts with $p = 1$

Recursive functions for fact() and facT()

Editor:

```
int fact(int n) {
  if (n != 0)
    return n*fact(n-1);
  else return 1;
}
```

Editor:

```
int facT(int n, int p) {
// first call: facT(n, 1);
  if (n != 0)
    return facT(n-1, n*p);
  else return p;
}
```

- Both formulations can be coded recursively, but facT() can be coded as an iterative routine, avoiding the recursive call
- It is a special kind of recursion called tail recursion, where nothing remains to be done after the recursive call
- Many recursive problem formuations lack a tail recursive version
- Tail recursion combines the elegance of recursion and the efficiency of iteration

Recursive functions for fact() and facT()

Editor:

```
int fact(int n) {
  if (n != 0)
    return n*fact(n-1);
  else return 1;
}
```

Editor:

```
int facT(int n, int p) {
// first call: facT(n, 1);
  if (n != 0)
    return facT(n-1, n*p);
  else return p;
}
```

- Both formulations can be coded recursively, but facT() can be coded as an iterative routine, avoiding the recursive call
- It is a special kind of recursion called tail recursion, where nothing remains to be done after the recursive call
- Many recursive problem formuations lack a tail recursive version
- Tail recursion combines the elegance of recursion and the efficiency of iteration

Recursive functions for fact() and facT()

Editor:

```
int fact(int n) {
  if (n != 0)
    return n*fact(n-1);
  else return 1;
}
```

Editor:

```
int facT(int n, int p) {
// first call: facT(n, 1);
  if (n != 0)
    return facT(n-1, n*p);
  else return p;
}
```

- Both formulations can be coded recursively, but facT() can be coded as an iterative routine, avoiding the recursive call
- It is a special kind of recursion called tail recursion, where nothing remains to be done after the recursive call
- Many recursive problem formuations lack a tail recursive version
- Tail recursion combines the elegance of recursion and the efficiency of iteration

Recursive functions for fact() and facT()

Editor:

```
int fact(int n) {
  if (n != 0)
    return n*fact(n-1);
  else return 1;
}
```

Editor:

```
int facT(int n, int p) {
// first call: facT(n, 1);
  if (n != 0)
    return facT(n-1, n*p);
  else return p;
}
```

- Both formulations can be coded recursively, but facT() can be coded as an iterative routine, avoiding the recursive call
- It is a special kind of recursion called tail recursion, where nothing remains to be done after the recursive call
- Many recursive problem formuations lack a tail recursive version
- Tail recursion combines the elegance of recursion and the efficiency of iteration

Iterative computation of facT()

Basis $\mathrm{facT}(0, p) = p$

Induction $\mathrm{facT}(n, p) = \mathrm{facT}(n-1, n \times p),\ n > 0$

fact() in terms of facT() $\mathrm{fact}(n) = \mathrm{facT}(n, 1)$

Iterative routine for facT(n, p)

```
facT(int n, int p) {
// handle the induction, if n > 0
while (n>0) {
  preparation to compute facT(n-1, p × n), next
  p = p*n; n=n-1;
} // carry on until n = 0
// inductive steps are now over
// now compute facT(0, p) -- trivial
return p; // as p is the result
}
```

Handling tail recursion (base cases coming last)

$\text{trR}(p_1, \ldots, p_n)$

Induction $[C_{I,1}]$
$A_{I,1};$
ret
$\text{trR}(p_{I_1,1}, \ldots, p_{I_1,n})$

Induction $[C_{I,2}]$
$A_{I,2};$
ret
$\text{tr}(p_{I_2,1}, \ldots, p_{I_2,n})$

...

Basis $[C_{B,1}]$
$A_{B,1};$ ret b_1

Basis $[C_{B,2}]$
$A_{B,2};$ ret b_2

...

Iterative routine for trR()

```
trR(p1, ..., pn) {
  while (1) { handle induction
    if (C_I,1) {
      code for A_I,1;
      p1=pI11=; ...; pn=pI11;
    } else if (C_I,2) {
      code for A_I,2;
      p1=pI21=; ...; pn=pI21;
    } else if ...
    else break;
  } // inductive steps over
  if (C_B,1) { // base conditions
    code for A_B,1; return b1;
  } else if (C_B,2) { ...
    code for A_B,2; return b2;
  } ...
}
```

Handling tail recursion (base cases coming last)

$\text{trR}(p_1, \ldots, p_n)$

Induction $[C_{I,1}]$
$A_{I,1}$;
ret
$\text{trR}(p_{I_1,1}, \ldots, p_{I_1,n})$

Induction $[C_{I,2}]$
$A_{I,2}$;
ret
$\text{tr}(p_{I_2,1}, \ldots, p_{I_2,n})$

...

Basis $[C_{B,1}]$
$A_{B,1}$; ret b_1

Basis $[C_{B,2}]$
$A_{B,2}$; ret b_2

...

Iterative routine for trR()

```
trR(p1, ..., pn) {
  while (1) { handle induction
    if (C_{I,1}) {
      code for A_{I,1};
      p1=pI11=; ...; pn=pI11;
    } else if (C_{I,2}) {
      code for A_{I,2};
      p1=pI21=; ...; pn=pI21;
    } else if ...
    else break;
  } // inductive steps over
  if (C_{B,1}) { // base conditions
    code for A_{B,1}; return b1;
  } else if (C_{B,2}) { ...
    code for A_{B,2}; return b2;
  } ...
}
```

Greatest of many numbers

Consider a sequence of numbers: $x_i, 1 \leq i \leq n$, it is necessary to identify the greatest number in this sequence.

Let m_i denote the max of the sequence of length n

Basis $m_1 = x_1$, as the first number is sequence of length 1

Induction $m_i = \max(m_{i-1}, x_i)$, for $i > 1$

In this tail recursion the base case comes first!

Editor:

```
#include <stdio.h>
main() {
 int n, i, x, mx;
 printf ("enter n: ");
 scanf ("%d", &n);
 scanf ("%d", &x);
 mx = x; // m1 = x
 for (i=1; i<n ; i++) {
 // handle remaining n-1 nos
  scanf ("%d", &x);
  if (x > mx) mx = x;
  // mi =max(mi-1, xi)
 }
 printf ("max: %d\n", mx);
}
```

Greatest of many numbers

Consider a sequence of numbers: $x_i, 1 \leq i \leq n$, it is necessary to identify the greatest number in this sequence.

Let m_i denote the max of the sequence of length n

Basis $m_1 = x_1$, as the first number is sequence of length 1

Induction $m_i = \max(m_{i-1}, x_i)$, for $i > 1$

In this tail recursion the base case comes first!

Editor:

```
#include <stdio.h>
main() {
 int n, i, x, mx;
 printf ("enter n: ");
 scanf ("%d", &n);
 scanf ("%d", &x);
 mx = x; // m1 = x
 for (i=1; i<n ; i++) {
 // handle remaining n-1 nos
  scanf ("%d", &x);
  if (x > mx) mx = x;
  // mi =max(mi-1, xi)
 }
 printf ("max: %d\n", mx);
}
```

Syllabus (Theory)

Introduction to the Digital Computer;
Introduction to Programming – Variables, Assignment; Expressions; Input/Output;
Conditionals and Branching; Iteration;
Functions; Recursion; Arrays; Introduction to Pointers; Strings; Structures;
Introduction to Data-Procedure Encapsulation;
Dynamic allocation; Linked structures;
Introduction to Data Structure – Stacks and Queues; Searching and Sorting; Time and space requirements.

Part VIII

Strings

Section outline

23 **Strings**

- Character strings
- Common string functions
- Reading a string

Character strings

- Strings are arrays of characters
- `char name[10];`
-

R	a	m	e	s	h	'\0'			

- At most 10 characters may be stored in `name` – including the `'\0'` at the end
- Strings typically store varying numbers of characters
- The end is indicated by the NULL character – `'\0'`
- Any character beyond the first `'\0'` is ignored

Character strings

- Strings are arrays of characters
- `char name[10];`
R	a	m	e	s	h	'\0'			

- At most 10 characters may be stored in `name` – including the `'\0'` at the end
- Strings typically store varying numbers of characters
- The end is indicated by the NULL character – `'\0'`
- Any character beyond the first `'\0'` is ignored

Common string functions

- `int strlen (const char s[]);` – Returns the length (the number of characters before the first NULL character) of the string `s`
- `int strcmp (const char s[], const char t[]);` – Returns 0 if the two strings are identical, a negative value if `s` is lexicographically smaller than `t` (`s` comes before `t` in the standard dictionary order), and a positive value if `s` is lexicographically larger than `t`
- `char *strcpy (char s[], const char t[]);` – Copies the string `t` to the string `s`; returns `s`
- `char *strcat (char s[], const char t[]);` – Appends the string `t` and then the NULL character at the end of `s`; returns `s`

Common string functions

- `int strlen (const char s[]);` – Returns the length (the number of characters before the first NULL character) of the string `s`
- `int strcmp (const char s[], const char t[]);` – Returns 0 if the two strings are identical, a negative value if `s` is lexicographically smaller than `t` (`s` comes before `t` in the standard dictionary order), and a positive value if `s` is lexicographically larger than `t`
- `char *strcpy (char s[], const char t[]);` – Copies the string `t` to the string `s`; returns `s`
- `char *strcat (char s[], const char t[]);` – Appends the string `t` and then the NULL character at the end of `s`; returns `s`

Common string functions

- `int strlen (const char s[]);` – Returns the length (the number of characters before the first NULL character) of the string `s`
- `int strcmp (const char s[], const char t[]);` – Returns 0 if the two strings are identical, a negative value if `s` is lexicographically smaller than `t` (`s` comes before `t` in the standard dictionary order), and a positive value if `s` is lexicographically larger than `t`
- `char *strcpy (char s[], const char t[]);` – Copies the string `t` to the string `s`; returns `s`
- `char *strcat (char s[], const char t[]);` – Appends the string `t` and then the NULL character at the end of `s`; returns `s`

Common string functions

- `int strlen (const char s[]);` – Returns the length (the number of characters before the first NULL character) of the string `s`
- `int strcmp (const char s[], const char t[]);` – Returns 0 if the two strings are identical, a negative value if `s` is lexicographically smaller than `t` (`s` comes before `t` in the standard dictionary order), and a positive value if `s` is lexicographically larger than `t`
- `char *strcpy (char s[], const char t[]);` – Copies the string `t` to the string `s`; returns `s`
- `char *strcat (char s[], const char t[]);` – Appends the string `t` and then the NULL character at the end of `s`; returns `s`

Reading a string

- `char name[10]; scanf("%s", name);` – Note that `name` rather than `&name` is passed (why?); `name` should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array;
  the `m` in the conversion specification `ms` instructs `scanf` that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for `nameDecl`; that is why `&nameDecl` is passed

Reading a string

- `char name[10]; scanf("%s", name);` – Note that `name` rather than `&name` is passed (why?); `name` should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array;
  the `m` in the conversion specification `ms` instructs `scanf` that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for `nameDecl`; that is why `&nameDecl` is passed

Reading a string

- `char name[10]; scanf("%s", name);` – Note that **name** rather than **&name** is passed (why?); **name** should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array; the **m** in the conversion specification **ms** instructs **scanf** that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for **nameDecl**; that is why **&nameDecl** is passed

Reading a string

- `char name[10]; scanf("%s", name);` – Note that **name** rather than **&name** is passed (why?); **name** should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array;
  the **m** in the conversion specification **ms** instructs **scanf** that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for **nameDecl**; that is why **&nameDecl** is passed

Reading a string

- `char name[10]; scanf("%s", name);` – Note that **`name`** rather than **`&name`** is passed (why?); **`name`** should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array;
  the **`m`** in the conversion specification **`ms`** instructs **`scanf`** that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for **`nameDecl`**; that is why `&nameDecl` is passed

Reading a string

- `char name[10]; scanf("%s", name);` – Note that **`name`** rather than **`&name`** is passed (why?); **`name`** should be a large enough array to accommodate the full name and the trailing `'\0'` – real problem if a bigger string is actually supplied (why?)
- `char nameDecl[]; scanf("%ms", &nameDecl);` – the declaration `char nameDecl[];` only allocates a pointer location but not an array;
  the **`m`** in the conversion specification **`ms`** instructs **`scanf`** that it should itself allocate the required space to accommodate the string it reads (and also the trailing `'\0'`); the allocated pointer is placed in the memory location for **`nameDecl`**; that is why **`&nameDecl`** is passed

Program for reading strings

Editor:

```
#include <stdio.h>
int main() {
  char s1[8], *s2;

  printf ("Enter a string of 5 characters or less: ");
  scanf ("%6s", s1); // dangerous if string is larger
  printf ("You typed: %s\n\n", s1);

  printf ("Now enter a string of any length.");
  scanf ("%as", &s2);
  printf ("You typed: %s\n", s2);
return 0; }
```

NB. **`scanf`** only reads a "word" – characters until the next white space

Memory view

s1	00000000	00000000	00000000	00000000
address	3075	3074	3073	**3072**
	00000000	00000000	00000000	00000000
address	3079	3078	3077	**3076**
s2	00000000	00000000	00000000	00000000
address	3083	3082	3081	**3080**
	00000000	00000000	00000000	00000000
address	3087	3086	3085	**3084**
	00000000	00000000	00000000	00000000
address	3091	3090	3089	**3088**

Locations 3072..3079 are allocated to `s1` (`char s1[8]`)

`s2` (`char s2[]`) can store a reference (pointer) to a string (with allocated memory)

Let `scanf`, with `%ms` allocate space at 3088 for storing a string it reads

3088 is then stored at the location for `s2` (3080), because 3080 was passed to `scanf` as `&s2`

Memory view

`s1`	00000000	00000000	00000000	00000000
address	3075	3074	3073	**3072**
	00000000	00000000	00000000	00000000
address	3079	3078	3077	**3076**
`s2`	00000000	00000000	00000000	00000000
address	3083	3082	3081	**3080**
	00000000	00000000	00000000	00000000
address	3087	3086	3085	**3084**
	00000000	00000000	00000000	00000000
address	3091	3090	3089	**3088**

Locations 3072..3079 are allocated to **`s1`** (**`char s1[8]`**)

`s2` (**`char s2[]`**) can store a reference (pointer) to a string (with allocated memory)

Let **`scanf`**, with **`%ms`** allocate space at 3088 for storing a string it reads

3088 is then stored at the location for **`s2`** (3080), because 3080 was passed to **`scanf`** as **`&s2`**

Memory view

s1	00000000	00000000	00000000	00000000
address	3075	3074	3073	**3072**
	00000000	00000000	00000000	00000000
address	3079	3078	3077	**3076**
s2	00000000	00000000	00000000	00000000
address	3083	3082	3081	**3080**
	00000000	00000000	00000000	00000000
address	3087	3086	3085	**3084**
	00000000	00000000	00000000	00000000
address	3091	3090	3089	**3088**

Locations 3072..3079 are allocated to **`s1`** (**`char s1[8]`**)

`s2` (**`char s2[]`**) can store a reference (pointer) to a string (with allocated memory)

Let **`scanf`**, with **`%ms`** allocate space at 3088 for storing a string it reads

3088 is then stored at the location for **`s2`** (3080), because 3080 was passed to **`scanf`** as **`&s2`**

Memory view

s1	00000000	00000000	00000000	00000000
address	3075	3074	3073	**3072**
	00000000	00000000	00000000	00000000
address	3079	3078	3077	**3076**
s2	00000000	00000000	00001100	00001000
address	3083	3082	3081	**3080**
	00000000	00000000	00000000	00000000
address	3087	3086	3085	**3084**
	00000000	00000000	00000000	00000000
address	3091	3090	3089	**3088**

Locations 3072..3079 are allocated to **`s1`** (**`char s1[8]`**)

`s2` (**`char s2[]`**) can store a reference (pointer) to a string (with allocated memory)

Let **`scanf`**, with **`%ms`** allocate space at 3088 for storing a string it reads

3088 is then stored at the location for **`s2`** (3080), because 3080 was passed to **`scanf`** as **`&s2`**

Program for reading strings

Editor:

```
#include <stdio.h>
#define LMAX 85
int main() {
  char line[LMAX];
  printf ("Enter a line of text: ");
  fgets(line, LMAX, stdin); // just accept, for now
  printf ("fgets accepted: %s\n", line);
return 0; }
```

NB. In the above call, **`fgets`** reads at most LMAX-1 characters and terminates the string with **`'\0'`**

The simpler **`gets()`**, eg. **`gets(line)`**, should never be used

Program for reading strings

Editor:

```
#include <stdio.h>
#define LMAX 85
int main() {
  char line[LMAX];
  printf ("Enter a line of text: ");
  fgets(line, LMAX, stdin); // just accept, for now
  printf ("fgets accepted: %s\n", line);
return 0; }
```

NB. In the above call, **`fgets`** reads at most LMAX-1 characters and terminates the string with **`'\0'`**

The simpler **`gets()`**, eg. **`gets(line)`**, should never be used

Section outline

24 **String Examples**

- String length
- Appending one string to another
- Substrings
- Deletion
- Insertion
- Substring replacement
- Str fn prototypes

Length of a string

Recursive version:

$$L(s) = \begin{cases} \text{if } (s[0] = '\backslash 0') \text{ then } 0 & (1) \\ \text{else } 1 + L(s+1) & (2) \end{cases}$$

$$L(s,n) = \begin{cases} \text{if } (s[0] = '\backslash 0') \text{ then } n & (1) \\ \text{else } L(s+1, n+1) & (2) \end{cases}$$

Tail recursive version, called as $l(s, 0)$ (3)

Length of a string (iterative)

Editor:

```
int c_strlen(const char s[]) {
  int n=0; // by clause 3
  while (s[0] != '\0') { // by complement of clause 1
    s++ ; n++; // by clause 2
  }
  return n; // by clause 1 & 2
}
```

Appending one string to another

$$A(s,t,p,q) = \begin{cases} s[p] = t[q] & (1) \\ \text{if } (t[q] = '\backslash 0') \text{ then done} & (2) \\ \text{else } A(s,t,p+1,q+1) & (3) \end{cases}$$

To be called as $A(s,t,L(s),0)$ (4)

String concatenation (iterative)

Editor:

```
void c_strcat(char s[], const char t[]) {
  int p, q=0; // by clause 4
  p = c_strlen(s); // by clause 4
  do {
    s[p] = t[q]; // by clause 1
    if (t[q] == '\0') break; // by clause 2
    p++; q++; // by clause 3
  } while (1);
}
```

Substring identification

$$S(s,t,p,f,m,n) = \begin{cases} \text{if } (n=0) \text{ then } p & (1) \\ \text{else if } (n>m) \text{ then } -1 & (2) \\ \text{else} \\ \begin{cases} \text{if } (s[p]=t[0] \text{ and } S(s,t+1,p+1,0,m-1,n-1) \neq -1) & (3) \\ \text{then } p & (4) \\ \text{else } \begin{cases} \text{if } (f \neq 0) \text{ then } S(s,t,p+1,1,m-1,n) & (5) \\ \text{else } -1 & (6) \end{cases} \end{cases} \end{cases}$$

Use to be called as $S(s,t,0,1,L(s),L(t))$ (7)

f `f`=`0`: matching strictly at `p`

(1) success on reaching end of `t`

(2) failure on reaching end of `s` but not `t`

(3) first char of `t` matches char at postion `p` in `s` and remaining chars of `t` match at position `p+1` in `s`

(4) success if (3) is satisfied

(5) `f`≠`0`: search for match at next position

Substring identification (recursive)

Editor:

```
int c_ss_aux (char s[], const char t[], int p, int f, int
m, int n) {
  if (n==0) return p; // by clause 1
  else if (n > m) return -1; // by clause 2
  else {
    if (s[p] == t[0] && // by clause 3
        c_ss_aux(s, t + 1, p+1, 0, m-1, n-1) != -1)
     return p; // by clause 4
    else {
     if (f!=0) return c_ss_aux(s, t, p + 1, 1, m-1, n);
     // by clause 5
     else return -1; // by clause 6
    }
  }
}
```

Substring identification (Contd.)

Editor:

```
int c_substr (const char s[], const char t[]) {
  return c_ss_aux (s, t, 0, 1,
                   c_strlen(s), c_strlen(t));
          // by clause 7
}
```

Deletion from string

$$D(s,p,n) = \begin{cases} \text{if } (n=0) \text{ done} & (1) \\ \text{else } F(s,p,p+n,L(s+p+n)+1) & (2) \end{cases}$$

- required to delete n characters from postion p in string s
- achieved by shifting the characters starting at $p+n$ to the end of s, including the '“0' character using the shift forward function, defined below
- the total number of characters to be shifted is $L(s+p+n)+1$
- the shift forward functino moves n characters from postion f to postion t ($f \geq t$) s
- definition of F is tail recursive

$$F(s,t,f,n) = \begin{cases} \text{if } (n=0) \text{ done} & (1) \\ \text{else} \\ \begin{cases} s[t] = s[f] & (2) \\ F(s,t+1,f+1,n-1) & (3) \end{cases} \end{cases}$$

Deletion from a string (iterative)

Editor:

```
void c_moveForward (char s[], int t, int f, int n) {
  while (n) { // by complement of clause 1
    s[t] = s[f]; // by clause 2
    t++; f++; n--; // by clause 3
  }
}

void c_delstr (char s[], int p, int n) {
  if (n == 0) return; // by clause 1
  else c_moveForward (s, p, p + n, c_strlen(s+p+n) + 1);
  // by clause 2
}
```

Insertion in a string

$$I(s,t,p) = \begin{cases} \text{Let } n = L(t) & (1) \\ \text{if } (n = 0) \text{ done} & (2) \\ \text{else} & \\ \begin{cases} B(s, p, p+n, L(s+p)+1) & (3) \\ C(s+p, t, L(t)) & (4) \end{cases} \end{cases}$$

- Insert string t at postion p in string s
- Shift backward from postion f to postion t, n characters in f
- Definition of B is tail recursive

$$B(s,f,t,n) = \begin{cases} \text{if } (n = 0) \text{ done} & (1) \\ \text{else} & \\ \begin{cases} s[t+n-1] = s[f+n-1] & (2) \\ B(s,f,t,n-1) & (3) \end{cases} \end{cases}$$

Definition of B is tail recursive

Insertion in a string (iterative)

Editor:

```
void c_copyArr(char s[], const char t[], int n) {
  while (n) { // while characters remain to be copied
    *s = *t; // copy character at t to s
    s++; t++; n--; // s & t to next pos, decr n
  }
}
void c_moveBack(char s[], int f, int t, int n) {
  n--; // to avoid -1 in clause 2
  while (n>=0) {
  // by clause 1 and accounting for the previous n--
    s[t + n] = s[f + n];
    // by clause 2 and accounting for the previous n--
    n--; // by clause 3
  }
}
```

Insertion in a string (iterative) (Contd.)

Editor:

```
void c_instr(char s[], const char t[], int p) {
  int n = c_strlen(t); // by clause 1
  if (n) { // by complement of clause 2
    c_moveBack(s, p, p + n, c_strlen(s + p) + 1);
    // by clause 3
    c_copyArr(s + p, t, n); // by clause 4
  }
}
```

Substring replacement

$$R(s,t,r) = \begin{cases} \text{Let } p = S(s,t,0,1) & (1) \\ \text{if } (p = -1) \text{ absent} & (2) \\ \text{else} & \\ \quad \begin{cases} D(s,p,L(t)) & (3) \\ I(s,r,p) & (4) \\ \text{replaced} & (5) \end{cases} & \end{cases}$$

(1) first find the position where t matches in s

(2) if no match, then nothing to do

(3) delete as many characters there are in t, from position p in s

(4) insert from position p in s, characters in the replacement string r

Substring replacement (Contd.)

Editor:

```
int c_replace(char s[], const char t[], const char r[]) {
  int p = c_substr(s, t); // by clause 1
  if (p == -1) return -1; // by clause 2
  else {
    c_delstr(s, p, c_strlen(t)); // by clause 3
    c_instr(s, r, p); // by clause 4
    return 1; // by clause 5
  }
}
```

Prototypes of our string functions

Editor: `c_string.h`

```
int c_strlen(const char s[]);
void c_strcat(char s[], const char t[]);
int c_substr(const char s[], const char t[]);
int c_replace(char s[], const char t[], const char r[]);
```

Testing string functions

Editor:

```
#include <stdio.h>
#include "c_string.h"
int main() {
  char s[100]="this "; char t[15]="and thar.";
  printf ("length of t=\"%s\" is %d\n", t, c_strlen(t));
  printf ("length of s=\"%s\" is %d\n", t, c_strlen(s));

  c_strcat(s, t);
  printf ("after concatenating t to s: %s\n", s);
  printf ("\"thar\" occurs at position %d in %s\n",
      c_substr (s, "thar"), s);

  c_replace(s, "thar", "that");
  printf ("after correction: %s\n", s);

  printf ("\"thar\" occurs at position %d in %s\n",
      c_substr (s, "thar"), s);
```

String Functions

Editor: Output from program

```
# cc -Wall -o strTest strings.c strTest.c
# ./strTest
length of t="and thar." is 9
length of s="and thar." is 5
after concatenating t to s: this and thar.
"thar" occurs at position 9 in this and thar.
after correction: this and that.
"thar" occurs at position -1 in this and that.
```

Substring Matching at Work

Editor: c_ss_aux(const char s[], const char t[], int p, int f, m, int n)

```
s+p:"this and thar.", t:"thar", p=0, f=1, m=14, n=4
s+p:"his and thar.", t:"har", p=1, f=0, m=13, n=3
s+p:"is and thar.", t:"ar", p=2, f=0, m=12, n=2
s+p:"his and thar.", t:"thar", p=1, f=1, m=13, n=4
s+p:"is and thar.", t:"thar", p=2, f=1, m=12, n=4
s+p:"s and thar.", t:"thar", p=3, f=1, m=11, n=4
s+p:" and thar.", t:"thar", p=4, f=1, m=10, n=4
s+p:"and thar.", t:"thar", p=5, f=1, m=9, n=4
s+p:"nd thar.", t:"thar", p=6, f=1, m=8, n=4
s+p:"d thar.", t:"thar", p=7, f=1, m=7, n=4
s+p:" thar.", t:"thar", p=8, f=1, m=6, n=4
s+p:"thar.", t:"thar", p=9, f=1, m=5, n=4
s+p:"har.", t:"har", p=10, f=0, m=4, n=3
s+p:"ar.", t:"ar", p=11, f=0, m=3, n=2
s+p:"r.", t:"r", p=12, f=0, m=2, n=1
s+p:".", t:"", p=13, f=0, m=1, n=0
"thar" occurs at position 9 in "this and thar."
```

Remove whitespace preceeding punctuation marks

Blanks and tabs preceeding commas, semicolons and periods are to be removed using the functions described earlier.

Substring Identification Revisited

$$S(s,t,p,m,n) = \begin{cases} \text{if } (n=0) \text{ then } p & (1) \\ \text{else if } (n>m) \text{ then } -1 & (2) \\ \text{else} \\ \begin{cases} \text{if } (s[p]=t[0] \text{ and } T(s+p+1,t+1,0,n-1) \neq -1) & (3) \\ \text{then } p & (4) \\ \text{else } S(s,t,p+1,m-1,n) & (5) \end{cases} \end{cases}$$

- To be called as $S(s,t,0,L(s),L(t))$ (6)
- $T(s+p+1,t+1,0,n-1)$ looks for a match of $t+1$ (having $n-1$ characters) exactly at $s+p+1$
- Now S is tail recursive

Substring Identification Revisited (code)

Editor:

```
int c_substr_I(const char s[], const char t[]) {
int m=c_strlen(s), n=c_strlen(t), p=0; // by clause 6
while (n != 0) { // by complement of clause 1
  if (n > m) return -1; // by clause 2
  if (s[p]==t[0] && c_ss2(s+p+1, t+1, 0, n-1)!=1)
  // by clause 3
   return p; // by clause 4
  else {
   p++; m--; // by clause 5
  }
}
return p; // by clauses 1 & 4
}
```

Match at fixed position

$$T(u,v,q,l) = \begin{cases} \text{if } (l=0) \text{ then } 1 & (1) \\ \text{else} & \\ \begin{cases} \text{if } (s[q]=t[q]) & (2) \\ \text{then } T(u,v,q+1,l-1) & (3) \\ \text{else } -1 & (4) \end{cases} \end{cases}$$

- To be called as $S(s,t,0,L(t))$ (5)
- T is tail recursive

Match at fixed position (code)

Editor:

```
int c_ss2(const char u[], const char v[], int l) {
int q=0; // by clause 5
while (l != 0) { // by complement of clause 1
  if (u[q]==v[q]) { // by clause 2
   q++; l--; // by clause 3
  } else
   return -1; // by clause 4
}
return 1; // by clauses 1
}
```

Optional Code Optimisation

Editor:

```
int c_substr_I(const char s[], const char t[]) {
int m=c_strlen(s), n=c_strlen(t), p=0; // by clause 6
while (n != 0) { // by complement of clause 1
  if (n > m) return -1; // by clause 2
  if (s[p]==t[0]) {
   if (c_ss2(s+p+1, t+1, 0, n-1)!=1)
   return p;
   else {
   p++; m--;
  } else {
   p++; m--;
  }
}
return p; // by clauses 1 & 4
}
```

Optional Code Optimisation

Editor:

```
int c_substr_2(const char s[], const char t[]) {
int m=c_strlen(s), n=c_strlen(t), p=0; // by clause 6
while (n != 0) { // by complement of clause 1
  if (n > m) return -1; // by clause 2
  if (s[p]==t[0]) {
    const char *u=s+p+1, *v=t+1; int l=n-1;
    int q=0;
    while (l != 0) {
      if (u[q]==v[q]) {
        q++; l--;
      } else {
        p++; m--; break; // instead of return -1
      }
    }
    if (l==0) return p; // instead of return 1
  } else {
   p++; m--;
  }
```

Part IX

Searching and simple sorting

Section outline

25. **Fast searching**
    - Binary search formulation
    - Example
    - Rec, indices
    - Rec, indices, fail pos
    - Rec, splitting
    - Rec, splitting, fail pos
    - Iter, indices, fail pos

Searching in a sorted array

- Numbers in the array are sorted in ascending order
  - If the array is empty, then report failure
  - Compare the key to the middle element
  - If equal, then done
  - else, if key is smaller than middle element, then search in upper half
  - else, search in lower half

Searching in a sorted array

- Numbers in the array are sorted in ascending order
  - If the array is empty, then report failure
  - Compare the key to the middle element
  - If equal, then done
  - else, if key is smaller than middle element, then search in upper half
  - else, search in lower half

Searching in a sorted array

- Numbers in the array are sorted in ascending order
  - If the array is empty, then report failure
  - Compare the key to the middle element
  - If equal, then done
  - else, if key is smaller than middle element, then search in upper half
  - else, search in lower half

Searching in a sorted array

- Numbers in the array are sorted in ascending order
    - If the array is empty, then report failure
    - Compare the key to the middle element
    - If equal, then done
    - else, if key is smaller than middle element, then search in upper half
    - else, search in lower half

Searching in a sorted array

- Numbers in the array are sorted in ascending order
  - If the array is empty, then report failure
  - Compare the key to the middle element
  - If equal, then done
  - else, if key is smaller than middle element, then search in upper half
  - else, search in lower half

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Searching in a sorted array

	23?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	23?
0	03
1	23
2	27

	24?
0	03
1	23
2	27
3	38
4	53
5	58
6	85

	24?
0	03
1	23
2	27

	24?
2	27

	24?

Binary search – recursive, array indices

Editor: Ranges by array index

```
int searchBinRI(int Z[], int ky, int is, int ie) {
// is: starting index, ie: ending index
// invoked as: searchBinRI(A, ky, 0, SIZE-1)
 int mid=is+(ie-is)/2;
 if (is>ie) {
  return -1; // empty array
 } else if (ky==Z[mid]) {
  return mid;
 } else if (ky<Z[mid]) { // search in upper half
  return searchBinRI(Z, ky, is, mid-1);
 } else { // search in lower half
  return searchBinRI(Z, ky, mid+1, ie);
 }
}
```

Binary search – recursive, array indices, where failed

Editor: Ranges by array index, failure position

```
int searchBinRIF(int Z[], int ky, int is, int ie) {
// is: starting index, ie: ending index
// invoked as: searchBinRIF(A, ky, 0, SIZE-1)
  int mid=is+(ie-is)/2;
  if (is>ie) {
    return -is-10; // empty array
  } else if (ky==Z[mid]) {
    return mid;
  } else if (ky<Z[mid]) { // search in upper half
    return searchBinRIF(Z, ky, is, mid-1);
  } else { // search in lower half
    return searchBinRIF(Z, ky, mid+1, ie);
  }
}
```

Searching in a sorted array

index	address	size of part
0	Z	
1	Z+1	
...	...	
mid-1	Z+mid-1	mid
mid	Z+mid	
mid+1	Z+mid+1	
...	...	
SIZE-1	Z+SIZE-1	SIZE-mid-1

Binary search – recursive, address arithmetic

Editor: Ranges by address arithmetic

```
int searchBinRA(int Z[], int ky, int sz, int pos) {
// invoked as: searchBinRA(A, ky, SIZE, 0)
 int mid=sz/2;
 if (sz<=0) { // array is empty
  return -1;
 } else if (ky==Z[mid]) {
  return pos+mid;
 } else if (ky<Z[mid]) { // search in upper half
  return searchBinRA(Z, ky, mid, pos);
 } else { // search in lower half
  return searchBinRA(Z+mid+1, ky, sz-mid-1, pos+mid+1);
 }
}
```

Binary search – recursive, addresses, where failed

Editor: Ranges by address arithmetic, failure position

```
int searchBinRAF(int Z[], int ky, int sz, int pos) {
// invoked as: searchBinRAF(A, ky, SIZE, 0)
 int mid=sz/2;
 if (sz<=0) {
  return -pos-10;
 } else if (ky==Z[mid]) {
  return pos+mid;
 } else if (ky<Z[mid]) { // search in upper half
  return searchBinRAF(Z, mid, ky, pos);
 } else { // search in lower half
  return searchBinRAF(Z+mid+1, sz-mid-1, ky, pos+mid+1);
 }
}
```

Compiling tail recursive binary search

- To generate optimised code where tail recursion is eliminated:
  \# gcc -Wall -O2 -o search search.c
- To generate optimised assembler code without tail recursion:
  \# gcc -Wall -O2 -S search.c
- To view assembler code:
  \# gvim search.s
- Search for searchBinRAF or searchBinRAF in vi or gvim:
  /searchB.*R.F←↩
- Search for next occurence of pattern in vi or gvim:
  n
- What to look for?
  Inside searchBinRAF: **`call searchBinRAF`**
  Inside searchBinRIF: **`call searchBinRIF`**
- If these calls are absent inside functions **`searchBinRAF`** and **`searchBinRAF`**, respectively, then these functions are not recursive

Binary search – recursive, array indices, where failed

Run Results:

```
int A[5]=1, 3, 5, 7, 9;

RAF: 1 found at 0
RIF: 1 found at 0
RAF: 7 found at 3
RIF: 7 found at 3
RAF: search for 0 failed at 0
RIF: search for 0 failed at 0
RAF: search for 2 failed at 1
RIF: search for 2 failed at 1
RAF: search for 10 failed at 5
RIF: search for 10 failed at 5
```

Calling program for binary search functions

Editor:

```
#include <stdio.h>
int main() {
 int A[5]=1, 3, 5, 7, 9, ky, pos;

 ky = 1 ; pos = searchBinRAF(A, ky, 5, 0);
 printf(pos<0 ? "RAF: search for %d failed at %d\n"
        :"RAF: %d found at %d\n",
        ky, pos<0 ? -(pos+10):pos);
 ky = 1 ; pos = searchBinRIF(A, ky, 0, 4);
 printf(pos<0 ? "RIF: search for %d failed at %d\n"
        :"RIF: %d found at %d\n",
        ky, pos<0 ? -(pos+10):pos);
 return 0;
}
```

Binary search – iterative, array indices, where failed

Editor:

```
int searchBinIIF(int Z[], int ky, int sz,) {
int is=0;
int ie=sz-1;
while (is <= ie) do { // exit loop on failure
  int mid=is+(ie-is)/2;
  if (ky==Z[mid]) break; // exit loop on match
  else if (ky<Z[mid]) // search in upper half
    ie = mid - 1;
  else // search in lower half
    is = mid - 1;
}
if (is>ie)
  return -is-10; // failure
else
  return mid; // matched at mid
```

Section outline

26 **Simple sorting**

- Selection Sort
- Bubble Sort
- Insertion Sort

Motivation of Selection Sort

- Select smallest element
- Interchange with top element
- Repeat procedure leaving out the top element

Recursive Selection Sort

Editor:

```
void selectionSortR(int Z[], int sz) {
 int sel, i, t;
   if (sz<=0) return;
   for (i=sz-1,minI=i,i--;i=>0;i--)
    // select the smallest element
    if (Z[i]<Z[minI]]) minI = i;
    // interchange the min element with the top element
    t=Z[minI];
    Z[minI]=Z[0];
    Z[0]=t;
    // now sort the rest of the array
    selectionSortR(Z+1, sz-1);
}
```

Iterative Selection Sort

Editor:

```
void selectionSortI(int Z[], int sz) {
 int sel, i, t;
 for (j=sz; j>0; j--) { // from full array, decrease
   for (i=sz-1,minI=i,i--;i=>sz-j;i--)
   // sz-j varies from 0 to sz-1 and i from sz-2 to sz-j
    // select the smallest element
    if (Z[i]<Z[minI]) minI = i;
    // interchange the min element with the top element
    t=Z[minI];
    Z[minI]=Z[sz-j];
    Z[sz-j]=t;
    // now sort the rest of the array
 }
}
```

Motivation of Bubble Sort

- Start from the bottom and move upwards
- If an element is smaller than the one over it, then interchange the two
- The smaller element bubbles up
- Smallest element at top at the end of the pass
- Repeat procedure leaving out the top element

Recursive Bubble Sort

Editor:

```
void bubbleSortR(int Z[], int sz) {
 int i;
   if (sz<=0) return;
   for (i=sz-1;i>0;i--)
    // the smallest element bubbles up to the top
    if (Z[i]<Z[i-1]) {
      int t;
      t=Z[i];
      Z[i]=Z[i-1];
      Z[i-1]=t;
    }
    // now sort the rest of the array
    bubbleSortR(Z+1, sz-1);
}
```

Iterative Bubble Sort

Editor:

```
void bubbleSortI(int Z[], int sz) {
 int i, j;
 for (j=sz; j>0; j--) // from full array, decrease
   for (i=sz-1;i>sz-j;i--)
    // the smallest element bubbles up to the top
    if (Z[i]<Z[i-1]) {
      int t;
      t=Z[i];
      Z[i]=Z[i-1];
      Z[i-1]=t;
    }
}
```

Insert sorted

Editor:

```
void insertSorted(int Z[], int ky, int sz) {
// insert ky at the correct place
// original array should have free locations
// sz is number of elements currently in the array
// sz is not the allocated size of the array
 int i, pos=searchBinRAF(Z, ky, sz, 0);
 if (pos<0) pos=-(pos+10);
 // compensation specific to searchBinRAF
 // now shift down all elements from pos onwards
 for (i=sz;i>pos;i--) // start from the end! (why?)
  Z[i]=Z[i-1];
 Z[pos]=ky; // now the desired position is available
}
```

Insertion Sort

Editor:

```
void insertionSort(int Z[], int sz) {
 int i;
 for (i=1;i<sz;i++)
  // elements 0..(i-1) are sorted, element Z[i]
  // is to be placed so that elements 0..i are also
sorted
  insertSorted(Z, Z[i], i);
}
```

Part X

Runtime measures

Section outline

27 **Program complexity**

- Asymptotic Complexity
- Big-O Notation
- Big-Theta Notation
- Big-Omega Notation
- Sample Growth Functions
- Common Recurrences

Asymptotic Complexity

- Suppose we determine that a program takes $8n + 5$ steps to solve a problem of size n
- What is the significance of the 8 and $+5$?
- As n gets large, the $+5$ becomes insignificant
- The 8 is inaccurate as different operations require varying amounts of time
- What is fundamental is that the time is *linear* in n
- *Asymptotic Complexity*: As n gets large, ignore all lower order terms and concentrate on the highest order term only

Asymptotic Complexity (Contd.)

- $8n + 5$ is said to *grow asymptotically* like n
- So does $119n - 45$
- This gives us a simplified approximation of the complexity of the algorithm, leaving out details that become insignificant for larger input sizes

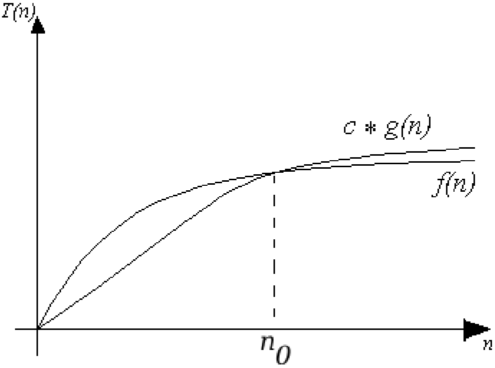
Big-O Notation

- We have talked of $O(n)$, $O(n^2)$ and $O(n^3)$ before
- The Big-O notation is used to express the upper bound on a function, hence used to denote the worst case running time of a program
- If $f(n)$ and $g(n)$ are two functions then we can say:

  $f(n) \in O(g(n))$ if there exists a positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n > n_0$
- $cg(n)$ dominates $f(n)$ for $n > n_0$ (for large n)
- This is read "$f(n)$ is order $g(n)$", or "$f(n)$ is big-O of $g(n)$"
- Loosely speaking, $f(n)$ is no larger than $g(n)$
- Sometimes people also write $f(n) = O(g(n))$, but that notation is misleading, as there is no straightforward equality involved
- This characterisation is not tight, if $f(n) \in O(n)$, then $f(n) \in O(n^2)$

Diagramatic representation of Big-O

$f(n) \in O(g(n))$ if there exists a positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n > n_0$



Big-Theta Notation

- The Big-Theta notation is used to express the notion that a function $g(n)$ is a good (preferably simpler) characterisation of another function $f(n)$
- If $f(n)$ and $g(n)$ are two functions then we can say:

$f(n) \in \Theta(g(n))$ if there exists a positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, for all $n > n_0$

- Loosely speaking, $f(n)$ is like $g(n)$
- Sometimes people also write $f(n) = \Theta(g(n))$, but that notation is misleading
- This characterisation is tight

Big-Omega Notation

- While discussing matrix evaluation by Crammer's ruled we mentioned that the number of operations to be performed is worse that $n!$
- The Big-Omega notation is used to express the lower bound on a function
- If $f(n)$ and $g(n)$ are two functions then we can say:

$f(n) \in \Omega(g(n))$ if there exists a positive constant c and n_0 such that $0 \le cg(n) \le f(n)$, for all $n > n_0$

- $f(n)$ dominates $cg(n)$ for $n > n_0$ (for large n)
- Loosely speaking, $f(n)$ is larger than $g(n)$
- Sometimes people also write $f(n) = \Omega(g(n))$, but that notation is misleading, as there is no straightforward equality involved
- This characterisation is also not tight

Summary

- If $f(n) = \Theta(g(n))$ we say that $f(n)$ and $g(n)$ grow at the same rate asymptotically
- If $f(n) = O(g(n))$ but $f(n) \neq \Omega(g(n))$, then we say that $f(n)$ is asymptotically slower growing than $g(n)$.
- If $f(n) = \Omega(g(n))$ but $f(n) \neq O(g(n))$, then we say that $f(n)$ is asymptotically faster growing than $g(n)$.

Sample Growth Functions

The functions below are given in ascending order:

$O(k) = O(1)$	Constant Time
$O(\log_b n) = O(\log n)$	Logarithmic Time
$O(n)$	Linear Time
$O(n \log n)$	
$O(n^2)$	Quadratic Time
$O(n^3)$	Cubic Time
$\ldots$	
$O(k^n)$	Exponential Time
$O(n!)$	Exponential Time

Sample Recurrences and Their Solutions

$$
\begin{aligned}
T(N) &= 1 && \text{for } N = 1 && (1)\\
T(N) &= T(N-1) + 1 && \text{for } N \geq 2 && (2)
\end{aligned}
$$

$$T(N) = N \in O(N)$$

Show that this recurrence captures the running time complexity of determining the maximum element, searching in an un-sorted array

Sample Recurrences and Their Solutions (Contd.)

$$
\begin{aligned}
T(N) &= 1 && \text{for } N = 1 && (1)\\
T(N) &= T(N-1) + N && \text{for } N \geq 2 && (2)
\end{aligned}
$$

$$T(N) = \frac{N(N+1)}{2} \in O(N^2)$$

Show that this recurrence captures the running time complexity of bubble/insertion/selection sort

Sample Recurrences and Their Solutions (Contd.)

$$T(N) = 1 \qquad \text{for } N = 1 \qquad (1)$$

$$T(N) = T(N/2) + 1 \qquad \text{for } N \geq 2 \qquad (2)$$

$$T(N) = \lg N + 1 \in O(\lg N)$$

Show that this recurrence captures the running time complexity of binary search

Sample Recurrences and Their Solutions (Contd.)

$$T(N) = 0 \qquad \text{for } N = 1 \qquad (1)$$

$$T(N) = T(N/2) + N \qquad \text{for } N \geq 2 \qquad (2)$$

$$T(N) = 2N \in O(N)$$

No problem examined so far in this course whose behaviour is modelled by this recurrence relation

Sample Recurrences and Their Solutions (Contd.)

$$T(N) = 1 \qquad \text{for } N = 1 \qquad (1)$$
$$T(N) = 2T(N/2) + N \qquad \text{for } N \geq 2 \qquad (2)$$

$$T(N) = N \lg N \in O(N \lg N)$$

Show that this recurrence captures the running time complexity of quicksort

Sample Recurrences and Their Solutions (Contd.)

$$T(N) = 1 \qquad \text{for } N = 1 \qquad (1)$$

$$T(N) = 2T(N-1) + 1 \qquad \text{for } N \geq 2 \qquad (2)$$

$$T(N) = 2^N - 1 \in O(2^N)$$

Show that this recurrence captures the running time complexity of the towers of Hanoi problem

Part XI

2D Arrays

Section outline

28 Two dimensional arrays

- Usage
- Element addresses
- Points to note
- Declaring 2D arrays
- Array of arrays

Usage

- `int A[4][5]` – 4×5 array of `int` – four rows and five columns
- Row and column values must be positive integer constants

Usage

- `int A[4][5]` – 4×5 array of `int` – four rows and five columns
- Row and column values must be positive integer constants

Addresses of elements

`int A[4][5]` – `A` has 4 rows and 5 columns

	0	1	2	3	4
0	(0,0)[0]	(0,1)[1]	(0,2)[2]	(0,3)[3]	(0,4)[4]
1	(1,0)[5]	(1,1)[6]	(1,2)[7]	(1,3)[8]	(1,4)[9]
2	(2,0)[10]	(2,1)[11]	(2,2)[12]	(2,3)[13]	(2,4)[14]
3	(3,0)[15]	(3,1)[16]	(3,2)[17]	(3,3)[18]	(3,4)[19]

`int A[R][C]` address of location (i, j)?:$i \times C + j$

	0	1	2	3	4
0	$0 \times 5 + 0$	$0 \times 5 + 1$	$0 \times 5 + 2$	$0 \times 5 + 3$	$0 \times 5 + 4$
1	$1 \times 5 + 0$	$1 \times 5 + 1$	$1 \times 5 + 2$	$1 \times 5 + 3$	$1 \times 5 + 4$
2	$2 \times 5 + 0$	$2 \times 5 + 1$	$2 \times 5 + 2$	$2 \times 5 + 3$	$2 \times 5 + 4$
3	$3 \times 5 + 0$	$3 \times 5 + 1$	$3 \times 5 + 2$	$3 \times 5 + 3$	$3 \times 5 + 4$

`A[i][j]` $\equiv$ `*((int *)A+i*C+j)`

`&A[i][j]` $\equiv$ `((int *)A+i*C+j)`

Addresses of elements

`int A[4][5]` – `A` has 4 rows and 5 columns

	0	1	2	3	4
0	(0,0)[0]	(0,1)[1]	(0,2)[2]	(0,3)[3]	(0,4)[4]
1	(1,0)[5]	(1,1)[6]	(1,2)[7]	(1,3)[8]	(1,4)[9]
2	(2,0)[10]	(2,1)[11]	(2,2)[12]	(2,3)[13]	(2,4)[14]
3	(3,0)[15]	(3,1)[16]	(3,2)[17]	(3,3)[18]	(3,4)[19]

`int A[R][C]` address of location (i, j)?: $i \times C + j$

	0	1	2	3	4
0	$0 \times 5 + 0$	$0 \times 5 + 1$	$0 \times 5 + 2$	$0 \times 5 + 3$	$0 \times 5 + 4$
1	$1 \times 5 + 0$	$1 \times 5 + 1$	$1 \times 5 + 2$	$1 \times 5 + 3$	$1 \times 5 + 4$
2	$2 \times 5 + 0$	$2 \times 5 + 1$	$2 \times 5 + 2$	$2 \times 5 + 3$	$2 \times 5 + 4$
3	$3 \times 5 + 0$	$3 \times 5 + 1$	$3 \times 5 + 2$	$3 \times 5 + 3$	$3 \times 5 + 4$

`A[i][j]` $\equiv$ `*((int *)A+i*C+j)`

`&A[i][j]` $\equiv$ `((int *)A+i*C+j)`

Addresses of elements

int `A[4][5]` – `A` has 4 rows and 5 columns

	0	1	2	3	4
0	(0,0)[0]	(0,1)[1]	(0,2)[2]	(0,3)[3]	(0,4)[4]
1	(1,0)[5]	(1,1)[6]	(1,2)[7]	(1,3)[8]	(1,4)[9]
2	(2,0)[10]	(2,1)[11]	(2,2)[12]	(2,3)[13]	(2,4)[14]
3	(3,0)[15]	(3,1)[16]	(3,2)[17]	(3,3)[18]	(3,4)[19]

int `A[R][C]` address of location (i, j)?:$i \times C + j$

	0	1	2	3	4
0	$0 \times 5 + 0$	$0 \times 5 + 1$	$0 \times 5 + 2$	$0 \times 5 + 3$	$0 \times 5 + 4$
1	$1 \times 5 + 0$	$1 \times 5 + 1$	$1 \times 5 + 2$	$1 \times 5 + 3$	$1 \times 5 + 4$
2	$2 \times 5 + 0$	$2 \times 5 + 1$	$2 \times 5 + 2$	$2 \times 5 + 3$	$2 \times 5 + 4$
3	$3 \times 5 + 0$	$3 \times 5 + 1$	$3 \times 5 + 2$	$3 \times 5 + 3$	$3 \times 5 + 4$

`A[i][j]` $\equiv$ `*((int *)A+i*C+j)`

`&A[i][j]` $\equiv$ `((int *)A+i*C+j)`

Addresses of elements

`int A[4][5]` – `A` has 4 rows and 5 columns

	0	1	2	3	4
0	(0,0)[0]	(0,1)[1]	(0,2)[2]	(0,3)[3]	(0,4)[4]
1	(1,0)[5]	(1,1)[6]	(1,2)[7]	(1,3)[8]	(1,4)[9]
2	(2,0)[10]	(2,1)[11]	(2,2)[12]	(2,3)[13]	(2,4)[14]
3	(3,0)[15]	(3,1)[16]	(3,2)[17]	(3,3)[18]	(3,4)[19]

`int A[R][C]` address of location (i, j)?:$i \times C + j$

	0	1	2	3	4
0	$0 \times 5 + 0$	$0 \times 5 + 1$	$0 \times 5 + 2$	$0 \times 5 + 3$	$0 \times 5 + 4$
1	$1 \times 5 + 0$	$1 \times 5 + 1$	$1 \times 5 + 2$	$1 \times 5 + 3$	$1 \times 5 + 4$
2	$2 \times 5 + 0$	$2 \times 5 + 1$	$2 \times 5 + 2$	$2 \times 5 + 3$	$2 \times 5 + 4$
3	$3 \times 5 + 0$	$3 \times 5 + 1$	$3 \times 5 + 2$	$3 \times 5 + 3$	$3 \times 5 + 4$

`A[i][j]` $\equiv$ `*((int *)A+i*C+j)`

`&A[i][j]` $\equiv$ `((int *)A+i*C+j)`

Addresses of elements

`int A[4][5]` – `A` has 4 rows and 5 columns

	0	1	2	3	4
0	(0,0)[0]	(0,1)[1]	(0,2)[2]	(0,3)[3]	(0,4)[4]
1	(1,0)[5]	(1,1)[6]	(1,2)[7]	(1,3)[8]	(1,4)[9]
2	(2,0)[10]	(2,1)[11]	(2,2)[12]	(2,3)[13]	(2,4)[14]
3	(3,0)[15]	(3,1)[16]	(3,2)[17]	(3,3)[18]	(3,4)[19]

`int A[R][C]` address of location (i, j)?:$i \times C + j$

	0	1	2	3	4
0	$0 \times 5 + 0$	$0 \times 5 + 1$	$0 \times 5 + 2$	$0 \times 5 + 3$	$0 \times 5 + 4$
1	$1 \times 5 + 0$	$1 \times 5 + 1$	$1 \times 5 + 2$	$1 \times 5 + 3$	$1 \times 5 + 4$
2	$2 \times 5 + 0$	$2 \times 5 + 1$	$2 \times 5 + 2$	$2 \times 5 + 3$	$2 \times 5 + 4$
3	$3 \times 5 + 0$	$3 \times 5 + 1$	$3 \times 5 + 2$	$3 \times 5 + 3$	$3 \times 5 + 4$

`A[i][j]` $\equiv$ `*((int *)A+i*C+j)`

`&A[i][j]` $\equiv$ `((int *)A+i*C+j)`

Array facts – for 'C'

- Array elements are stored in memory, one element after another
- Two dimensional arrays are also stored the same way – in row major order – one row after another
- Size of a single dimensional array not required to compute element addresses – both declarations `Z[]` and `Z[SIZE]` work
- Column size of a two dimensional array (but not the row size) of a two dimensional array is required to compute element addresses – both declarations `Z[][COL]` and `Z[ROW][COL]` work, but `Z[][]` does not work
- Array bounds are not checked – `int A[5]; A[8]=0;` is usually accepted by the compiler, but it over writes memory locations outside the array region – serious problem

Array facts – for ‘C’

- Array elements are stored in memory, one element after another
- Two dimensional arrays are also stored the same way – in row major order – one row after another
- Size of a single dimensional array not required to compute element addresses – both declarations `Z[]` and `Z[SIZE]` work
- Column size of a two dimensional array (but not the row size) of a two dimensional array is required to compute element addresses – both declarations `Z[][COL]` and `Z[ROW][COL]` work, but `Z[][]` does not work
- Array bounds are not checked – `int A[5]; A[8]=0;` is usually accepted by the compiler, but it over writes memory locations outside the array region – serious problem

Array facts – for 'C'

- Array elements are stored in memory, one element after another
- Two dimensional arrays are also stored the same way – in row major order – one row after another
- Size of a single dimensional array not required to compute element addresses – both declarations `Z[]` and `Z[SIZE]` work
- Column size of a two dimensional array (but not the row size) of a two dimensional array is required to compute element addresses – both declarations `Z[][COL]` and `Z[ROW][COL]` work, but `Z[][]` does not work
- Array bounds are not checked – `int A[5]; A[8]=0;` is usually accepted by the compiler, but it over writes memory locations outside the array region – serious problem

Array facts – for 'C'

- Array elements are stored in memory, one element after another
- Two dimensional arrays are also stored the same way – in row major order – one row after another
- Size of a single dimensional array not required to compute element addresses – both declarations `Z[]` and `Z[SIZE]` work
- Column size of a two dimensional array (but not the row size) of a two dimensional array is required to compute element addresses – both declarations `Z[][COL]` and `Z[ROW][COL]` work, but `Z[][]` does not work
- Array bounds are not checked – `int A[5]; A[8]=0;` is usually accepted by the compiler, but it over writes memory locations outside the array region – serious problem

Array facts – for ‘C’

- Array elements are stored in memory, one element after another
- Two dimensional arrays are also stored the same way – in row major order – one row after another
- Size of a single dimensional array not required to compute element addresses – both declarations `Z[]` and `Z[SIZE]` work
- Column size of a two dimensional array (but not the row size) of a two dimensional array is required to compute element addresses – both declarations `Z[][COL]` and `Z[ROW][COL]` work, but `Z[][]` does not work
- Array bounds are not checked – `int A[5]; A[8]=0;` is usually accepted by the compiler, but it over writes memory locations outside the array region – serious problem

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Summing all elements in an 2-D array

- We definitely need to know the number of columns
- How do we declare the array?
- Can only declare an array for constant dimensions
- Arbitrary arrays cannot be handled via declaration
- Explicit address computation required
- Type of array elements must be fixed
- `#define ADDR2D(C,I,J) C*I+j`
- `#define EL2D(T,Z,C,I,J) *((T*)Z+C*I+j)`

Sum 2D

Editor:

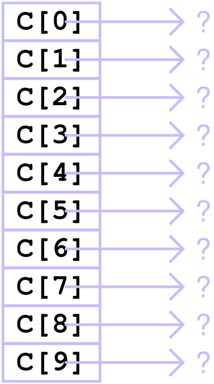
```
#define ADDR2D(C,I,J) (C)*(I)+(J)
int sum2D(int *Z, int R, int C) {
// the 2D array is passed simply as an int pointer
// row and column sizes are passed separately
 int i, j, s=0;
 for (i=0; i<R; i++)
  for (j=0; j<C; j++)
   s += Z[ADDR2D(C,i,j)];
 return s;
}
```

Declaring 2D arrays

- `int A[10][20]` – also definition
- `int B[][20], (*Y)[20]` – only pointer allocation, no array allocation

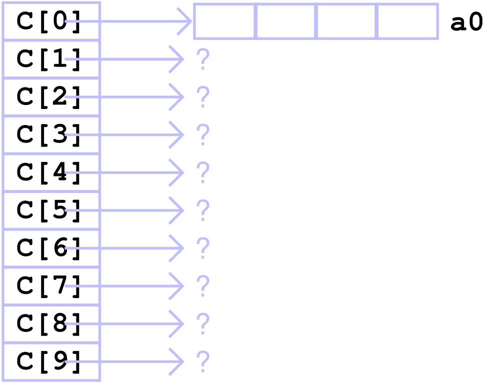
Declaring 2D arrays (Contd.)

- `int *C[10]` – `C` is a vector of integer pointers
- `int **D` – pointer to (a vector of) integer pointer(s)



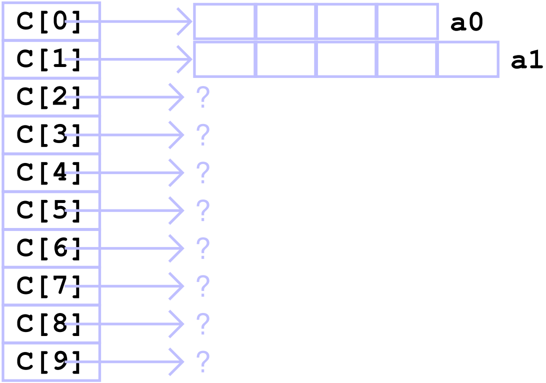
Declaring 2D arrays (Contd.)

- `int *C[10]` – `C` is a vector of integer pointers
- `int a0[4]; C[0]=a0;`



Declaring 2D arrays (Contd.)

- `int *C[10]` – `C` is a vector of integer pointers
- `int a0[4]; C[0]=a0;`
- `int a1[5]; C[1]=a1;`



Handling 2D arrays

Editor: arr.c

```
int main () {
  int i, j;
  int b[3][4], (*r)[4], *q[3];

for (i=0; i<3; i++)
  q[i] = (int *) malloc (4*sizeof(int));

r = (int (*)[4]) malloc (3*4*sizeof(int));

printf("declarations: int b[3][4], (*r)[4], *q[3]\n");
printf ("address of r: %12p, b: %12p, q: %12p\n",
    &r, &b, &q);
printf (" value of r: %12p, b: %12p, q: %12p\n",
    r, b, q);

for (i=0; i<3; i++)
  for (j=0; j<4; j++)
    b[i][j] = q[i][j] = r[i][j] = pow(2,i)*pow(3,j);
```

Handling 2D arrays (Contd.)

Editor: arr.c (Contd.)

```
for (i=0; i<3; i++)
  for (j=0; j<4; j++) {
    printf ("b[%d][%d] = %d\t@ %p \t",
        i, j, b[i][j], &(b[i][j]));
    printf ("b[%d(=%d*4 + %d)] = %d\t",
      i*4+j, i, j, ((int *) b)[i*4+j]);
    printf ("q[%d][%d] = %d\n", i, j, q[i][j]);
    printf ("r[%d(=i)][%d(=j)] = %d \t@ %p\t",
        i, j, r[i][j],
      &(r[i][j]));
    printf ("r[%d(=%d*4 + %d)] = %d\n\n",
      i*4+j, i, j, ((int *) r)[i*4+j]);
  }

return 0;
}
```

Handling 2D arrays (Contd.)

Shell: run of arr

```
$ arr
declarations: int b[3][4], (*r)[4], *q[3]
address of r: 0xbf99948c, b: 0xbf999490, q: 0xbf999480
 values of r: 0x804a088, b: 0xbf999490, q: 0xbf999480
b[0][0] = 1 @ 0xbf999490 b[0(=0*4 + 0)] = 1 q[0][0] = 1
r[0(=i)][0(=j)] = 1 @ 0x804a088 r[0(=0*4 + 0)] = 1

b[0][1] = 3 @ 0xbf999494 b[1(=0*4 + 1)] = 3 q[0][1] = 3
r[0(=i)][1(=j)] = 3 @ 0x804a08c r[1(=0*4 + 1)] = 3

b[0][2] = 9 @ 0xbf999498 b[2(=0*4 + 2)] = 9 q[0][2] = 9
r[0(=i)][2(=j)] = 9 @ 0x804a090 r[2(=0*4 + 2)] = 9

b[0][3] = 27 @ 0xbf99949c b[3(=0*4 + 3)] = 27 q[0][3] = 27
r[0(=i)][3(=j)] = 27 @ 0x804a094 r[3(=0*4 + 3)] = 27
```

Handling 2D arrays (Contd.)

Shell: run of arr

```
b[1][0] = 2 @ 0xbf9994a0 b[4(=1*4 + 0)] = 2 q[1][0] = 2
r[1(=i)][0(=j)] = 2 @ 0x804a098 r[4(=1*4 + 0)] = 2

b[1][1] = 6 @ 0xbf9994a4 b[5(=1*4 + 1)] = 6 q[1][1] = 6
r[1(=i)][1(=j)] = 6 @ 0x804a09c r[5(=1*4 + 1)] = 6

b[1][2] = 18 @ 0xbf9994a8 b[6(=1*4 + 2)] = 18 q[1][2] = 18
r[1(=i)][2(=j)] = 18 @ 0x804a0a0 r[6(=1*4 + 2)] = 18

b[1][3] = 54 @ 0xbf9994ac b[7(=1*4 + 3)] = 54 q[1][3] = 54
r[1(=i)][3(=j)] = 54 @ 0x804a0a4 r[7(=1*4 + 3)] = 54
```

Handling 2D arrays (Contd.)

Shell: run of arr

```
b[2][0] = 4 @ 0xbf9994b0 b[8(=2*4 + 0)] = 4 q[2][0] = 4
r[2(=i)][0(=j)] = 4 @ 0x804a0a8 r[8(=2*4 + 0)] = 4

b[2][1] = 12 @ 0xbf9994b4 b[9(=2*4 + 1)] = 12 q[2][1] = 12
r[2(=i)][1(=j)] = 12 @ 0x804a0ac r[9(=2*4 + 1)] = 12

b[2][2] = 36 @ 0xbf9994b8 b[10(=2*4 + 2)] = 36 q[2][2] = 36
r[2(=i)][2(=j)] = 36 @ 0x804a0b0 r[10(=2*4 + 2)] = 36

b[2][3] = 108 @ 0xbf9994bc b[11(=2*4 + 3)] = 108 q[2][3] =
r[2(=i)][3(=j)] = 108 @ 0x804a0b4 r[11(=2*4 + 3)] = 108
```

Handling 2D arrays (Contd.)

Editor: arr.c

```
#include <stdlib.h>
#include <math.h>

int (*allocate_r())[4]{
int (*r)[4], i, j;
r = (int (*)[4]) malloc (3*4*sizeof(int));

for (i=0; i<3; i++)
for (j=0; j<4; j++) {
r[i][j] = pow(2,i)*pow(3,j);
}
return r;
}
```

Print command-line arguments

Editor: showArgs.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int i;

  for (i=0; i<argc; i++)
    printf("arg-%d: %s\n", i, argv[i]);
return 0;
}
```

Print command-line arguments (Contd.)

Shell: run of showArgs

```
$ make showArgs
cc showArgs.c -o showArgs
$ showArgs arg1 arg2 ... argn
arg-0: showArgs
arg-1: arg1
arg-2: arg2
arg-3: ...
arg-4: argn
```

Section outline

Determinant of a matrix

- Leibniz formula:

$$\det(A) = \sum_{j=1}^{n} A_{i,j} C_{i,j} = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j}$$

- Time complexity of computing the determinant by this mechanism is important.

$$T(n) = \begin{cases} \text{if } (n = 1) \text{ then } 1 \\ \text{otherwise } n \times T(n-1) + N \end{cases}$$

- $T(N)$ is worse than $n!$
- Routines for determinant evaluation by Leibniz formula essentially for programming practice

Determinant of a matrix (Contd.)

Editor: determinant.c

```
int determinant (int N, int A[N][N]) {
  int i, j, k, l, sum=0, sign=1, B[N-1][N-1];
  if (N==1) return A[0][0];
  for (i=0;i<N;i++,sign*=-1) {
    // Now form B
    for (j=0;j<N;j++) {
      if (j==i) continue;
      for (k=1;k<N;k++) {
        l = j<i ? j : j-1;
        B[k-1][l] = A[k][j];
      }
    } // B formed
    sum += sign * A[0][i] * determinant(N-1, B);
  }
  return sum;
}
```

Determinant of a matrix (Contd.)

Editor: determinant.c

```
#include <stdio.h>
#define SIZE 3
int main () {
  int A[SIZE][SIZE], i, j;
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      A[i][j] = (i+1)*(j+1);
      printf ("%4d ", A[i][j]);
    } printf ("\n");
  }
  printf ("determinant of above matrix is %d\n",
      determinant(SIZE, A));
return 0;
}
```

Determinant of a matrix (Contd.)

Shell: run of determinant

```
$ make determinant
cc determinant.c -o determinant
$ determinant
   1    2    3
   2    4    6
   3    6    9
determinant of above matrix is 0
```

Determinant of a matrix (Contd.)

Editor: determinant.c

```
#include <stdio.h>
#define SIZE 3
int main () {
  int A[SIZE][SIZE], i, j;
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      A[i][j] = (i+1)*(j+1) + i*i + j*j;
      printf ("%4d ", A[i][j]);
    } printf ("\n");
  }
  printf ("determinant of above matrix is %d\n",
      determinant(SIZE, A));
return 0;
}
```

Determinant of a matrix (Contd.)

Shell: run of determinant

```
$ make determinant
cc determinant.c -o determinant
$ determinant
   1    3    7
   3    6   11
   7   11   17
determinant of above matrix is -4
```

Determinant of a matrix (Contd.)

Editor: determinant.c

```
int detEval (int N, int A[N][N], char p[N], int M) {
  int i, j, k, l, sum=0, sign=1; // p->present
  if (M==1) return findP(N, A, p);
  for (i=0;i<N;i++) {
    if (p[i]==0) continue; // not present
    p[i] = 0; // skip to compute cofactor
    sum += sign * A[N-M][i] * detEval(N, A, p, M-1);
    p[i] = 1; // re-introduce and continue
    sign *= -1;
  }
  return sum;
}
```

- Marked parts in the code are inefficient
- Avoidable by representing information in **p[]** differently?
- Find a logical solution, as home assignment

Determinant of a matrix (Contd.)

Editor: determinant.c

```
int findP (int N, int A[N][N], char p[N]) {
  int i;
  for (i=0;i<N;i++) {
    if (p[i]) return A[N-1][i] ;
  }
}

int determinant2 (int N, int A[N][N]) {
  char p[N]; int i;
  for (i=0; i<N; i++) p[i]=1;
  return detEval (N, A, p, N);
}
```

Matrix Operations

Matrix Operations

- When two rows or two columns of a matrix are interchanged, the resulting determinant will differ only in sign.
- If you multiply a row or column by a non-zero constant, the determinant is multiplied by that same non-zero constant.
- If you multiply a row or column by a non-zero constant and add it to another row or column, replacing that row or column, there is no change in the determinant.
- Columns to the right of the diagonal element can be eliminated using the above principles to make the matrix *lower triangular*
- Determinant of a triangular matrix is the product of the diagonal elements
- Problem when diagonal element is zero
- Move largest element (among active elements) to the pivot position

Row-Column interchange

Editor:

```
void swapRow (int N, float A[N][N], int r1, int r2) {
  float t; int i;
  for (i=0; i<N; i++) { // swap elements in each col
    t = A[r1][i];
    A[r1][i] = A[r2][i];
    A[r2][i] = t;
  }
}

void swapCol (int N, float A[N][N], int c1, int c2) {
  float t; int i;
  for (i=0; i<N; i++) { // swap elements in each row
    t = A[i][c1];
    A[i][c1] = A[i][c2];
    A[i][c2] = t;
  }
}
```

Time Complexity of Interchange Rows and Columns

For both **`rowSwap`** and **`colSwap`**,

$$T(N) = O(N)$$

Eliminating columns

Editor:

```
void eliminateCols(int N, float A[N][N], int c) {
  float sf; int i, j;
  for (i=c+1; i<N; i++) { // columns after c
    sf = A[c][i]/A[c][c];
#ifdef DEBUG
  printf("eliminateCols: A[%d][%d]=%f, A[%d][%d]=%f,
sf=%f\n",
      c, i, A[c][i], c, c, A[c][c], sf);
#endif
    for (A[c][i]=0, j=c+1; j<N; j++ ) {
    // no change to rows 0..(c-1) with zero elements
      A[j][i] -= sf * A[j][c];
      // no change to sign of determinant
    }
  }
}
```

Time Complexity of Eliminate Columns

On account of the two nested loops,

$$T(N) = O(N^2)$$

Setting pivot

Editor:

```
int setPivot (int N, float A[N][N], int c) {
// move largest element among A[i][j], i, j >= c
// return value: 1: no sign change -1: sign change 0:
A[c][c]==0
  int i, j, mR, mC, sign=1; float max = fabs(A[c][c]);
for (i=c; i<N; i++) // find the max element
  for (j=c; j<N; j++) {
    if (fabs(A[i][j]) > max) {
      max = A[i][j];
      mR = i; mC = j;
    }
  }
#ifdef DEBUG
  printf("setPivot: max=%f, c=%d, mR=%d, mC=%d\n", max,
c, mR, mC);
#endif
}
```

Setting pivot (contd.)

Editor:

```
if (max == 0) return 0;
if (mR != c) { // interchange row, if necessary
  swapRow (N, A, c, mR);
  sign *= -1;
}
if (mC != c) { // interchange row, if necessary
  swapCol (N, A, c, mC);
  sign *= -1;
}
return sign;
}
```

Time Complexity of Setting the Pivot Element

- Maximim element identified in $O(N^2)$ time
- Swapping or rows and columns done in $O(N)$ time
- Overall time complexity is $O(N^2)$

Compute Determinant by Elimination

Editor:

```
float det_byElim (int N, float A[N][N]) {
#ifdef DEBUG
  printf ("det_byElim: address of A=%p\n", A);
#endif
  int i, j, sign=1; float prod=1;
  for (i=0; i<N-1; i++) {
    sign *= setPivot (N, A, i);
#ifdef DEBUG
  showMatrix (N, A, "setPivot: after setPivot");
#endif
    if (sign == 0) return 0;
    prod *= A[i][i];
    eliminateCols(N, A, i);
```

Compute Determinant by Elimination (Contd.)

Editor:

```
#ifdef DEBUG
  printf("det_byElim: sign=%d, prod=%f, A[%d][%d]=%f\n",

      sign, prod, i, i, A[i][i]);
  showMatrix (N, A, "setPivot: after eliminateCols");
#endif
  }
  return sign * prod * A[N-1][N-1];
}
```

Time Complexity of Determinant by Elimination

- `setPivot` called $N-1$ times, each call done in $O(N^2)$ time, hence $O(N^3)$
- `eliminateCols` called $N-1$ times, each call done in $O(N^2)$ time, hence $O(N^3)$
- Overall time complexity is $O(N^3)$ – polynomial in N
- Much better than direct use of Leibniz formula – exponential in N

Compute Determinant by Elimination (Contd.)

Editor:

```
#define SIZE 3
int main () {
  float A[SIZE][SIZE]; int i, j;
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      A[i][j] = (i+1)*(j+1) + i*i + j*j;
      printf ("%f ", A[i][j]);
    } printf ("\n");
  } printf ("***\n");

  printf ("determinant of above matrix (elimination) is
%f\n",
      det_byElim(SIZE, A));
return 0;
}
```

Compute Determinant by Elimination (Contd.)

Shell: Compile and run

```
$ cc -DDEBUG determinant.c -o determinant -lm ;
determinant
1.000000 3.000000 7.000000
3.000000 6.000000 11.000000
7.000000 11.000000 17.000000
***
det_byElim: address of A=0xbfd68804
setPivot: max=17.000000, c=0, mR=2, mC=2
17.000000 11.000000 7.000000
11.000000 6.000000 3.000000
7.000000 3.000000 1.000000
--- setPivot: after setPivot
eliminateCols: A[0][1]=11.000000, A[0][0]=17.000000,
sf=0.647059
eliminateCols: A[0][2]=7.000000, A[0][0]=17.000000,
sf=0.411765
det_byElim: sign=1, prod=17.000000, A[0][0]=17.000000
```

Compute Determinant by Elimination (Contd.)

Shell: Compile and run

```
--- setPivot: after eliminateCols
setPivot: max=-1.882353, c=1, mR=2, mC=2
17.000000 0.000000 0.000000
7.000000 -1.882353 -1.529412
11.000000 -1.529412 -1.117647
--- setPivot: after setPivot
eliminateCols: A[1][2]=-1.529412, A[1][1]=-1.882353,
sf=0.812500
det_byElim: sign=1, prod=-32.000000, A[1][1]=-1.882353
17.000000 0.000000 0.000000
7.000000 -1.882353 0.000000
11.000000 -1.529412 0.125000
--- setPivot: after eliminateCols
determinant of above matrix (elimination) is -3.999996
```

Compute Determinant by Elimination (Contd.)

Shell: Compile and run

```
$ cc determinant.c -o determinant -lm
$ ./determinant
1.000000 3.000000 7.000000
3.000000 6.000000 11.000000
7.000000 11.000000 17.000000
***
determinant of above matrix (elimination) is -3.999996
```

Section outline

30 **More on 2-D arrays**
- Initialisation
- Address arithmetic
- Sizeof
- Type

Initialisation of 2-D Arrays

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[MAXROW][MAXCOL] = {
    { 0,  1,  2,  3,  4},
    {10, 11, 12, 13, 14},
    {20, 21, 22, 23, 24},
    {30, 31, 32, 33, 34},
    {40, 41, 42, 43, 44},
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[MAXROW][MAXCOL] = {
      0,  1,  2,  3,  4,
     10, 11, 12, 13, 14,
     20, 21, 22, 23, 24,
     30, 31, 32, 33, 34,
     40, 41, 42, 43, 44,
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[][MAXCOL] = {
    { 0,  1,  2,  3,  4},
    {10, 11, 12, 13, 14},
    {20, 21, 22, 23, 24},
    {30, 31, 32, 33, 34},
    {40, 41, 42, 43, 44},
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[][MAXCOL] = {
      0,  1,  2,  3,  4,
     10, 11, 12, 13, 14,
     20, 21, 22, 23, 24,
     30, 31, 32, 33, 34,
     40, 41, 42, 43, 44,
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[][MAXCOL] = {
    { 0,  1,  2 },
    {10, 11, 12, 13 },
    {20, 21, 22, 23, 24},
    {30, 31, 32, 33, 34},
    {40, 41, 42, 43, 44},
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[][MAXCOL] = {
    { 0,  1,  2 },
    {10, 11, 12, 13 },
     20, 21, 22, 23, 24,
     30, 31, 32, 33, 34,
     40, 41, 42, 43, 44,
  };
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Editor:

```
#include <stdio.h>
#define MAXROW 5
#define MAXCOL 5
int main() {
  int A[][MAXCOL] = {
    { 0, 1, 2 },
    {10, 11, 12, 13 },
    20, 21, 22, 23, 24,
    30, 31,
  }; A has only four rows
  int i, j;
  for (i=0; i<MAXROW; i++) {
    for (j=0; j<MAXCOL; j++)
      printf ("%3d ", A[i][j]);
    printf ("\n");
  } there is no fifth row
return 0;
}
```

Initialisation of 2-D Arrays (Contd.)

Shell:

```
$ make init2D ; init2D
cc init2D.c -o init2D
   0    1    2    0    0
  10   11   12   13    0
  20   21   22   23   24
  30   31    0    0    0
   4    1 -1079444080 -1079443992 -1210214564
```

NB: Elements of only four rows are properly initialised. Presence of four rows can be inferred from the initialising values that are given in the program.

Address Arithmetic of Arrays Revisited

- `#define N 10`
- `#define R 10`
- `#define C 20`
- `int A[N], B[R][C];`
- Element index of `A[i]` is i
- Address of `A[i]` is `A+i`
- Element index of `B[i][j]` is $C \times i + j$
- Address of `B[i][j]` is `(int *)B + C*i + j`
- Why do we need the type casting?
- What is `A + C*i + j`?

Address Arithmetic of Arrays Revisited (Contd.)

- The number of columns is known in `int A[][C], B[R][C];` NB. those were defined constants
- `A` and `B` are the addresses of the 0^{th} rows of `A` and `B`, respectively
- `A+1` and `B+1` are the addresses of the 1^{st} rows of `A` and `B`, respectively
- `A+i` and `B+i` are the addresses of the i^{th} rows of `A` and `B`, respectively
- The number of bytes in a row are: `C` $\times$ `sizeof(int)`
- `A + C*i + j` does not make sense
- `(int *)A + C*i + j` is okay because `(int *)A` is treated as an int pointer because of the type casting
- Both `A` and `B` are pointer constants of type `int [][C]`

Address Arithmetic of Arrays Revisited (Contd.)

- `int A[][10], B[10][20];`, important: the column size is a constant
- `A+i` and `B+i` are the addresses of the i^{st} rows of `A` and `B`, respectively
- `*(A+i)` and `*(B+i)` are the addresses of the 0^{th} elements of the i^{st} rows of `A` and `B`, respectively
- `*(A+i) + j` and `*(B+i) + j` are the addresses of `A[i][j]` and `B[i][j]`, respectively
- `*(A+i) + j` adds j ints to the address of the 0^{th} element i^{st} row of `A`, and hence is the address of `A[i][j]`
- `&A[i][j]` is also the address of `A[i][j]`
- `*(*(A+i) + j)` is `A[i][j]`
- NB: When the column size is a constant, the above address arithmetic is rarely required

Address Arithmetic of Arrays Revisited (Contd.)

- `int A[][10], B[10][20];`, important: the column size is a constant
- `A+i` and `B+i` are the addresses of the i^{st} rows of `A` and `B`, respectively
- `*(A+i)` and `*(B+i)` are the addresses of the 0^{th} elements of the i^{st} rows of `A` and `B`, respectively
- `*(A+i) + j` and `*(B+i) + j` are the addresses of `A[i][j]` and `B[i][j]`, respectively
- `*(A+i) + j` adds *j* ints to the address of the 0^{th} element i^{st} row of `A`, and hence is the address of `A[i][j]`
- `&A[i][j]` is also the address of `A[i][j]`
- `*(*(A+i) + j)` is `A[i][j]`
- NB: When the column size is a constant, the above address arithmetic is rarely required

2-D Array Address Arithmetic Summary

When the column size is a constant:

- `*(*(A + i) + j)` ≡ `A[i][j]`
- `*(A + i) + j` ≡ `&A[i][j]`
- `*(A[i] + j)` ≡ `A[i][j]`
- `A[i] + j` ≡ `&A[i][j]`
- `(*(A+i))[j]` ≡ `A[i][j]`
- `A + i` ≡ `A[i]`

The last item is useful when trying to work with a sequence of rows of `A` starting at row `i`

Splitting 2-D Arrays

Editor:

```
int searchBinRAF2(int Z[][2], int ky, int sz, int pos) {
// invoked as: searchBinRAF2(A, ky, SIZE, 0)
  int mid=sz/2;
#ifdef DEBUG
  printf ("sz=%d, mid=%d, pos=%d\n", sz, mid, pos);
#endif
  if (sz<=0) {
   return -pos-10;
  } else if (ky==Z[mid][0]) {
   return pos+mid;
  } else if (ky<Z[mid][0]) { // search in upper half
   return searchBinRAF2(Z, ky, mid, pos);
  } else { // search in lower half
   return searchBinRAF2(Z+mid+1, ky, sz-mid-1,
pos+mid+1);
  }
}
```

Splitting 2-D Arrays (Contd.)

Editor:

```
int main(){ int sz=7, ky,pos,i, A2[7][2]={{1, 78},{2, 26},
   {3, 352}, {4, 532}, {5, 272}, {6, 823}, {7, 945}};
ky = 1 ; pos = searchBinRAF2(A2, ky, sz, 0);
printf(pos<0 ? "RAF2: search for %d failed at %d\n":
  "RAF2: %d found at %d\n", ky, pos<0?-(pos+10):pos);
ky = 7 ; pos = searchBinRAF2(A2, ky, sz, 0);
printf(pos<0 ? "RAF2: search for %d failed at %d\n":
  "RAF2: %d found at %d\n", ky, pos<0?-(pos+10):pos);
ky = 0 ; pos = searchBinRAF2(A2, ky, sz, 0);
printf(pos<0 ? "RAF2: search for %d failed at %d\n":
  "RAF2: %d found at %d\n", ky, pos<0?-(pos+10):pos);
ky = 2 ; pos = searchBinRAF2(A2, ky, sz, 0);
printf(pos<0 ? "RAF2: search for %d failed at %d\n":
  "RAF2: %d found at %d\n", ky, pos<0?-(pos+10):pos);
ky = 10 ; pos = searchBinRAF2(A2, ky, sz, 0);
printf(pos<0 ? "RAF2: search for %d failed at %d\n":
  "RAF2: %d found at %d\n", ky, pos<0?-(pos+10):pos);
return 0; }
```

Splitting 2-D Arrays (Contd.)

Shell: compile and run

```
$ make search
cc search.c -o search
$ search
RAF2: 1 found at 0
RAF2: 7 found at 6
RAF2: search for 0 failed at 0
RAF2: 2 found at 1
RAF2: search for 10 failed at 7
```

Handling of `sizeof`

Editor:

```
#include <stdio.h>
void showSize (int R, int C, int A[R][C]) {
  printf ("showSize: R=%d, C=%d, sizeof(A)=%d\n",
    R, C, sizeof(A));
}
int main(){
  int A[3][4], B[4][5];
  showSize(3, 4, A);
  printf ("main: R=%d, C=%d, sizeof(A)=%d\n",
    3, 4, sizeof(A));
  showSize(4, 5, B);
  printf ("main: R=%d, C=%d, sizeof(A)=%d\n",
    4, 5, sizeof(B));
return 0;
}
```

Handling of `sizeof` (Contd.)

Shell: compile and run

```
$ make sizeofArr ; ./sizeofArr
cc sizeofArr.c -o sizeofArr
showSize: R=3, C=4, sizeof(A)=4
main: R=3, C=4, sizeof(A)=48
showSize: R=4, C=5, sizeof(A)=4
main: R=4, C=5, sizeof(A)=80
```

NB. Note the different values of **`sizeof(A)`** reported from **`showSize`** and **`main`**.

Type of `A[R][C]`

- Inside the **`showSize`** function **`A`** is treated as an integer pointer rather than of the type int [][4] or int [][4]
- This may be considered a shortcoming of the current implementation of the gcc compiler
- When the array dimensions (row or column sizes) is variable rather than constants, the type of the array variable is just a pointer of type of the array elements (eg int *)
- When `C` is not a constant "int [][C]" is not well defined
- May lead to problems if address arithmetic is performed assuming that inside **`showSize`** **`A`** is of type "int [][C]"
- But, gcc seems to get it right (program and results next)
- Conclusion: Be very careful with address arithmetic, avoid where possible

Type of `A[R][C]`

- Inside the **`showSize`** function **`A`** is treated as an integer pointer rather than of the type int [][4] or int [][4]
- This may be considered a shortcoming of the current implementation of the gcc compiler
- When the array dimensions (row or column sizes) is variable rather than constants, the type of the array variable is just a pointer of type of the array elements (eg int *)
- When `C` is not a constant "int [][C]" is not well defined
- May lead to problems if address arithmetic is performed assuming that inside **`showSize`** **`A`** is of type "int [][C]"
- But, gcc seems to get it right (program and results next)
- Conclusion: Be very careful with address arithmetic, avoid where possible

Type of `A[R][C]`

- Inside the **`showSize`** function **`A`** is treated as an integer pointer rather than of the type int [][4] or int [][4]
- This may be considered a shortcoming of the current implementation of the gcc compiler
- When the array dimensions (row or column sizes) is variable rather than constants, the type of the array variable is just a pointer of type of the array elements (eg int *)
- When **`C`** is not a constant "int [][C]" is not well defined
- May lead to problems if address arithmetic is performed assuming that inside **`showSize`** **`A`** is of type "int [][C]"
- But, gcc seems to get it right (program and results next)
- Conclusion: Be very careful with address arithmetic, avoid where possible

Type of `A[R][C]`

- Inside the **`showSize`** function **`A`** is treated as an integer pointer rather than of the type int [][4] or int [][4]
- This may be considered a shortcoming of the current implementation of the gcc compiler
- When the array dimensions (row or column sizes) is variable rather than constants, the type of the array variable is just a pointer of type of the array elements (eg int *)
- When **`C`** is not a constant "int [][C]" is not well defined
- May lead to problems if address arithmetic is performed assuming that inside **`showSize`** **`A`** is of type "int [][C]"
- But, gcc seems to get it right (program and results next)
- Conclusion: Be very careful with address arithmetic, avoid where possible

Splitting 2-D Arrays with Variable Column Size (Contd.)

Editor:

```
int searchBinRAFQ(int C, int Z[][C], int ky, int sz,
  int pos) { // invoked as: searchBinRAF2(A, ky, SIZE, 0)
int mid=sz/2;
#ifdef DEBUG
  printf ("sz=%d, mid=%d, pos=%d\n", sz, mid, pos);
#endif
  if (sz<=0) {
   return -pos-10;
  } else if (ky==Z[mid][0]) {
   return pos+mid;
  } else if (ky<Z[mid][0]) { // search in upper half
   return searchBinRAFQ(C, Z, ky, mid, pos);
  } else { // search in lower half
   return searchBinRAFQ(C, Z+mid+1, ky, sz-mid-1, pos+mid+1
  }
}
```

Splitting 2-D Arrays with Variable Column Size (Contd.)

Shell:

```
$ make search ; search
cc search.c -o search
RAFQ: 1 found at 0
RAFQ: 7 found at 6
RAFQ: search for 0 failed at 0
RAFQ: 2 found at 1
RAFQ: search for 10 failed at 7
```

Section outline

Array of strings

- These are arrays of arrays
- `char *strings[5]` – array of 5 strings (un-initialised)
- Each element of `strings` is a string pointer and can be assigned independently
- `char s1[]="first string", s2[]="second string";`
- `strings[0]=s1; strings[1]=s2;`
- `strings[0][1]` is `'i'` – element as position 1 of `strings[0]`
- `strings` is a 1D array of string pointers
- `strings[i]` is a 1D array of characters at position `i` of `strings`, if `strings` is properly initialised

Command-line arguments

Editor: showArgs

```
int main(int argc, char
**argv) {
int i;

for (i=0; i<argc; i++)
  printf
    ("CL arg %d: %s\n",
      i, argv[i]);
return 0;
}
```

- A program can be run with arguments
- **`showArgs arg1 arg2`**
- Total number of arguments is set in **`argc`**
- **`argv`** is an array of strings
- Each command-line argument is set as an entry of **`argv`**

Part XII

Structures and dynamic data types

Section outline

32 **Structures and Type definitions**

- Representing complex numbers
- Using `typedef` for structures
- Structures with functions
- Data type for rationals
- Simple student records

Data Type for complex numbers

- A complex number c can be represented using two real numbers a and b such that $c = a + \mathrm{i}b$
- But can we avoid the overhead of keeping track of two numbers and do with just a single entity?
- Operations also need to be performed on complex numbers (just as they are performed on integers and floating point numbers)
- How well can we do this is ‘C’?
- Not particularly well!
- A single entity can be defined
- Necessary functions can be written
- But those cannot be nicely grouped together – need to keep track of details

Data Type for complex numbers

- A complex number c can be represented using two real numbers a and b such that $c = a + \mathrm{i}b$
- But can we avoid the overhead of keeping track of two numbers and do with just a single entity?
- Operations also need to be performed on complex numbers (just as they are performed on integers and floating point numbers)
- How well can we do this is 'C'?
- Not particularly well!
- A single entity can be defined
- Necessary functions can be written
- But those cannot be nicely grouped together – need to keep track of details

Data Type for complex numbers

- A complex number c can be represented using two real numbers a and b such that $c = a + \mathrm{i}b$
- But can we avoid the overhead of keeping track of two numbers and do with just a single entity?
- Operations also need to be performed on complex numbers (just as they are performed on integers and floating point numbers)
- How well can we do this is ‘C’?
- Not particularly well!
- A single entity can be defined
- Necessary functions can be written
- But those cannot be nicely grouped together – need to keep track of details

Data Type for complex numbers

- A complex number c can be represented using two real numbers a and b such that $c = a + \mathrm{i}b$
- But can we avoid the overhead of keeping track of two numbers and do with just a single entity?
- Operations also need to be performed on complex numbers (just as they are performed on integers and floating point numbers)
- How well can we do this is ‘C’?
- Not particularly well!
- A single entity can be defined
- Necessary functions can be written
- But those cannot be nicely grouped together – need to keep track of details

Structure for complex numbers

Editor:

```c
// declare a structure with two members -- re, im
// structure "tag" is complexTag
struct complexTag {
  double re, im;
}

// declare variables of this type of structure
struct complexTag c1, c2;

// declare pointers to such a structure
struct complexTag *c1P, *c2P;
```

Using `typedef` for structures

Editor:

```
// define a type name for such a structure
typedef struct complexTag complexTyp;

// declare variables of this type of structure
complexTyp c1, c2;

// now a type name for pointers to such a structure
typedef struct complexTag *complexPtr;

// declare pointers to such a structure
complexPtr c1P, c2P;

// direct use of typedef with struct
typedef struct complexTag {
  double re, im;
} complexTyp, *complexPtr;
```

Complex type and functions

Editor:

```
typedef struct complexTag {   // direct use of typedef
  double re, im;
} complexTyp, *complexPtr;

void showComplex (complexTyp a);
complexTyp cnjC (complexTyp a);
complexTyp sclC (complexTyp a, double r);
complexTyp addC (complexTyp a, complexTyp b);
complexTyp subC (complexTyp a, complexTyp b);
complexTyp mulC (complexTyp a, complexTyp b);
complexTyp divC (complexTyp a, complexTyp b);

#include <stdio.h>
void showComplex (complexTyp a) {
  printf ("%e␣+␣i␣%e", a.re, a.im);
}
```

Complex type and functions

Editor:

```
typedef struct complexTag {   // direct use of typedef
  double re, im;
} complexTyp, *complexPtr;

void showComplex (complexTyp a);
complexTyp cnjC (complexTyp a);
complexTyp sclC (complexTyp a, double r);
complexTyp addC (complexTyp a, complexTyp b);
complexTyp subC (complexTyp a, complexTyp b);
complexTyp mulC (complexTyp a, complexTyp b);
complexTyp divC (complexTyp a, complexTyp b);

#include <stdio.h>
void showComplex (complexTyp a) {
  printf ("%e + i %e", a.re, a.im);
}
```

Complex type and functions

Editor:

```
typedef struct complexTag {   // direct use of typedef
  double re, im;
} complexTyp, *complexPtr;

void showComplex (complexTyp a);
complexTyp cnjC (complexTyp a);
complexTyp sclC (complexTyp a, double r);
complexTyp addC (complexTyp a, complexTyp b);
complexTyp subC (complexTyp a, complexTyp b);
complexTyp mulC (complexTyp a, complexTyp b);
complexTyp divC (complexTyp a, complexTyp b);

#include <stdio.h>
void showComplex (complexTyp a) {
  printf ("%e + i %e", a.re, a.im);
}
```

Complex type and functions (Contd.)

Editor:

```
#include <stdio.h>

main() {
  complexTyp a={1,2};
  complexTyp b={3,4};
  printf ("  complex a: "); showComplex(a); printf("\n");
  printf ("  complex b: "); showComplex(b); printf("\n");
  printf (" complex  b: "); showComplex(cnjC(b)); printf("\
  printf ("complex a+b: "); showComplex(addC(a, b)); printf
  printf ("complex a-b: "); showComplex(subC(a, b)); printf
  printf ("complex a*b: "); showComplex(mulC(a, b)); printf
  printf ("complex a/b: "); showComplex(divC(a, b)); printf
}
```

Complex type and functions (Contd.)

Editor:

```
$ ./complex
  complex a: 1.000000e+00_+_i_2.000000e+00
  complex b: 3.000000e+00_+_i_4.000000e+00
 complex  b: 3.000000e+00_+_i_-4.000000e+00
complex a+b: 4.000000e+00_+_i_6.000000e+00
complex a-b: -2.000000e+00_+_i_-2.000000e+00
complex a*b: -5.000000e+00_+_i_9.000000e+00
complex a/b: 4.400000e-01_+_i_4.000000e-02
```

Complex type and functions (Contd.)

Editor:

```
complexTyp cnjC (complexTyp a) {
  complexTyp s;
  s.re = a.re;
  s.im = -a.im;
  return s;
}

complexTyp sclC (complexTyp a, double r) {
  complexTyp s;
  s.re = r * a.re;
  s.im = r * a.im;
  return s;
}
```

Complex type and functions (Contd.)

Editor:

```
complexTyp addC (complexTyp a, complexTyp b) {
  complexTyp s;
  s.re = a.re + b.re;
  s.im = a.im + b.im;
  return s;
}

complexTyp subC (complexTyp a, complexTyp b) {
  complexTyp s;
  s.re = a.re - b.re;
  s.im = a.im - b.im;
  return s;
}
```

Complex type and functions (Contd.)

Editor:

```
complexTyp mulC (complexTyp a, complexTyp b) {
  complexTyp s;
  s.re = a.re * b.re - a.im * b.im;
  s.im = a.re * b.im + a.im + b.re;
  return s;
}

complexTyp divC (complexTyp a, complexTyp b) {
  complexTyp s, d;
  s = mulC(a, cnjC(b));
  d = mulC(b, cnjC(b));
  return sclC(s, 1.0/d.re);
}
```

Rational type and functions

Editor:

```
typedef struct ratTag {
  int nu, de;
} ratTyp, *ratPtr;

void showRat (ratTyp a);
ratTyp redRat (ratTyp a);
ratTyp invRat (ratTyp a);
ratTyp sclRat (ratTyp a, int r);
ratTyp addRat (ratTyp a, ratTyp b);
ratTyp subRat (ratTyp a, ratTyp b);
ratTyp mulRat (ratTyp a, ratTyp b);
ratTyp divRat (ratTyp a, ratTyp b);
```

Rational type and functions

Editor:

```
typedef struct ratTag {
  int nu, de;
} ratTyp, *ratPtr;

void showRat (ratTyp a);
ratTyp redRat (ratTyp a);
ratTyp invRat (ratTyp a);
ratTyp sclRat (ratTyp a, int r);
ratTyp addRat (ratTyp a, ratTyp b);
ratTyp subRat (ratTyp a, ratTyp b);
ratTyp mulRat (ratTyp a, ratTyp b);
ratTyp divRat (ratTyp a, ratTyp b);
```

Rational type and functions (Contd.)

Editor:

```
#include <stdio.h>

main() {
  ratTyp a={1,2};
  ratTyp b={3,4};
  printf ("  rat a: "); showRat(a); printf("\n");
  printf ("  rat b: "); showRat(b); printf("\n");
  printf ("  rat b: "); showRat(redRat(b)); printf("\n");
  printf ("rat 1/b: "); showRat(invRat(b)); printf("\n");
  printf ("rat a+b: "); showRat(addRat(a, b)); printf("\n")
  printf ("rat a-b: "); showRat(subRat(a, b)); printf("\n")
  printf ("rat a*b: "); showRat(mulRat(a, b)); printf("\n")
  printf ("rat a/b: "); showRat(divRat(a, b)); printf("\n")
}
```

Rational type and functions (Contd.)

Editor:

```
$ make rat ; ./rat
cc rat.c -o rat
  rat a: 1/2
  rat b: 3/4
  rat b: 3/4
rat 1/b: 4/3
rat a+b: 5/4
rat a-b: -1/4
rat a*b: 3/8
rat a/b: 2/3
```

Rational type and functions (Contd.)

Editor:

```
int gcd(int a, int b) { // a >= b
  int r;
  if (a < 0) a *= -1;
  if (b < 0) b *= -1;
  if (b < a) {
    r = a; a = b; b = r;
  }
  while (b!=0) {
    r = a % b;
    a=b; b=r;
  }
  return a ;
}
```

Rational type and functions (Contd.)

Editor:

```
void showRat (ratTyp a) {
  printf ("%d/%d", a.nu, a.de);
}

ratTyp invRat (ratTyp a) { // a is reduced
  ratTyp s;
  s.nu = a.de;
  s.de = a.nu;
  return s;
}
```

Rational type and functions (Contd.)

Editor:

```
ratTyp redRat (ratTyp a) {
  int d = gcd(a.nu, a.de);
  ratTyp s;
  s.nu = a.nu / d;
  s.de = a.de / d;
  return s;
}
ratTyp sclRat (ratTyp a, int r) {
  int d = gcd(r, a.de);
  ratTyp s;
  s.nu = a.nu * (r/d);
  s.de = a.de / d;
  return s;
}
```

Rational type and functions (Contd.)

Editor:

```
ratTyp addRat (ratTyp a, ratTyp b) {
  int d = gcd(a.nu, a.de);
  ratTyp s;
  s.nu = a.nu * (b.de/d) + b.nu * (a.de/d);
  s.de = a.de * (b.de/d);
  return redRat(s);
}

ratTyp subRat (ratTyp a, ratTyp b) {
  int d = gcd(a.nu, a.de);
  ratTyp s;
  s.nu = a.nu * (b.de/d) - b.nu * (a.de/d);
  s.de = a.de * (b.de/d);
  return redRat(s);
}
```

Rational type and functions (Contd.)

Editor:

```
ratTyp mulRat (ratTyp a, ratTyp b) {
  int d1 = gcd(a.nu, b.de);
  int d2 = gcd(b.nu, a.de);
  ratTyp s;
  a.nu = a.nu/d1; b.de = b.de/d1;
  b.nu = b.nu/d2; a.de = a.de/d2;
  s.nu = a.nu * b.nu;
  s.de = a.de * b.de;
  return s;
}

ratTyp divRat (ratTyp a, ratTyp b) {
  return mulRat(a, invRat(b));
}
```

Simple Student Records

Editor:

```
typedef struct subInfoTag {
  char subCode[10];
  int credit, gradeWt;
  // Ex: 10, A:9, B:8, C:7, D:6, X,F,I:0
} subInfoTyp, *subInfoPtr;
typedef struct semInfoTag {
  float sgpa, cgpa;
  subInfoPtr sbjA; // unallocated array
  int creditS, nSbj; // initialize to 0
} semInfoTyp, *semInfoPtr;
typedef struct studTag {
  char roll[10];
  char hall[10];
  char *fname, *sname;
  semInfoPtr semA; // unallocated array
  int nSem, semSz; // initialize to 0
} studTyp, *studPtr;
```

Simple Student Records (Contd.)

Editor:

```
main () {
   studTyp s;
   interactiveRegStud(&s); displayRegStud(s);
   interactiveSemStud(&s); displaySemStud(s);
}
```

Editor: stud.dat

```
Rakesh Kumar 07SI2035 MMM
3
CS1101 5 10
EC1101 5  9
CE1101 3  8
```

Simple Student Records (Contd.)

Shell:

```
$ make studRec ; ./studRec <stud.dat
cc     studRec.c   -o studRec
First name? Surname? Roll number? Hall code? First name: Ra
Surname: Kumar
Roll number: 07SI2035
Hall code: MMM
Semesters:  0
Number of subjects? subCode? credit? gradWt? subCode? credi
subCode credit  gradeWt
CS1101    5         10
EC1101    5          9
CE1101    3          8
```

Simple Student Records (Contd.)

Shell:

```
$ make studRec ; ./studRec <stud.dat 2>/dev/null
cc     studRec.c   -o studRec
First name: Rakesh
Surname: Kumar
Roll number: 07SI2035
Hall code: MMM
Semesters:  0
semester 0: sgpa: 9.15 cgpa: 9.15
subCode credit  gradeWt
CS1101    5        10
EC1101    5         9
CE1101    3         8
```

Simple Student Records (Contd.)

Editor:

```
void displayRegStud (studTyp s) {
  printf ("First name: %s\n", s.fname);
  printf ("Surname: %s\n", s.sname);
  printf ("Roll number: %s\n", s.roll);
  printf ("Hall code: %s\n", s.hall);
  printf ("Semesters: %d\n", s.nSem);
}
```

Simple Student Records (Contd.)

Editor:

```
void interactiveRegStud (studPtr s) {
  fprintf(stderr, "First name? ");
  scanf (" %as", &(*s).fname);
  fprintf(stderr, "Surname? ");
  scanf (" %as", &(*s).sname);
  fprintf(stderr, "Roll number? ");
  scanf (" %9s", (*s).roll);
  fprintf(stderr, "Hall code? ");
  scanf (" %9s", (*s).hall);
  s->nSem = s->semSz = 0;
}
```

Simple Student Records (Contd.)

Editor:

```
void displaySemStud (studTyp s) {
  int i, j;
  for (i=0; i<s.nSem; i++) {
    printf ("semester %d: sgpa: %.2f cgpa: %.2f\n",
        i, s.semA[i].sgpa, s.semA[i].cgpa);
    printf ("subCode\tcredit\tgradeWt\n");
    for (j=0; j<s.semA[i].nSbj; j++)
      printf("%s\t%3d\t%5d\n",
          s.semA[i].sbjA[j].subCode,
          s.semA[i].sbjA[j].credit,
          s.semA[i].sbjA[j].gradeWt);
  }
}
```

Simple Student Records (Contd.)

Editor:

```
void interactiveSemStud (studPtr s) {
  int i, n;
  subInfoPtr sA;
  if (s->semSz == 0) {
    s->semSz = 8;
    s->semA = (semInfoPtr) malloc
      (s->semSz*sizeof(semInfoTyp));
  }
  if (s->semSz > (*s).nSem) {
    s->nSem += 1;
  } else
    exit(1);
  fprintf(stderr, "Number of subjects? ");
  scanf ("%d", &n);
  sA = (subInfoPtr) malloc (n*sizeof(subInfoTyp));
  s->semA[s->nSem-1].nSbj = n;
  s->semA[s->nSem-1].sbjA = sA;
```

Simple Student Records (Contd.)

Editor:

```
  for (i=0; i<n; i++) {
    fprintf(stderr, "subCode? ");
    scanf(" %9s", sA[i].subCode);
    fprintf(stderr, "credit? ");
    scanf("%d", &(sA[i].credit));
    fprintf(stderr, "gradWt? ");
    scanf("%d", &(sA[i].gradeWt));
  }
  computeSGPA(s->semA + (s->nSem-1));
  computeLastCGPA(s->semA, s->nSem);
}
```

Simple Student Records (Contd.)

Editor:

```
void computeSGPA(semInfoPtr semP) {
  subInfoPtr sbjA=semP->sbjA;
  int nSbj = semP->nSbj;
  int i, s, ws;
  for (i=0,ws=s=0; i<nSbj; i++) {
    ws += sbjA[i].credit * sbjA[i].gradeWt;
    s += sbjA[i].credit;
  }
  if (nSbj && s) {
    semP->sgpa = ((float) ws)/s ;
    semP->creditS = s;
  } else {
    semP->sgpa = 0;
    semP->creditS = 0;
  }
}
```

Simple Student Records (Contd.)

Editor:

```
void computeLastCGPA(semInfoPtr semA, int nSem) {
  int i, s=0; float ws=0;
  for (i=0; i<(nSem-1); i++) s += semA[i].creditS;
  if (nSem > 1) ws = semA[nSem-2].cgpa * s;
  ws += semA[nSem-1].sgpa * semA[nSem-1].creditS;
  s += semA[nSem-1].creditS;
  semA[nSem-1].cgpa = (s==0 ? 0 : ws/s);
}
```

Section outline

33 **Linked lists**

- Typedef for linked lists
- Inserting in a linked list
- Deleting from a linked list

Self referential typedef for linked lists

node1

data

- `struct lNodeTag {`
  ```
  int data;
  struct lNodeTag *next;
  ```
  `} ;`
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure

Self referential typedef for linked lists

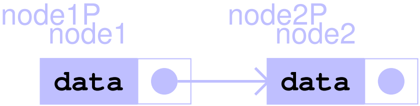
node1	node2
data	data

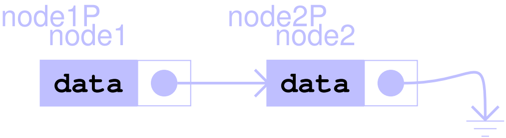
- ```
  struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } ;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Self referential typedef for linked lists


- ```
  struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } ;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Self referential typedef for linked lists

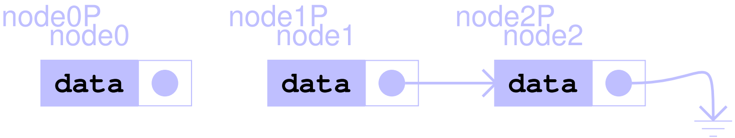

- ```
  struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } ;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Self referential typedef for linked lists


- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

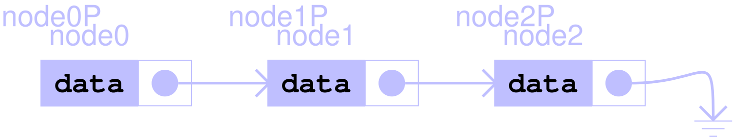
Self referential typedef for linked lists



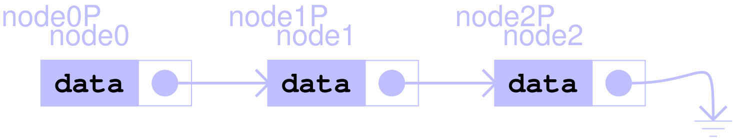

- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Self referential typedef for linked lists




- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

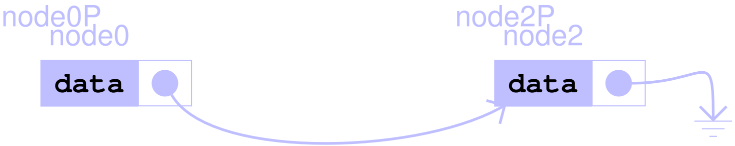
Self referential typedef for linked lists



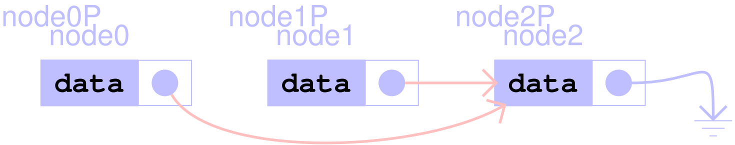

- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Self referential typedef for linked lists




- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

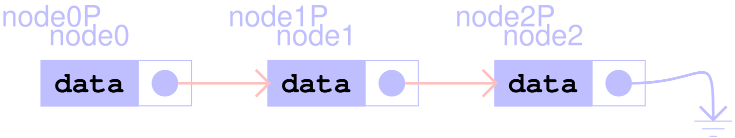
Self referential typedef for linked lists



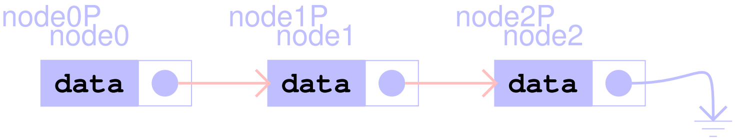

- ```
  typedef struct lNodeTag {
      int data;
      struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P->next = node2P; // assume node1 is present`
- `node2P->next = NULL;`
- `node0P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node0P->next = node1P;`
- New node was introduced at the left end of the linked structure
  ```

Inserting in the Middle (after `node0`)



-
```
typedef struct lNodeTag {
   int data;
   struct lNodeTag *next;
} lNodeTyp, *lNodePtr;
```
- `node1P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node0P->next;`
- `node0P->next = node1P;`
- New node was introduced after node0 in the linked structure
- Do not forget to assign the data fields

Inserting in the Middle (after `node0`)


- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node0P->next;`
- `node0P->next = node1P;`
- New node was introduced after node0 in the linked structure
- Do not forget to assign the data fields
  ```

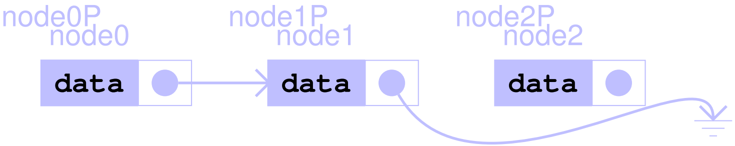
Inserting in the Middle (after `node0`)



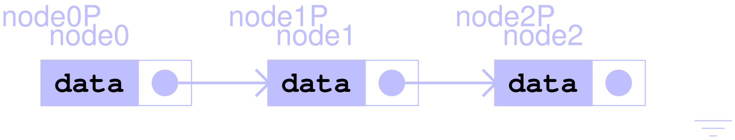

- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node0P->next;`
- `node0P->next = node1P;`
- New node was introduced after node0 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting in the Middle (after `node0`)




- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node0P->next;`
- `node0P->next = node1P;`
- New node was introduced after node0 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting in the Middle (after `node0`)



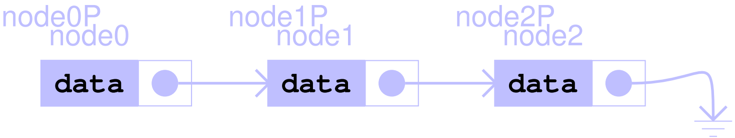

- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node1P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node0P->next;`
- `node0P->next = node1P;`
- New node was introduced after node0 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting at the end (after `node1`)

-
```
typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
} lNodeTyp, *lNodePtr;
```
- `node2P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node2P;`
- `node2P->next = NULL;`
- New node was introduced after node1 in the linked structure
- Do not forget to assign the data fields

Inserting at the end (after `node1`)




- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node2P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node2P;`
- `node2P->next = NULL;`
- New node was introduced after node1 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting at the end (after `node1`)



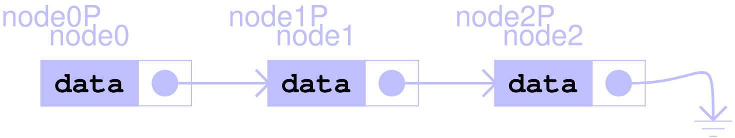

- ```c
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node2P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node2P;`
- `node2P->next = NULL;`
- New node was introduced after node1 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting at the end (after `node1`)



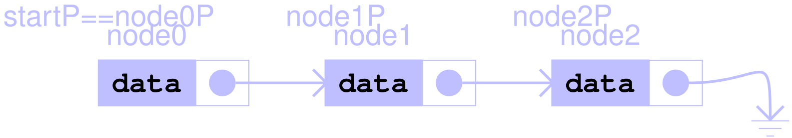

- ```
  typedef struct lNodeTag {
     int data;
     struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node2P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node2P;`
- `node2P->next = NULL;`
- New node was introduced after node1 in the linked structure
- Do not forget to assign the data fields
  ```

Inserting at the end (after `node1`)




- ```
  typedef struct lNodeTag {
    int data;
    struct lNodeTag *next;
  } lNodeTyp, *lNodePtr;
  ```
- `node2P = (lNodePtr) malloc(sizeof(lNodeTyp));`
- `node1P->next = node2P;`
- `node2P->next = NULL;`
- New node was introduced after node1 in the linked structure
- Do not forget to assign the data fields
  ```

Deleting from Start



- Initially

  `startP == node0P`

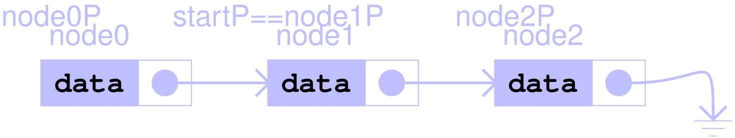
- Next

  `startP=node0P->next`

- Finally release node0

  `free(node0P)`

Deleting from Start



- Initially

  `startP == node0P`

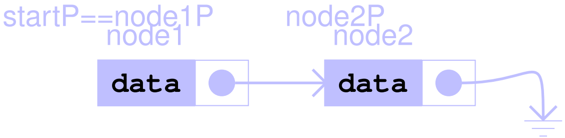
- Next

  `startP=node0P->next`

- Finally release node0

  `free(node0P)`

Deleting from Start



- Initially

  `startP == node0P`

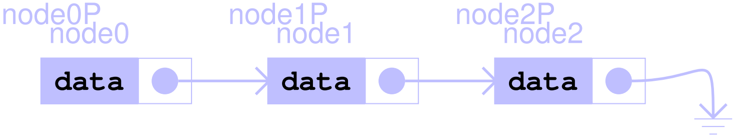
- Next

  `startP=node0P->next`

- Finally release node0

  `free(node0P)`

Deleting from Within (`node1`)



- Need to know the predecessor of the node to be deleted

```
node1P=node0P->next // identify node to be deleted
                    // and its predecessor
```

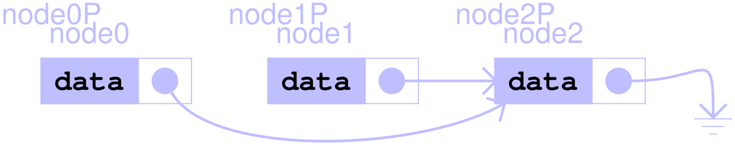
- Next, skip the node to be deleted

```
node0P->next=node1P->next
```

- Finally release node1

```
free(node1P)
```

Deleting from Within (`node1`)



- Need to know the predecessor of the node to be deleted

```
node1P=node0P->next // identify node to be deleted
                    // and its predecessor
```

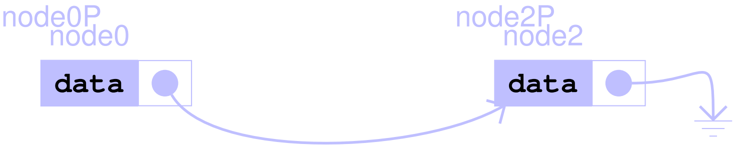
- Next, skip the node to be deleted

```
node0P->next=node1P->next
```

- Finally release node1

```
free(node1P)
```

Deleting from Within (`node1`)



- Need to know the predecessor of the node to be deleted

```
node1P=node0P->next // identify node to be deleted
                    // and its predecessor
```

- Next, skip the node to be deleted

```
node0P->next=node1P->next
```

- Finally release node1

```
free(node1P)
```

Section outline

34 **Stacks using lists**

- Function prototypes for stack
- Typedefs for stack
- Functions for the prototypes

Functions of interest for a stack

- Types for items: **`itemTyp, itemPtr`**
- Types for stack: **`stackTyp, stackPtr`**
- **`stackPtr stackNew();`**
  returns a pointer to a new stack structure
- **`int stackIsEmpty(stackPtr);`**
  returns 0 if not empty, 1 otherwise
- **`int stackIsFull(stackPtr);`**
  returns 0 if not full, 1 otherwise

Functions of interest for a stack

- Types for items: **`itemTyp, itemPtr`**
- Types for stack: **`stackTyp, stackPtr`**
- **`stackPtr stackNew();`**
  returns a pointer to a new stack structure
- **`int stackIsEmpty(stackPtr);`**
  returns 0 if not empty, 1 otherwise
- **`int stackIsFull(stackPtr);`**
  returns 0 if not full, 1 otherwise

Functions of interest for a stack

- Types for items: **`itemTyp, itemPtr`**
- Types for stack: **`stackTyp, stackPtr`**
- **`stackPtr stackNew();`**
  returns a pointer to a new stack structure
- **`int stackIsEmpty(stackPtr);`**
  returns 0 if not empty, 1 otherwise
- **`int stackIsFull(stackPtr);`**
  returns 0 if not full, 1 otherwise

Functions of interest for a stack

- Types for items: `itemTyp, itemPtr`
- Types for stack: `stackTyp, stackPtr`
- `stackPtr stackNew();`
  returns a pointer to a new stack structure
- `int stackIsEmpty(stackPtr);`
  returns 0 if not empty, 1 otherwise
- `int stackIsFull(stackPtr);`
  returns 0 if not full, 1 otherwise

Functions of interest for a stack (contd.)

- `int stackPush(stackPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int stackPop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, popped item returned via second argument
- `int stackTop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, top item returned via second argument
- `void stackDestroy(stackPtr);`

Functions of interest for a stack (contd.)

- `int stackPush(stackPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int stackPop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, popped item returned via second argument
- `int stackTop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, top item returned via second argument
- `void stackDestroy(stackPtr);`

Functions of interest for a stack (contd.)

- `int stackPush(stackPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int stackPop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, popped item returned via second argument
- `int stackTop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, top item returned via second argument
- `void stackDestroy(stackPtr);`

Functions of interest for a stack (contd.)

- `int stackPush(stackPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int stackPop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, popped item returned via second argument
- `int stackTop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, top item returned via second argument
- `void stackDestroy(stackPtr);`

Functions of interest for a stack (contd.)

- `int stackPush(stackPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int stackPop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, popped item returned via second argument
- `int stackTop(stackPtr, itemPtr);`
  returns 0 for failure, 1 for success, top item returned via second argument
- `void stackDestroy(stackPtr);`

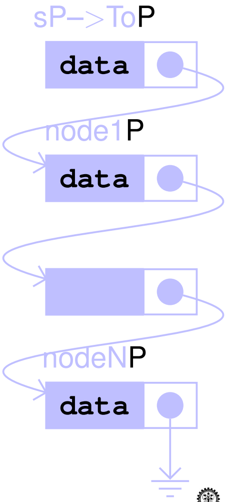
Linked List based typedefs for stack

Editor:

```
// Types for items: itemTyp, itemPtr
typedef int itemTyp, *itemPtr;

typedef struct lNodeTag {
itemTyp data;
struct lNodeTag *next;
} lNodeTyp, *lNodePtr;

// Types for stack: stackTyp, stackPtr
typedef struct stackTag {
  lNodePtr toP;
} stackTyp, *stackPtr;
```

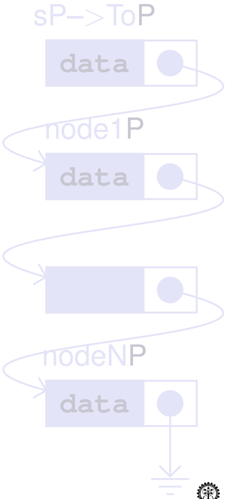


Linked List based Stack API Functions

Editor:

```
stackPtr stackNew() { // returns:
// pointer to a new stack structure
  stackPtr sP;
  sP = (stackPtr) malloc
    (sizeof(stackTyp));
  sP->toP=NULL; // empty stack
  return sP;
}

int stackIsEmpty(stackPtr sP) {
// returns 0 if not empty, 1 otherwise
  return (sP->toP==NULL);
}
```

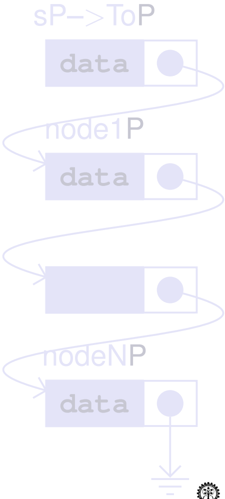


Linked List based Stack API Functions

Editor:

```
stackPtr stackNew() { // returns:
// pointer to a new stack structure
  stackPtr sP;
  sP = (stackPtr) malloc
    (sizeof(stackTyp));
  sP->toP=NULL; // empty stack
  return sP;
}

int stackIsEmpty(stackPtr sP) {
// returns 0 if not empty, 1 otherwise
  return (sP->toP==NULL);
}
```

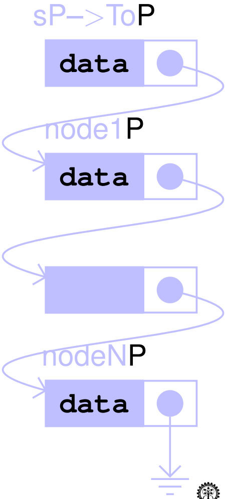


Linked List based Stack API Functions

Editor:

```
stackPtr stackNew() { // returns:
// pointer to a new stack structure
  stackPtr sP;
  sP = (stackPtr) malloc
    (sizeof(stackTyp));
  sP->toP=NULL; // empty stack
  return sP;
}

int stackIsEmpty(stackPtr sP) {
// returns 0 if not empty, 1 otherwise
  return (sP->toP==NULL);
}
```

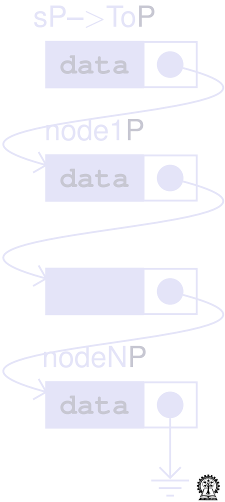


Linked List based Stack API Functions (Contd.)

Editor:

```
int stackIsFull(stackPtr sP) {
// returns 0 if not full, 1 otherwise
  return 0; // never full
}

int stackPush(stackPtr sP, itemTyp d) {
// returns 0 for failure, 1 for success
  lNodePtr  sNdP;
  sNdP = (lNodePtr) malloc
    (sizeof(lNodeTyp));
  // allocate a new node for the new data
  sNdP->data = d; // copy data to new node
  sNdP->next = sP->toP;
  // the older top will go below new node
  sP->toP= sNdP; // make new node the top
  return 1; // always successful
}
```

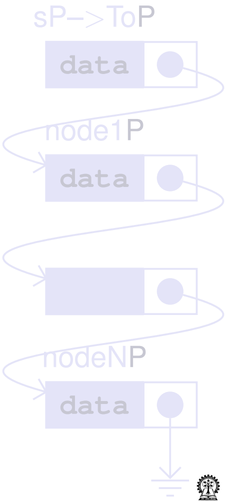


Linked List based Stack API Functions (Contd.)

Editor:

```
int stackIsFull(stackPtr sP) {
// returns 0 if not full, 1 otherwise
  return 0; // never full
}

int stackPush(stackPtr sP, itemTyp d) {
// returns 0 for failure, 1 for success
  lNodePtr  sNdP;
  sNdP = (lNodePtr) malloc
    (sizeof(lNodeTyp));
  // allocate a new node for the new data
  sNdP->data = d; // copy data to new node
  sNdP->next = sP->toP;
  // the older top will go below new node
  sP->toP= sNdP; // make new node the top
  return 1; // always successful
}
```

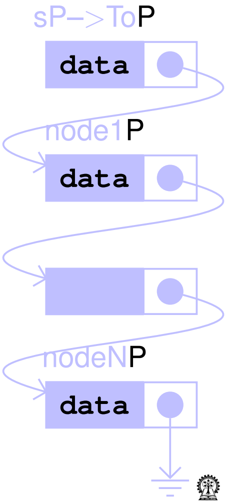


Linked List based Stack API Functions (Contd.)

Editor:

```
int stackIsFull(stackPtr sP) {
// returns 0 if not full, 1 otherwise
  return 0; // never full
}

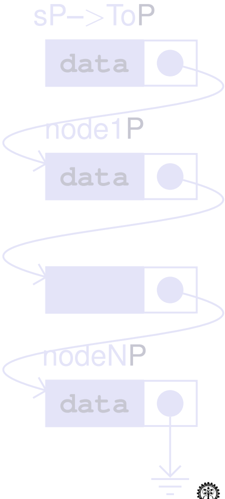
int stackPush(stackPtr sP, itemTyp d) {
// returns 0 for failure, 1 for success
  lNodePtr  sNdP;
  sNdP = (lNodePtr) malloc
    (sizeof(lNodeTyp));
  // allocate a new node for the new data
  sNdP->data = d; // copy data to new node
  sNdP->next = sP->toP;
  // the older top will go below new node
  sP->toP= sNdP; // make new node the top
  return 1; // always successful
}
```



Linked List based Stack API Functions (Contd.)

Editor:

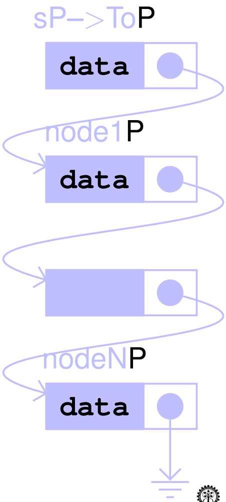
```
int stackPop(stackPtr sP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// popped item returned via dP
  lNodePtr oldToP;
  if (stackIsEmpty(sP)) return 0;
  *dP = sP->toP->data;
    // data copied to dP location
  oldToP = sP->toP; // for freeing later
  sP->toP = sP->toP->next;
    // top moves down
  free(oldToP); // older top is freed
  return 1;
}
```



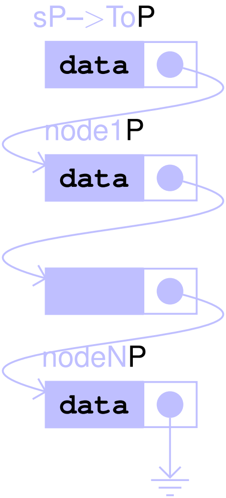
Linked List based Stack API Functions (Contd.)

Editor:

```
int stackPop(stackPtr sP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// popped item returned via dP
  lNodePtr oldToP;
  if (stackIsEmpty(sP)) return 0;
  *dP = sP->toP->data;
    // data copied to dP location
  oldToP = sP->toP; // for freeing later
  sP->toP = sP->toP->next;
    // top moves down
  free(oldToP); // older top is freed
  return 1;
}
```



Linked List based Stack API Functions (Contd.)



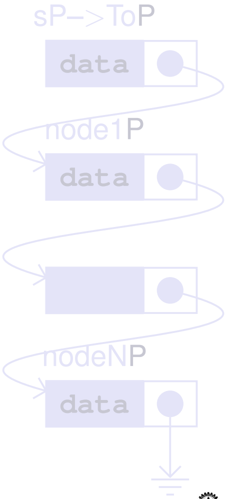
Editor:

```
int stackTop(stackPtr sP, itemPtr dP) {
// returns 0 for failure, 1 for success
// top item returned via
// second argument
  if (stackIsEmpty(sP)) return 0;
  *dP = sP->toP->data;
  return 1;
}
```

Linked List based Stack API Functions (Contd.)

Editor:

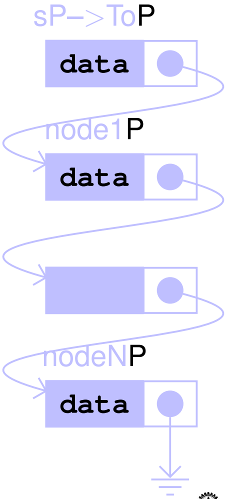
```
void stackDestroy(stackPtr sP) {
// free all memory taken up this stack
  lNodePtr nextP, thisP=sP->toP;
  while (thisP) {
    nextP = thisP->next;
    free (thisP);
    thisP=nextP;
  }
  free(sP);
}
```



Linked List based Stack API Functions (Contd.)

Editor:

```
void stackDestroy(stackPtr sP) {
// free all memory taken up this stack
  lNodePtr nextP, thisP=sP->toP;
  while (thisP) {
    nextP = thisP->next;
    free (thisP);
    thisP=nextP;
  }
  free(sP);
}
```



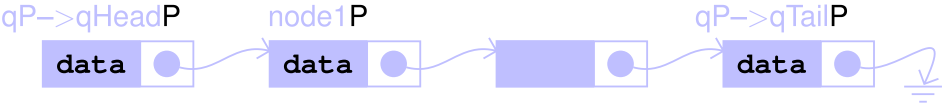
‘

Section outline

35 **Queues using lists**

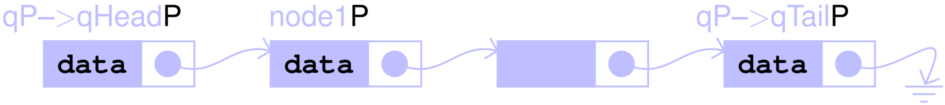
- Function prototypes for queues
- Typedefs for queues
- Functions for the prototypes

Functions of interest for a queue



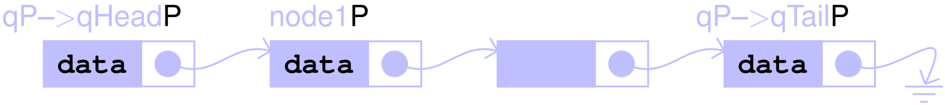
- Types for items: `itemTyp, itemPtr`
- Types for queue: `QTyp, QPtr`
- `QPtr QNew();`
  returns a pointer to a new Q structure
- `int QIsEmpty(QPtr);`
  returns 0 if not empty, 1 otherwise
- `int QIsFull(QPtr);`
  returns 0 if not full, 1 otherwise

Functions of interest for a queue



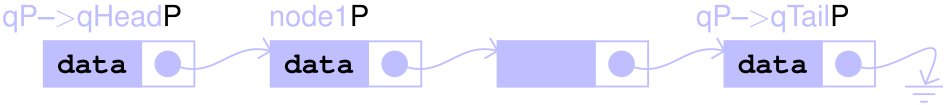
- Types for items: **`itemTyp, itemPtr`**
- Types for queue: **`QTyp, QPtr`**
- **`QPtr QNew();`**
  returns a pointer to a new Q structure
- **`int QIsEmpty(QPtr);`**
  returns 0 if not empty, 1 otherwise
- **`int QIsFull(QPtr);`**
  returns 0 if not full, 1 otherwise

Functions of interest for a queue



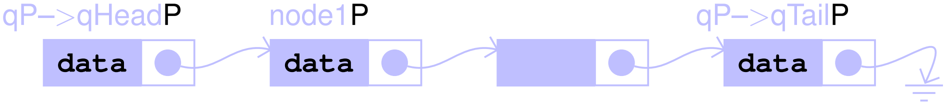
- Types for items: `itemTyp, itemPtr`
- Types for queue: `QTyp, QPtr`
- `QPtr QNew();`
  returns a pointer to a new Q structure
- `int QIsEmpty(QPtr);`
  returns 0 if not empty, 1 otherwise
- `int QIsFull(QPtr);`
  returns 0 if not full, 1 otherwise

Functions of interest for a queue



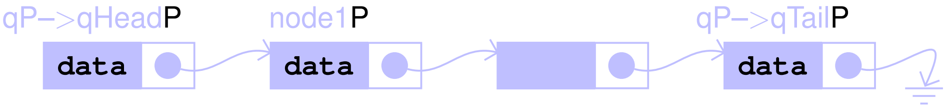
- Types for items: **`itemTyp, itemPtr`**
- Types for queue: **`QTyp, QPtr`**
- **`QPtr QNew();`**
  returns a pointer to a new Q structure
- **`int QIsEmpty(QPtr);`**
  returns 0 if not empty, 1 otherwise
- **`int QIsFull(QPtr);`**
  returns 0 if not full, 1 otherwise

Functions of interest for a queue (contd.)



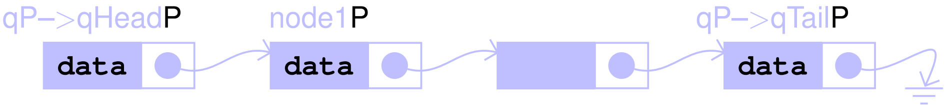
- `int QEnque(QPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int QDeque(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, dequeued item returned via second argument
- `int QFront(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, front item returned via second argument
- `void QDestroy(QPtr);`

Functions of interest for a queue (contd.)



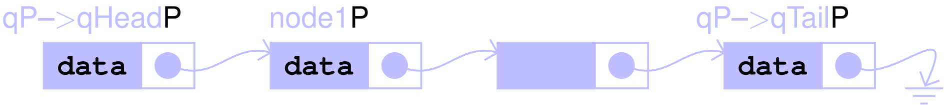
- `int QEnque(QPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int QDeque(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, dequeued item returned via second argument
- `int QFront(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, front item returned via second argument
- `void QDestroy(QPtr);`

Functions of interest for a queue (contd.)



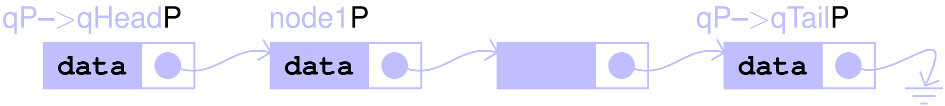
- `int QEnque(QPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int QDeque(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, dequeued item returned via second argument
- `int QFront(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, front item returned via second argument
- `void QDestroy(QPtr);`

Functions of interest for a queue (contd.)



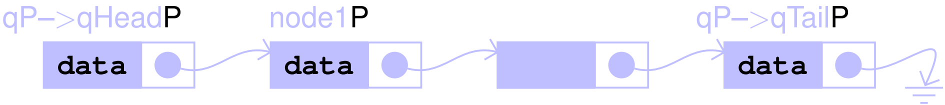
- `int QEnque(QPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int QDeque(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, dequeued item returned via second argument
- `int QFront(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, front item returned via second argument
- `void QDestroy(QPtr);`

Functions of interest for a queue (contd.)



- `int QEnque(QPtr, itemTyp);`
  returns 0 for failure, 1 for success
- `int QDeque(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, dequeued item returned via second argument
- `int QFront(QPtr, itemPtr);`
  returns 0 for failure, 1 for success, front item returned via second argument
- `void QDestroy(QPtr);`

Linked List based Typedefs for Queue

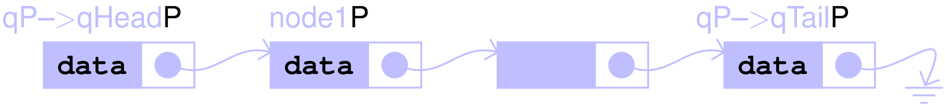


- Reuse **`itemTyp`** and **`lNodeTyp`** from Stack
- Principal differneces with stack? – FIFO rather than LIFO
- Do we need to work with the linked list differently?
- Easy to insert at "grounded" end, but hard to remove from there
- At other end both insert and delete are easy – so dequeue here and enqueue at "grounded" end

Editor:

```
// Types for queue: QTyp, QPtr
typedef struct QTag {
  lNodePtr headP, tailP;
} QTyp, *QPtr;
```

Linked List based Typedefs for Queue

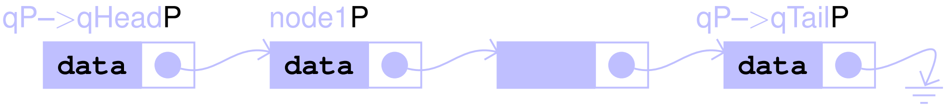


- Reuse **`itemTyp`** and **`lNodeTyp`** from Stack
- Principal differneces with stack? – FIFO rather than LIFO
- Do we need to work with the linked list differently?
- Easy to insert at "grounded" end, but hard to remove from there
- At other end both insert and delete are easy – so dequeue here and enqueue at "grounded" end

Editor:

```
// Types for queue: QTyp, QPtr
typedef struct QTag {
  lNodePtr headP, tailP;
} QTyp, *QPtr;
```

Linked List based Typedefs for Queue

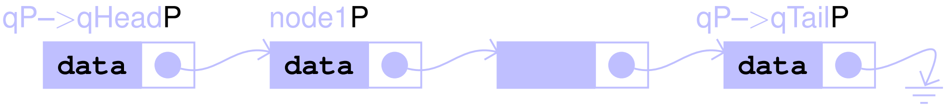


- Reuse **`itemTyp`** and **`lNodeTyp`** from Stack
- Principal differneces with stack? – FIFO rather than LIFO
- Do we need to work with the linked list differently?
- Easy to insert at "grounded" end, but hard to remove from there
- At other end both insert and delete are easy – so dequeue here and enqueue at "grounded" end

Editor:

```
// Types for queue: QTyp, QPtr
typedef struct QTag {
  lNodePtr headP, tailP;
} QTyp, *QPtr;
```

Linked List based Typedefs for Queue

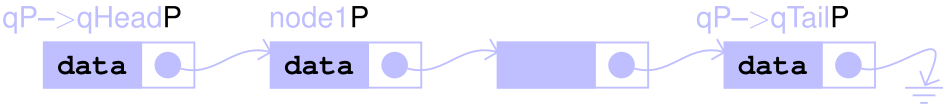


- Reuse **`itemTyp`** and **`lNodeTyp`** from Stack
- Principal differneces with stack? – FIFO rather than LIFO
- Do we need to work with the linked list differently?
- Easy to insert at “grounded” end, but hard to remove from there
- At other end both insert and delete are easy – so dequeue here and enqueue at “grounded” end

Editor:

```
// Types for queue: QTyp, QPtr
typedef struct QTag {
  lNodePtr headP, tailP;
} QTyp, *QPtr;
```

Linked List based Typedefs for Queue

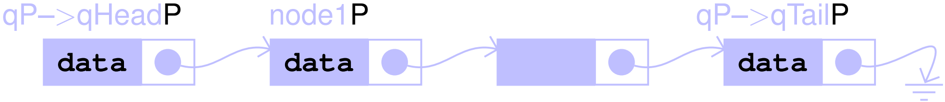


- Reuse **itemTyp** and **lNodeTyp** from Stack
- Principal differneces with stack? – FIFO rather than LIFO
- Do we need to work with the linked list differently?
- Easy to insert at “grounded” end, but hard to remove from there
- At other end both insert and delete are easy – so dequeue here and enqueue at “grounded” end

Editor:

```
// Types for queue: QTyp, QPtr
typedef struct QTag {
  lNodePtr headP, tailP;
} QTyp, *QPtr;
```

Linked List based Queue API Functions



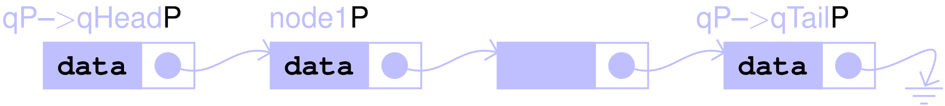
Editor:

```
QPtr QNew() { // returns a pointer to a new queue struct
  QPtr qP = (QPtr) malloc (sizeof(QTyp));
  qP->headP=qP->tailP=NULL;
  return qP;
}

int QIsEmpty(QPtr qP) { // ret: 1 if empty, 0 otherwise
  return (qP->headP==NULL);
}

int QIsFull(QPtr qP) { // ret: 1 if full, 0 otherwise
  return 0; // never full
}
```

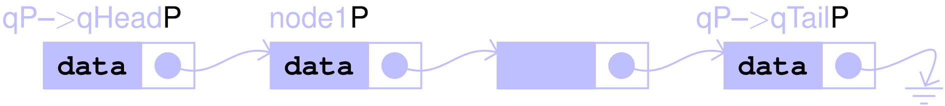
Linked List based Queue API Functions (Contd.)



Editor:

```
int QEnque(QPtr qP, itemTyp d) { // new data goes to tail
// return: 0 for failure, 1 for success
  lNodePtr  qNdP = (lNodePtr) malloc (sizeof(lNodeTyp));
  qNdP->data = d; // copy data to new node
  qNdP->next = NULL; // as this will be the new end
  if (qP->tailP) // if Q is not empty
    qP->tailP->next= qNdP; // append after current tail
  else // Q empty -- no nodes in the list
    qP->headP=qNdP; // so, new node becomes a fresh head
  qP->tailP = qNdP; // new node is the new tail, always
  return 1; // always successful
}
```

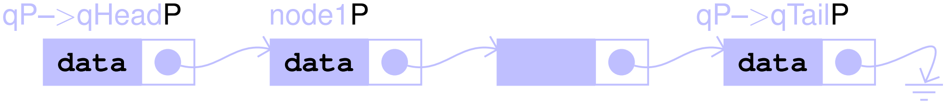
Linked List based Queue API Functions (Contd.)



Editor:

```
int QEnque(QPtr qP, itemTyp d) { // new data goes to tail
// return: 0 for failure, 1 for success
  lNodePtr  qNdP = (lNodePtr) malloc (sizeof(lNodeTyp));
  qNdP->data = d; // copy data to new node
  qNdP->next = NULL; // as this will be the new end
  if (qP->tailP) // if Q is not empty
    qP->tailP->next= qNdP; // append after current tail
  else // Q empty -- no nodes in the list
    qP->headP=qNdP; // so, new node becomes a fresh head
  qP->tailP = qNdP; // new node is the new tail, always
  return 1; // always successful
}
```

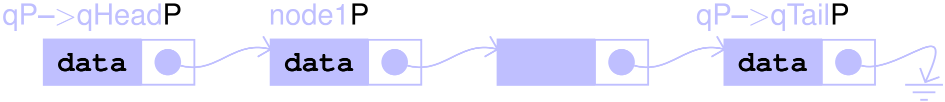
Linked List based Queue API Functions (Contd.)



Editor:

```
int QDeque(QPtr qP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// dequeued item returned via second argument
// needs to be removed from the head of the list
  lNodePtr oldHeadP = qP->headP;
  if (QIsEmpty(qP)) return 0; // return 0 for empty Q
  *dP = oldHeadP->data; // copy data from head node to dP
  qP->headP = oldHeadP->next; // that's the new head
  if (qP->headP == NULL) qP->tailP=NULL;
  // set qP->tailP to NULL if list should become empty
  free(oldHeadP); // release memory taken up old
  return 1;
}
```

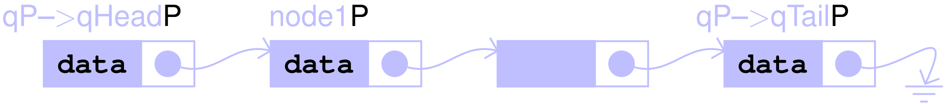
Linked List based Queue API Functions (Contd.)



Editor:

```
int QDeque(QPtr qP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// dequeued item returned via second argument
// needs to be removed from the head of the list
  lNodePtr oldHeadP = qP->headP;
  if (QIsEmpty(qP)) return 0; // return 0 for empty Q
  *dP = oldHeadP->data; // copy data from head node to dP
  qP->headP = oldHeadP->next; // that's the new head
  if (qP->headP == NULL) qP->tailP=NULL;
  // set qP->tailP to NULL if list should become empty
  free(oldHeadP); // release memory taken up old
  return 1;
}
```

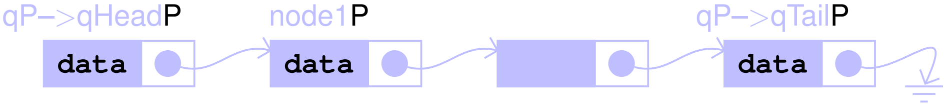
Linked List based Queue API Functions (Contd.)



Editor:

```
int QFront(QPtr qP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// front item returned via second argument
// needs to be taken from the head of the list
  if (QIsEmpty(qP)) return 0;
  *dP = qP->headP->data;
  return 1;
}
```

Linked List based Queue API Functions (Contd.)



Editor:

```
void QDestroy(QPtr qP) {
// free all memory taken up this Q
  lNodePtr nextP, thisP=qP->headP;
  while (thisP) {
    nextP = thisP->next;
    free (thisP);
    thisP=nextP;
  }
  free(qP);
}
```

Section outline

36 Array based implementations

- Stacks using arrays
- Queues using arrays

Array based Stack Typedef

Editor:

```
// Types for items: itemTyp, itemPtr
typedef int itemTyp, *itemPtr;

// Types for stack: stackTyp, stackPtr
#define STKSIZE 15
typedef struct stackTag {
  int topI; // current position of top element
  int sz;
  itemTyp *iArr;
} stackTyp, *stackPtr;
```

Array based Stack API Functions

Editor:

```
stackPtr stackNew() {
// returns a pointer to a new stack structure
  stackPtr sP;
  sP = (stackPtr) malloc (sizeof(stackTyp));
  sP->sz=STKSIZE;
  sP->iArr = (itemPtr) malloc (sP->sz*sizeof(itemTyp));
  sP->topI=-1; // empty stack
  return sP;
}
```

Array based Stack API Functions (Contd.)

Editor:

```
int stackIsEmpty(stackPtr sP) {
// returns 0 if not empty, 1 otherwise
  return (sP->topI<0);
}

int stackIsFull(stackPtr sP) {
// returns 0 if not full, 1 otherwise
  return (sP->topI>=sP->sz-1) ;
}

int stackPush(stackPtr sP, itemTyp d) {
// returns 0 for failure, 1 for success
  if (stackIsFull(sP)) return 0;
  sP->topI++;
  sP->iArr[sP->topI]=d;
  return 1;
}
```

Array based Stack API Functions (Contd.)

Editor:

```
int stackPop(stackPtr sP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// popped item returned via second argument
  if (stackIsEmpty(sP)) return 0;
  *dP = sP->iArr[sP->topI];
  sP->topI-=1;
  return 1;
}

int stackTop(stackPtr sP, itemPtr dP) {
// returns 0 for failure, 1 for success, top item
returned
// via second argument
  if (stackIsEmpty(sP)) return 0;
  *dP = sP->iArr[sP->topI];
  return 1;
}
```

Array based Stack API Functions (Contd.)

Editor:

```
void stackDestroy(stackPtr sP) {
// free all memory taken up this stack
  free(sP->iArr);
  free(sP);
}
```

Array based Queue Typedef

Editor:

```
// Types for items: itemTyp, itemPtr
typedef int itemTyp, *itemPtr;

// Types for queue: QTyp, QPtr
#define STKSIZE 15
typedef struct QTag {
  int front, rear, sz;
  itemTyp iArr[STKSIZE];
#if defined (Q_EFLAG) // Q Empty using flag
  int  emptyFlag;
#elif defined (Q_COUNT) // Q Empty/Full using counter
  int iCount;
#endif
} QTyp, *QPtr;
```

Array based Queue API Functions

Editor:

```
QPtr QNew() {
// returns a pointer to a new queue structure
  QPtr qP;
  qP->front=qP->rear=0;
#if defined (Q_EFLAG) // Q Empty using flag
  qP->emptyFlag=1;
#elif defined (Q_COUNT) // Q Empty/Full using counter
  qP->iCount=0;
#endif
  return qP;
}
```

Array based Queue API Functions (Contd.)

Editor:

```
int QIsEmpty(QPtr qP) {
// returns 0 if not empty, 1 otherwise
#if defined (Q_EFLAG) // Q Empty using flag
  return (qP->emptyFlag);
#elif defined (Q_COUNT) // Q Empty/Full using counter
  return (qP->iCount==0);
#else
  return (qP->rear == qP->front) ;
#endif
}
```

Array based Queue API Functions (Contd.)

Editor:

```
int QIsFull(QPtr qP) {
// returns 0 if not full, 1 otherwise
#if defined (Q_EFLAG) // Q Empty using flag
  if (qP->emptyFlag) return 0;
  else return (qP->front==qP->rear) ;
#elif defined (Q_COUNT) // Q Empty/Full using counter
  return (qP->iCount==qP->sz);
#else
  return ((qP->rear+1) % qP->sz == qP->front) ;
#endif
}
```

Array based Queue API Functions (Contd.)

Editor:

```
int QEnque(QPtr qP, itemTyp d) {
// returns 0 for failure, 1 for success
// needs to go at the end of the list
  if (QIsFull(qP)) return 0;
  qP->iArr[qP->rear]=d;
  qP->rear = (qP->rear+1) % qP->sz;
#if defined (Q_EFLAG) // Q Empty using flag
  qP->emptyFlag=0;
#elif defined (Q_COUNT) // Q Empty/Full using counter
  qP->iCount++;
#endif
  return 1;
}
```

Array based Queue API Functions (Contd.)

Editor:

```
int QDeque(QPtr qP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// dequeued item returned via second argument
// needs to be removed from the head of the list
  if (QIsEmpty(qP)) return 0;
  *dP = qP->iArr[qP->front];
  qP->front = (qP->front+1) % qP->sz;
#if defined (Q_EFLAG) // Q Empty using flag
  if (qP->front==qP->rear) qP->emptyFlag=1;
#elif defined (Q_COUNT) // Q Empty/Full using counter
  qP->iCount--;
#endif
  return 1;
}
```

Array based Queue API Functions (Contd.)

Editor:

```
int QFront(QPtr qP, itemPtr dP) {
// returns 0 for failure, 1 for success,
// front item returned via second argument
// needs to be taken from the head of the list
  if (QIsEmpty(qP)) return 0;
  *dP = qP->iArr[qP->front];
  return 1;
}

void QDestroy(QPtr qP) {
// free all memory taken up this Q
  free(qP->iArr);
  free(qP);
}
```

Section outline

37 **Applications**

- Evaluation of Postfix Expressions
- Postfix to Infix

Evaluation of Postfix Expressions

Editor:

```
#include <stdio.h>

typedef float itemTyp, *itemPtr;
#include "stack-ll.c"

void stackEmptyErr(void);
void addTop2(stackPtr sP, int iFlag);
void subTop2(stackPtr sP, int iFlag);
void mulTop2(stackPtr sP, int iFlag);
void divTop2(stackPtr sP, int iFlag);

void defaultAction(int iFlag){
  if (iFlag) printf("default: skipping\n");
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
interpretPostfix(stackPtr sP, int iFlag){
  float fNum; char ch;
  scanf(" %c", &ch);
  while (!feof(stdin)) {
    switch (ch) {
      case '+': addTop2(sP, iFlag); break;
      case '-': subTop2(sP, iFlag); break;
      case '*': mulTop2(sP, iFlag); break;
      case '/': divTop2(sP, iFlag); break;
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
      default :
      if ((ch>='0' && ch<='9') || (ch=='.')) {
        ungetc(ch, stdin);
        if (scanf("%f", &fNum)) {
          stackPush(sP, fNum);
          if (iFlag)
            printf("pushed %f\n", fNum);
        }
      } else
      defaultAction(iFlag);
      break;
    }
    scanf(" %c", &ch);
  }
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
void stackEmptyErr() {
  fprintf(stderr, "stack empty while popping,
exiting\n");
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
void addTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  if (!stackPop(sP, &fn2)) stackEmptyErr();
  if (!stackPop(sP, &fn1)) stackEmptyErr();
  stackPush(sP, fn1+fn2);
  if (iFlag) {
    printf("popped %f and %f, pushed sum=%f\n",
        fn2, fn1, fn1+fn2);
  }
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
void subTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  if (!stackPop(sP, &fn2)) stackEmptyErr();
  if (!stackPop(sP, &fn1)) stackEmptyErr();
  stackPush(sP, fn1-fn2);
  if (iFlag) {
    printf("popped %f and %f, pushed diff=%f\n",
            fn2, fn1, fn1-fn2);
  }
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
void mulTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  if (!stackPop(sP, &fn2)) stackEmptyErr();
  if (!stackPop(sP, &fn1)) stackEmptyErr();
  stackPush(sP, fn1*fn2);
  if (iFlag) {
    printf("popped %f and %f, pushed product=%f\n",
        fn2, fn1, fn1*fn2);
  }
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
void divTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  if (!stackPop(sP, &fn2)) stackEmptyErr();
  if (!stackPop(sP, &fn1)) stackEmptyErr();
  stackPush(sP, fn1/fn2);
  if (iFlag) {
    printf("popped %f and %f, pushed div result=%f\n",
        fn2, fn1, fn1/fn2);
  }
}
```

Evaluation of Postfix Expressions (Contd.)

Editor:

```
main(){
  stackPtr sP=stackNew();
  interpretPostfix(sP, 1);
}
```

Evaluation of Postfix Expressions (Contd.)

Shell:

```
$ cc postfix.c -o postfix
$ ./postfix
3 4 + 5 *
pushed 3.000000
pushed 4.000000
popped 4.000000 and 3.000000, pushed sum=7.000000
pushed 5.000000
popped 5.000000 and 7.000000, pushed product=35.000000
```

Postfix to Infix

Editor:

```
#include <stdio.h>
#include <string.h>

typedef struct {
  float fNum;
  char *expStr;
}  itemTyp, *itemPtr;
#include "stack-ll.c"

void stackEmptyErr(void);
void addTop2(stackPtr sP, int iFlag);
void subTop2(stackPtr sP, int iFlag);
void mulTop2(stackPtr sP, int iFlag);
void divTop2(stackPtr sP, int iFlag);
```

Postfix to Infix (Contd.)

Editor:

```
void defaultAction(int iFlag){
  if (iFlag) printf("default: skipping\n");
}

void fNumPush(stackPtr sP, float fNum) {
  itemTyp valExp;
  valExp.fNum=fNum;
  valExp.expStr=(char*)malloc(20*sizeof(char));
  sprintf(valExp.expStr, "%f", fNum);
  stackPush(sP, valExp);
}
```

Postfix to Infix (Contd.)

Editor:

```
void valExpPush(stackPtr sP, float fNum,
    char *expStrP1, char *expStrP2, const char *oprStrP,

    int iFlag) {
  int len = strlen(expStrP1) + strlen(expStrP2) +
        strlen(oprStrP) + 7;
  itemTyp valExp;
  valExp.fNum=fNum;
  valExp.expStr=(char*)malloc(len*sizeof(char));
  sprintf(valExp.expStr,"(%s %s %s)",
        expStrP1, oprStrP, expStrP2);
  stackPush(sP, valExp);
  free(expStrP1);
  free(expStrP2);
  if (iFlag) printf("new expr: %s\n", valExp.expStr);
}
```

Postfix to Infix (Contd.)

Editor:

```
int valExpPop(stackPtr sP, float *fn, char *expStrP[]) {
  itemTyp valExp;
  if (!stackPop(sP, &valExp)) stackEmptyErr();
  *fn = valExp.fNum;
  *expStrP = valExp.expStr;
  return 1;
}
```

Postfix to Infix (Contd.)

Editor:

```
interpretPostfix(stackPtr sP, int iFlag){
  float fNum; char ch;
  scanf(" %c", &ch);
  while (!feof(stdin)) {
    switch (ch) {
      case '+': addTop2(sP, iFlag); break;
      case '-': subTop2(sP, iFlag); break;
      case '*': mulTop2(sP, iFlag); break;
      case '/': divTop2(sP, iFlag); break;
      default :
```

Postfix to Infix (Contd.)

Editor:

```
      if ((ch>='0' && ch<='9') || (ch=='.')) {
        ungetc(ch, stdin);
        if (scanf("%f", &fNum)) {
          fNumPush(sP, fNum);
          if (iFlag)
            printf("pushed %f\n", fNum);
        }
      } else
      defaultAction(iFlag);
      break;
    }
    scanf(" %c", &ch);
  }
}
```

Postfix to Infix (Contd.)

Editor:

```
void stackEmptyErr() {
  fprintf(stderr, "stack empty while popping,
exiting\n");
}

void addTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  char *expStrP1, *expStrP2;
  valExpPop(sP, &fn2, &expStrP2);
  valExpPop(sP, &fn1, &expStrP1);
  valExpPush(sP, fn1+fn2, expStrP1, expStrP2, "+",
iFlag);
  if (iFlag) {
    printf("popped %f and %f, pushed sum=%f\n",
        fn2, fn1, fn1+fn2);
  }
}
```

Postfix to Infix (Contd.)

Editor:

```
void subTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  char *expStrP1, *expStrP2;
  valExpPop(sP, &fn2, &expStrP2);
  valExpPop(sP, &fn1, &expStrP1);
  valExpPush(sP, fn1-fn2, expStrP1, expStrP2, "-",
iFlag);
  if (iFlag) {
    printf("popped %f and %f, pushed diff=%f\n",
        fn2, fn1, fn1-fn2);
  }
}
```

Postfix to Infix (Contd.)

Editor:

```
void mulTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  char *expStrP1, *expStrP2;
  valExpPop(sP, &fn2, &expStrP2);
  valExpPop(sP, &fn1, &expStrP1);
  valExpPush(sP, fn1*fn2, expStrP1, expStrP2, "*",
iFlag);
  if (iFlag) {
    printf("popped %f and %f, pushed product=%f\n",
        fn2, fn1, fn1*fn2);
  }
}
```

Postfix to Infix (Contd.)

Editor:

```
void divTop2(stackPtr sP, int iFlag) {
  float fn1, fn2;
  char *expStrP1, *expStrP2;
  valExpPop(sP, &fn2, &expStrP2);
  valExpPop(sP, &fn1, &expStrP1);
  valExpPush(sP, fn1/fn2, expStrP1, expStrP2, "/",
iFlag);
  if (iFlag) {
    printf("popped %f and %f, pushed div result=%f\n",
         fn2, fn1, fn1/fn2);
  }
}
```

Postfix to Infix (Contd.)

Editor:

```
main(){
  stackPtr sP=stackNew();
  interpretPostfix(sP, 1);
}
```

Postfix to Infix (Contd.)

Shell:

```
$ cc -o post2infix post2infix.c
$ ./post2infix
3 4 + 5 *
pushed 3.000000
pushed 4.000000
new expr: (3.000000 + 4.000000)
popped 4.000000 and 3.000000, pushed sum=7.000000
pushed 5.000000
new expr: ((3.000000 + 4.000000) * 5.000000)
popped 5.000000 and 7.000000, pushed product=35.000000
```

Part XIII

File handling

Section outline

38 **File Input/Output**

- Streams
- Opening and Closing Files

Streams and the `FILE` Structure

- In C, **`stdin`** is the standard input file stream and refers to the keyboard, by default
- **`fscanf`** and **`fprintf`** may be used for reading from and writing to specified streams, including **`stdin`** and **`stdout`**, as appropriate
- **`scanf`** is the equivalent of **`fscanf`**, with the stream set to **`stdin`**, internally
- **`printf`** is the equivalent of **`fprintf`**, with the stream set to **`stdout`**, internally
- Necessary declarations are given in **`stdio.h`**, in particular there is a defined structure called **`FILE`**
- For file input and output, we usually create variables of type **`FILE *`** to point to a file located on the computer
- These are compatible with *streams* and we could pass a **`FILE`** pointer into an input or output function, for example, **`fscanf`**

Opening and Closing Files

- We have to first open a file to be able to do anything else with it.
- Done using **`fopen`**, which takes two arguments
- The first one is the path to your file (as a string), including the filename – either absolute or relative
- The second argument is another **`char *`** (string), and determines how the file is opened by your program.
- There are 12 different values that could be used – to be see later
- Finally, **`fopen`** returns a **`FILE`** pointer if the file was opened successfully, otherwise it returns **`NULL`**
- Closing files is easy, using **`fclose`**, with a **`FILE`** pointer to an open file

Sample Program to Open a File for Reading

Editor:

```
#include <stdio.h>

int main() {
  FILE *fileP; // declare a FILE pointer

  fileP = fopen("data.txt", "r");
  // open a text file for reading

  if(fileP==NULL) {
    printf("Error: failed to open file.\n");
    return 1;
  }
  else {
    printf("File successfully opened\n");
    fscanf(fileP, "%d", &data);
    // read an integer from the file
    fclose(fileP);
```

Sample Program to Open a File for Writing

Editor:

```
#include <stdio.h>

int main() {
  FILE *fileP; // declare a FILE pointer

  file = fopen("data/writing.txt", "w");
  // create a text file for writing

  if(fileP==NULL) {
    printf("Error: can't create file.\n");
    return 1;
  }
  else {
    printf("File created\n");
    // write an integer to the file
    fprintf(fileP, "%d\n", 10);
    fclose(fileP);
```

Other Options When Opening Files

The following four options are important:

- **`"a"`** lets you open a text file for appending - i.e. add data to the end of the current text.
- **`"r+"`** will open a text file to read from or write to.
- **`"w+"`** will create a text file to read from or write to.
- **`"a+"`** will either create or open a text file for appending.
- Add a **`"b"`** to the end if you want to use binary files instead of text files, as follows:
  `"rb"`, `"wb"`, `"ab"`, `"r+b"`, `"w+b"`, `"a+b"`

Sample Program to Open a File for Writing

Editor:

```
#include <stdio.h>

int main() {
char ch; // to read characters from the file
FILE *file; // the FILE pointer

file = fopen("date.txt", "r"); // input file
if(file==NULL) {
  printf("Error: failed to open file.\n");
  return 1;
}
printf("File successfully opened. Contents...:\n\n");

while(1) {
  ch = fgetc(file);
  if(ch!=EOF) printf("%c", ch);
  else break;
```