Programming & Data Structure

CS 11002

Partha Bhowmick
http://cse.iitkgp.ac.in/~pb

CSE Department
IIT Kharagpur

Spring 2012-2013

Instructors & TAs (1)

Venue & Time

Room V1

Sections 1, 2, 3

MON + THURS + FRI – All 4:30-5:25

Instructors

Dr. Partha Bhowmick

(Email: `bhowmick@gmail.com`)

+ Prof. A. Basu + Prof. R. Mall

Instructors & TAs

Teaching Assistants

Mr Sanjoy Pratihar

(Email: `sanjoy.pratihar@gmail.com`)

Mr Kunal Banerjee

(Email: `kunal.banerjee.cse@gmail.com`)

Computer Science and Engineering Department

IIT Kharagpur

Marks Distribution

Class Test I	**7 Feb** 2013, 6:45–7:45 PM	10 marks
Mid-Sem Exam	18–26 Feb 2013	25 + 5[a]
Class Test II	**21 Mar** 2013, 6:45–7:45 PM	10 marks
End-Sem Exam	15 Apr 2013	45 + 5[b]

[a] 5 marks for pre-midsem attendance;
[b] 5 marks for post-midsem attendance.

Less than 75% attendance at any point of time is subject to de-registration.

Course Content

Introduction; syllabus; books; class attendance; class test dates; evaluation policy; compiler; OS; data; variables & constants; storage; flowcharts; printf() and scanf(); macros; math library; operators precedence; data types & their operations; data operations (type cast); control statements (if, else); loops (for, while, do-while); logical operations; number system (decimal, binary, hex); fractions.
Functions; parameters; arguments; declarations; prototypes; examples; recursion; array.
Arrays; linear search; binary search (recursive); applications; bubble sort; 2D arrays; strings; structures; array of structures; pointers; pointer arithmetic for arrays; dynamic memory allocation; calloc & malloc; call by ref.

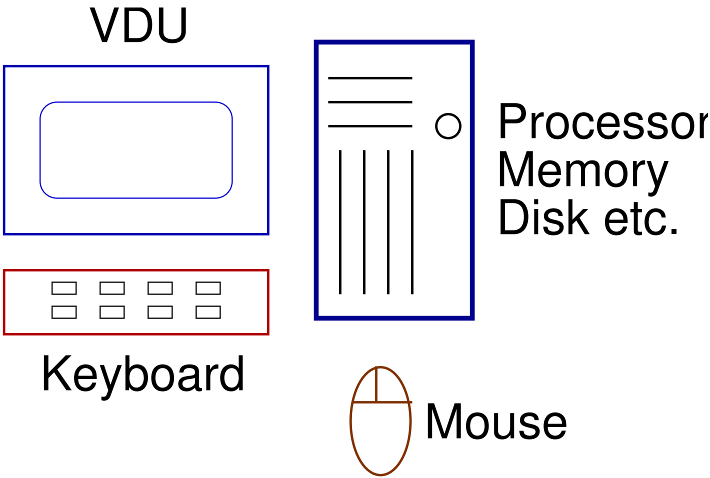
Course Content

File I/O; command line arguments; ADT; stack with array; linked list; circular list; double-ended list; queues; quick sort, etc.

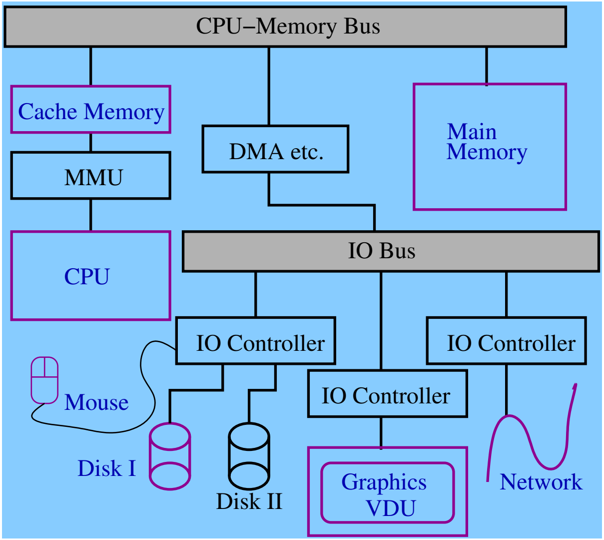
Text Books

1. **Programming with C**, B. S. Gottfried, **TMH**
2. **Data Structures**, S. Lipschutz, **TMH**
3. **The C Programming Language**, B. W. Kernighan & D. M. Ritchie, **PHI**
4. **Data Structures and Program Design**, R. L. Kruse, PHI
5. **Introduction to Algorithms**, T. H. Cormen et al., PHI

Computer System: *An Overview*



Computer System: *An Overview*



Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Stored program computer:** Processes data using CPU.
- **CPU** (Central Processing Unit): Follows a sequence of instructions, i.e., computer program.
- **Program:** A finite sequence of instructions.
- **Memory:** Stores both program and data in the computer.
- **Instruction Set:** A finite set of (machine) instructions associated with every CPU.

Computer & CPU

- **Program:** Finally, after compilation, it is an executable file or a binary file containing a finite sequence of machine instructions of the corresponding CPU.
- **Machine instruction:** A finite-length string of binary digits (bits).
- **CPU types:** Pentium, PowerPC, SPARC, x86-64 —all have different instruction sets! So the machine-language program of one computer need not run directly on another machine.

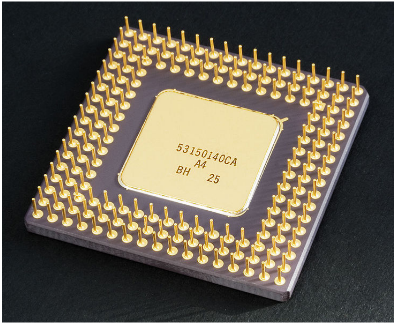
Computer & CPU

- **Program:** Finally, after compilation, it is an executable file or a binary file containing a finite sequence of machine instructions of the corresponding CPU.
- **Machine instruction:** A finite-length string of binary digits (bits).
- **CPU types:** Pentium, PowerPC, SPARC, x86-64 —all have different instruction sets! So the machine-language program of one computer need not run directly on another machine.

Computer & CPU

An Intel 80486DX2 CPU

from above

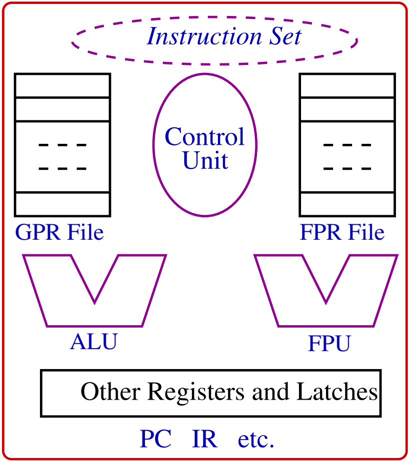


from below

Computer & CPU

Writable volatile random-access memory (RAM) —primarily used as main memory in personal computers, workstations, and servers.

Computer & CPU



Main Components of CPU

CU: Control unit extracts instructions from memory, decodes and executes them, calling on the ALU when necessary.
ALU: Arithmetic logic unit.
FPU: Floating-point unit (a math coprocessor).
GPR: General purpose registers.
FPR: Floating-point registers.
PC/IP: Program counter/ instruction pointer.
IR: Instruction register.

Operating System

A computer is very difficult to use unless a core master program is running on it. This core master program is called its **operating system (OS)**. The OS provides a better view of the available resources and also manages them efficiently.

Our Programming Environment

- PC with **Intel Core 2 Duo** CPU(+ MMU + cache).
- Operating System, **Linux** / **SunOS**.
- Editor: **emacs**, **gedit**, or **vi**.
- Compiler: **GNU gcc** for C language.
- **Thin Client** and **Server**.

A **thin client** is a computer (and necessary software) that does most its computational job on a more powerful **server**. Large number of clients share the same server.

First C Program

```
#include <stdio.h>
int main()
{
  printf("Hurray... My First C Program!\n");
  return 0;
} // first.c
```

Write, Compile, and Execute on Linux OS

1. Open a text editor **vi**, **emacs**, or **gedit**:
   `$ emacs first.c`
2. Write the C program and save it as `first.c`.
3. Compile `first.c` at $ prompt to get the executable file `first.out`:
   `$ cc first.c -o first.out`
   Caution: The file `first.c` should be in the current directory.
4. If there is an error, go back to the editor and fix it;
   otherwise, run the `first.out`:
   `$ ./first.out`

Write, Compile, and Execute on Linux OS

1. Open a text editor **vi**, **emacs**, or **gedit**:
   `$ emacs first.c`
2. Write the C program and save it as `first.c`.
3. Compile `first.c` at `$` prompt to get the executable file `first.out`:
   `$ cc first.c -o first.out`

   Caution: The file `first.c` should be in the current directory.
4. If there is an error, go back to the editor and fix it;
   otherwise, run the `first.out`:
   `$ ./first.out`

Write, Compile, and Execute on Linux OS

1. Open a text editor **vi**, **emacs**, or **gedit**:
   `$ emacs first.c`
2. Write the C program and save it as `first.c`.
3. Compile `first.c` at `$` prompt to get the executable file `first.out`:
   `$ cc first.c -o first.out`
   Caution: The file `first.c` should be in the current directory.
4. If there is an error, go back to the editor and fix it;
   otherwise, run the `first.out`:
   `$ ./first.out`

Write, Compile, and Execute on Linux OS

1. Open a text editor **vi**, **emacs**, or **gedit**:

   `$ emacs first.c`

2. Write the C program and save it as `first.c`.
3. Compile `first.c` at `$` prompt to get the executable file `first.out`:

   `$ cc first.c -o first.out`

   Caution: The file `first.c` should be in the current directory.

4. If there is an error, go back to the editor and fix it;

   otherwise, run the `first.out`:

   `$ ./first.out`

The Second Program

```
#include <stdio.h>
#define MAX 99
int main()
{
  int n;
  printf("Enter n: ");
  scanf("%d", &n);
  if(n>MAX)
    printf("\n Your number %d > %d\n", n, MAX);
  else
    printf("\n Your number %d <= %d\n", n, MAX);
} // second.c
```

Structure of a C Program

- A program in C language consists of **functions** and **declarations**.
  Ex: `printf(...)` is a function.
  `int n` or `int func(...)` are declarations.
- It also has **C preprocessor (cpp)** directives.
  Ex: `#include <...>` or `#define ...`
- A function named `main()` is mandatory.

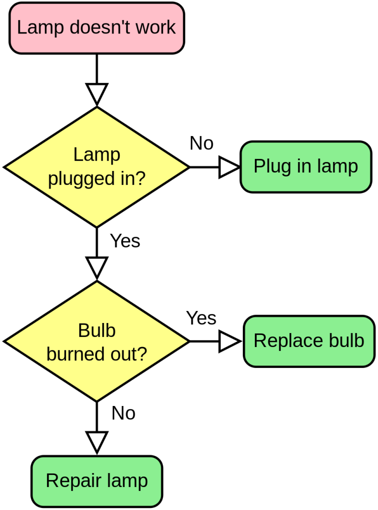
Structure of a C Program

- A program in C language consists of **functions** and **declarations**.
  Ex: `printf(...)` is a function.
  `int n` or `int func(...)` are declarations.
- It also has **C preprocessor (cpp)** directives.
  Ex: `#include <...>` or `#define ...`
- A function named `main()` is mandatory.

Structure of a C Program

- A program in C language consists of **functions** and **declarations**.
  Ex: `printf(...)` is a function.
  `int n` or `int func(...)` are declarations.
- It also has **C preprocessor (cpp)** directives.
  Ex: `#include <...>` or `#define ...`
- A function named `main()` is mandatory.

Flowchart


Lamp doesn't work
Lamp plugged in?
No
Plug in lamp
Yes
Bulb burned out?
Yes
Replace bulb
No
Repair lamp

Flowchart

A **flowchart** is a diagram (for a common people) representing an **algorithm or process**, showing the steps as boxes of various kinds, and their order by connecting them with arrows.

A flowchart provides an easy-to-understand, step-by-step solution to a given problem. Operations are shown in these boxes, and arrows connecting them represent the flow of control.

Flowchart

A **flowchart** is a diagram (for a common people) representing an **algorithm or process**, showing the steps as boxes of various kinds, and their order by connecting them with arrows.

A flowchart provides an easy-to-understand, step-by-step solution to a given problem.

Operations are shown in these boxes, and arrows connecting them represent the flow of control.

Flowchart

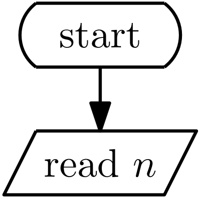
A **flowchart** is a diagram (for a common people) representing an **algorithm or process**, showing the steps as boxes of various kinds, and their order by connecting them with arrows.
A flowchart provides an easy-to-understand, step-by-step solution to a given problem.
Operations are shown in these boxes, and arrows connecting them represent the flow of control.

Flowchart

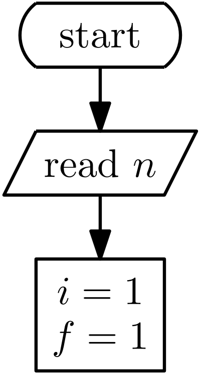
A **flowchart** is a diagram (for a common people) representing an **algorithm or process**, showing the steps as boxes of various kinds, and their order by connecting them with arrows.
A flowchart provides an easy-to-understand, step-by-step solution to a given problem.
Operations are shown in these boxes, and arrows connecting them represent the flow of control.

Flowchart to compute $n!$

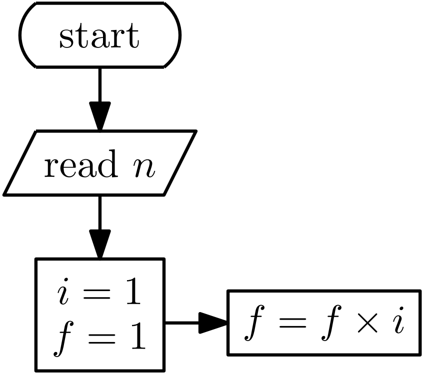
Flowchart to compute $n!$



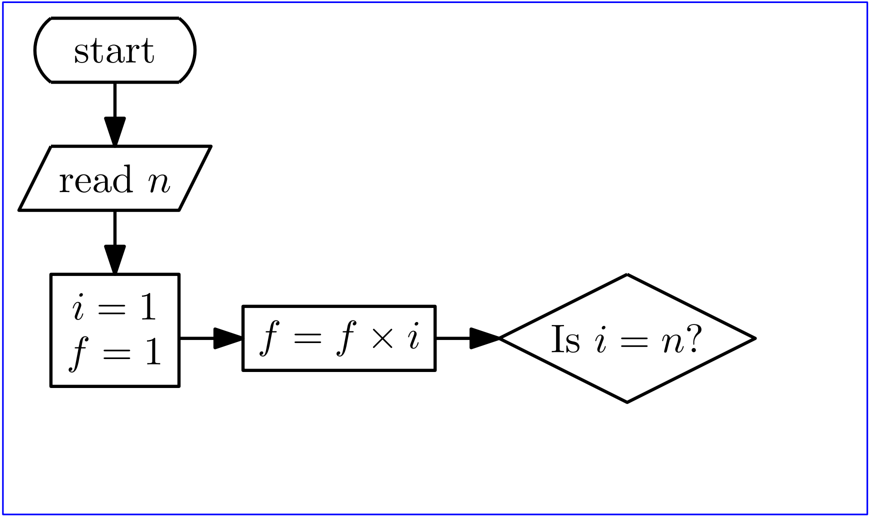
Flowchart to compute $n!$



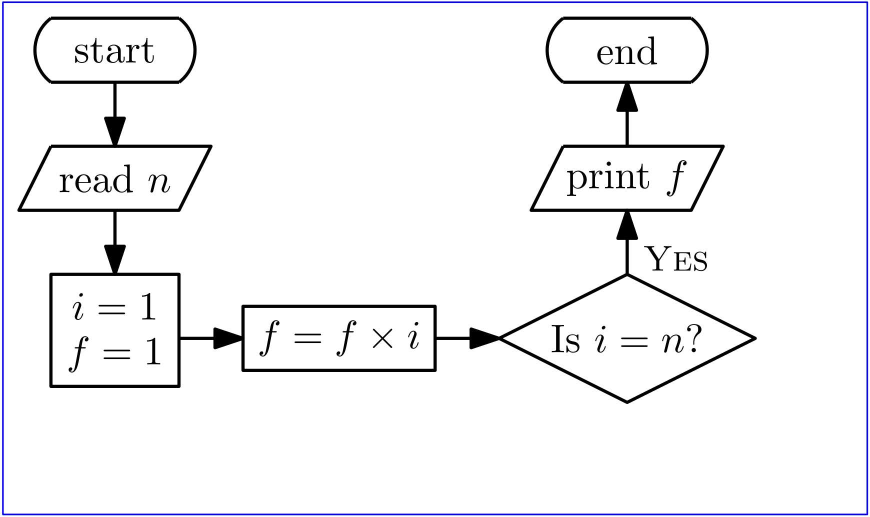
Flowchart to compute $n!$



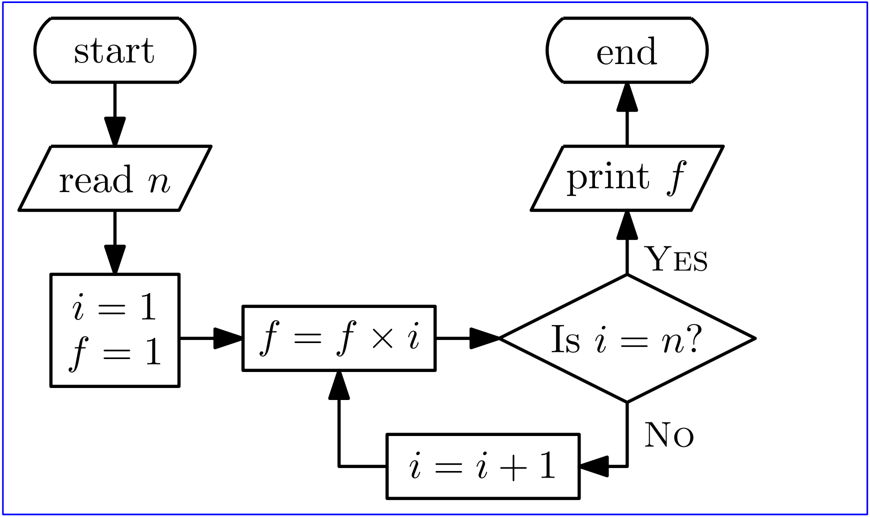
Flowchart to compute $n!$



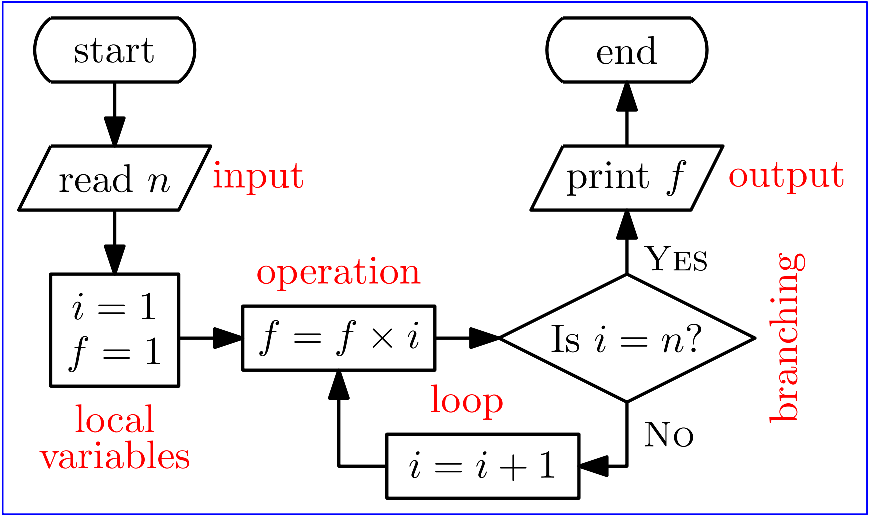
Flowchart to compute $n!$



Flowchart to compute $n!$



Flowchart to compute $n!$



C Function

A **function definition** in C language has

- a **name**
- a list of **parameters** (optional)
- **type** of the value it returns (if any)
- local variable **declarations** (if any)
- a sequence of **statements**
- comments (optional) for human understanding

```
int add(int a, int b){//sum of two numbers
  int c;
  c=a+b;
  return c;}
```

An Example with Three Functions (1)

```
#include <stdio.h> // cpp directive
int gcd(int, int); // Func declaration
int sum(int small, int large) // Def
{ int i, total = 0, temp;
  if(small> large){
    temp=small; small=large; large=temp;}
  for(i=small;i<=large;++i)
    total += i;
  return total;
} // Def ends
```

An Example with Three Functions (2)

```
int main() { // main function
  int large, small;
  printf("Enter two non-ve integers: ");
  scanf("%d%d", &large, &small);
  printf("%d + ... + %d = %d\n",
    small,large,sum(large,small));
  printf("GCD(%d, %d) = %d\n",
    large,small,gcd(large,small));
}
```

An Example with Three Functions (3)

```
int gcd(int large, int small) {
  // A recursive function
  if(small == 0)
    return large;
  else
    return gcd(small, large%small);
} // sample.c
```

Preprocessing

- File name of a C program ends with "`.c`".
  Ex: `sample.c`.
- `#include <stdio.h>`
  A line starting with `#` is a C preprocessor directive.
  This directive tells the preprocessor to include the **header file** for the standard I/O functions from the header files directory (often `/usr/include`).

Function invocation

Why declaring `int gcd(int, int);` before `main()`?

The function `gcd()` is called or invoked in `main()` before its definition.
The compiler translates the code sequentially, and hence it encounters the invocation of `gcd()` before its definition.
Without declaration, it does not have any clue about its return type and parameters.

Function invocation (2)

The declaration `int gcd(int, int);` provides the necessary information regarding its parameters and return type. It is known as **prototype** or **interface** of the function.

Note: The header file `stdio.h` provides the prototypes of `printf()` and `scanf()`.

Function definition

```
int sum(int small, int large) // Def
{
  int i, total = 0;
  if(small > large){ int temp=small;
     small=large; large=temp;
  }
  for(i=small;i<=large;++i) total += i;
  return total;
} // Def ends
```

The actual function definition specify name, parameters, computation, and return value.

Variable Declaration

```
int i, total = 0;
```

- `i` and `total` are variables **local** to the function `sum()`.
- Currently `i` does not have any value (contains garbage), but `total` is initialized to zero (0).

Variable Types

Basic: `char c; int n; float x; double z;`
Modifiers: `signed char`, `unsigned char`, `unsigned int`, `long double`, etc.

Type	Size	Range
`char`	1 byte	$[-128, 127]$
`int`	4 bytes	$[-2147483648, 2147483647]$
`float`	4 bytes	$\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}]$
`double`	8 bytes	$\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$

1 byte = 8 bits.

Statements

```
small=large; large=temp; ...
if(small > large){...}
for(i=small;i<=large;++i) ...;
return total;
```

Operators

Assignment: `=`
Arithmetic: `+, -, *, /, %, ++, --`
Relational: `==, !=, <, <=, >, >=`
Special assignment: `+=, -=, *=, /=, %=`
Logical: `&&, ||`

Library Functions

Library functions are supplied along with the compiler.

Ex:
Reads from Keyboard: `scanf()`
Writes on the VDU: `printf()`

Compile and Run

```
$ cc sample.c
$ ./a.out
Enter two non-ve integers
12 18
18 + ...  + 12 = 105
GCD(12, 18) = 6
```

Note: Replace `cc` by `gcc` in the laboratory.

Macro Definition: `#define`

#define *identifier tokens*

tokens should be at least one in number

Examples:

```
#define MAX 100
#define PI 3.14
#define HI(a,b) (((a)>(b))? (a):(b))
```

Macro Preprocessing

Original code

```
#include <stdio.h>
#define PI 3.14 //approx
int main() {
  float r,cir;
  printf("radius=");
  scanf("%f",&r);
  float cir=2*PI*r;
  printf("2*PI*r=%f",x);
  return 0;
}//perimeter.c
```

After preprocessing

Macros replaced & comments removed!

```
int main() {
  float r,cir;
  printf("radius=");
  scanf("%f",&r);
  float cir=2*3.14*r;
  printf("2*PI*r=%f",x);
  return 0;
}
```

Explain: Why `PI` is not replaced by `3.14` in `printf("2*PI*r=%f",x)`?

Macro Definition with Parameters

```
#include <stdio.h>
#define EXCH(X,Y,T) ((T)=(X),(X)=(Y),(Y)=(T))
int main() {
    int m, n, temp;
    scanf("%d%d", &m, &n);
    printf("m: %d, n: %d\n", m, n);
    EXCH(m,n,temp);
    printf("m: %d, n: %d\n", m, n);
    return 0;
} // preProc2.c
```

After Substitution

```
int main() {
    int m, n, temp;
    scanf("%d%d", &m, &n);
    printf("m: %d, n: %d\n", m, n);
    ((temp)=(m), (m)=(n), (n)=(temp));
    printf("m: %d, n: %d\n", m, n);
    return 0;
}
```

Parenthesis of Macros

Use of parenthesis around the parameters is safer. Otherwise there may be semantic error.

Ex:

```
#define MULT(X,Y) X*Y
```
← **wrong**

```
........
printf("2*(m+n): %d\n", MULT(2,m+n));
```

After substitution:

```
printf("2*(m+n): %d\n", 2*m+n);
```

Correct:

```
#define MULT(X,Y) (X)*(Y)
```

Math Library

```
#include <stdio.h>
#include <math.h> //math library
int main(){
  float r, a;
  printf("Enter r:"); scanf("%f", &r);
  a = 4.0 * atan(1.0) *r*r;
  printf("r=%f: Cir Area=%f\n", r, a);
  return 0;} //cirArea.c
```

To compile with **math library** (for `atan()`):

```
$ cc -lm cirArea.c -o cirArea.out
```

Formatted Output

```
#include <stdio.h>
#include <math.h> //math library
int main(){
  float r, a;
  printf("Enter r:"); scanf("%f", &r);
  a = 4.0 * atan(1.0) *r*r;
  printf("r=%6.3f: Cir Area=%6.2f\n", r,a);
  return 0;}
}
```

Formatted Output

`%6.2f` means the printed number will be of at least 6 characters—including digits, decimal point, and leading blanks—with always 2 characters in decimal place.

Ex:

```
......
printf("r=%6.3f: Cir Area=%6.2f\n", r,a);
......
```

```
$./cirArea.out
Enter r:1.0169
r= 1.017:  Cir Area=  3.25
```

if-else statement

```
#include <stdio.h>
int main(){
  int a, b, max;
  printf("Enter two integers: ");
  scanf("%d%d", &a, &b);
  if(a>b) max=a;
  else max=b;
  printf("Max(%d,%d)=%d.\n", a,b,max);
} //max.c
```

for loop

```
#include <stdio.h>
int main(){
  int n, i, sum=0;
  printf("Enter a +ve integer: ");
  scanf("%d", &n);
  for(i=1; i<=n; ++i) sum += i;
  printf("\nSum of 1+...+%d=%d.\n",n,sum);
}//sumn.c
```

Recursive Function

```
#include <stdio.h>
int sum(int n){
  if(n==0) return 0;
  else return n+sum(n-1);
}
int main() {
  int n;
  printf("Enter a +ve integer: ");
  scanf("%d", &n);
  printf("\nSum of 1+...+%d=%d.\n",n,sum(n));
}//sumnRec.c
```

Data

- Data is stored in the memory as a string of binary digits (`0` and `1`) having finite length.
- In a machine instruction, a memory location is identified by its **address**.
- In a high-level language like C or C++, a location is identified with a **name**, called a **variable**. A variable is bound to a memory location.
- Data can be read from a memory location and a memory location can also be updated.

Data

- Data is stored in the memory as a string of binary digits (`0` and `1`) having finite length.
- In a machine instruction, a memory location is identified by its **address**.
- In a high-level language like C or C++, a location is identified with a **name**, called a **variable**. A variable is bound to a memory location.
- Data can be read from a memory location and a memory location can also be updated.

Data

- Data is stored in the memory as a string of binary digits (`0` and `1`) having finite length.
- In a machine instruction, a memory location is identified by its **address**.
- In a high-level language like C or C++, a location is identified with a **name**, called a **variable**. A variable is bound to a memory location.
- Data can be read from a memory location and a memory location can also be updated.

Data

- Data is stored in the memory as a string of binary digits (`0` and `1`) having finite length.
- In a machine instruction, a memory location is identified by its **address**.
- In a high-level language like C or C++, a location is identified with a **name**, called a **variable**. A variable is bound to a memory location.
- Data can be read from a memory location and a memory location can also be updated.

Data

- Data is stored in the memory as a string of binary digits (`0` and `1`) having finite length.
- In a machine instruction, a memory location is identified by its **address**.
- In a high-level language like C or C++, a location is identified with a **name**, called a **variable**. A variable is bound to a memory location.
- Data can be read from a memory location and a memory location can also be updated.

Types of Data

In a high-level language:

- Data can be of many different types: **integers**, **rational numbers**, **real numbers**, **complex numbers**, **vectors**, **2D/3D points**, **matrices**, **characters**, etc.
- Some are **built-in** or **primitive** data types: `char`, `int`, `float`.
- Complex data types can be defined by **type constructors**.

Types of Data

In a high-level language:

- Data can be of many different types: **integers**, **rational numbers**, **real numbers**, **complex numbers**, **vectors**, **2D/3D points**, **matrices**, **characters**, etc.
- Some are **built-in** or **primitive** data types: `char`, `int`, `float`.
- Complex data types can be defined by **type constructors**.

Types of Data

In a high-level language:

- Data can be of many different types: **integers**, **rational numbers**, **real numbers**, **complex numbers**, **vectors**, **2D/3D points**, **matrices**, **characters**, etc.
- Some are **built-in** or **primitive** data types: `char`, `int`, `float`.
- Complex data types can be defined by **type constructors**.

Simple Variable Declaration in C

Built-in data types of C language

```
char flag, grade = 'B';
int count, index = 1;
float interest=7.25, principal=5000.0,
year;
```

int

- `int` has only (4 bytes =) **32 bits**.
- Its representation is in **2's complement form.**

  Ex:
  $00000101_2 = 11111010 (1\text{'s complement}) = 11111010 + 1 = 11111011 (2\text{'s complement})$
- Its range is $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$.

int

- `int` has only (4 bytes =) **32 bits**.
- Its representation is in **2's complement form.**

  Ex:
  $00000101_2 = 11111010(1\text{'s complement}) = 11111010 + 1 = 11111011(2\text{'s complement})$
- Its range is $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$.

int

- `int` has only (4 bytes =) **32 bits**.
- Its representation is in **2's complement form.**

  Ex:
  $00000101_2 = 11111010(1\text{'s complement}) = 11111010 + 1 = 11111011(2\text{'s complement})$
- Its range is $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$.

float

- A real number may have infinite information content (irrational numbers) that cannot be stored in a finite computer.
- Data type `float` is an approximation of real numbers with a fixed **32-bit** size.
- Special values such as `nan` (not a number, e.g., $\sqrt{-1}$) and `inf` (infinity: 1.0/0.0) are defined to handle errors in floating-point operation.

float

- A real number may have infinite information content (irrational numbers) that cannot be stored in a finite computer.
- Data type `float` is an approximation of real numbers with a fixed **32-bit** size.
- Special values such as `nan` (not a number, e.g., $\sqrt{-1}$) and `inf` (infinity: 1.0/0.0) are defined to handle errors in floating-point operation.

float

- A real number may have infinite information content (irrational numbers) that cannot be stored in a finite computer.
- Data type `float` is an approximation of real numbers with a fixed **32-bit** size.
- Special values such as `nan` (not a number, e.g., $\sqrt{-1}$) and `inf` (infinity: 1.0/0.0) are defined to handle errors in floating-point operation.

`char` is a Short Integer

- In the binary world of computer, every data—primitive or constructed—is encoded as a bit string of finite length.
- The useful set of characters is encoded as a set of 8-bit (one byte) or 16-bit integers.
- The C language uses 8-bit ASCII encoding.[1]

[1] ASCII stands for `American Standard Code for Information Interchange`.

`char` is a Short Integer

- In the binary world of computer, every data—primitive or constructed—is encoded as a bit string of finite length.
- The useful set of characters is encoded as a set of 8-bit (one byte) or 16-bit integers.
- The C language uses 8-bit ASCII encoding.[1]

[1] ASCII stands for `American Standard Code for Information Interchange`.

`char` is a Short Integer

- In the binary world of computer, every data—primitive or constructed—is encoded as a bit string of finite length.
- The useful set of characters is encoded as a set of 8-bit (one byte) or 16-bit integers.
- The C language uses 8-bit ASCII encoding.[1]

[1] ASCII stands for `American Standard Code for Information Interchange`.

A few ASCII Codes

char	decimal	binary	hex
0	48	0011 0000	30
9	57	0011 1001	39
A	65	0100 0001	41
Z	90	0101 1010	5a
a	97	0110 0001	61
z	122	0111 1010	7a

Binary to Hex

It is tedious to write a long string of binary digits. A better way is to use **radix-16** or **hexadecimal (Hex)** number system with 16 digits $\{0, 1, \cdots, 9, A(10), B(11), C(12), D(13), E(14), F(15)\}$.

To convert from binary to hex representation, the bit string is grouped in blocks of **4 bits (nibble)** from the least significant side. Each block is replaced by the corresponding **hex** digit.

Binary to Hex

$$
\begin{array}{cccccccc}
0011 & 1110 & 0101 & 1011 & 0001 & 1101 & 0110 & 1001 \\
 & & & & \Downarrow & & & \\
3 & E & 5 & B & 1 & D & 6 & 9
\end{array}
$$

We write `0x3E5B1D69` (in upper or lower case) for a hex constant in C language.

Binary to Hex

int Data

7529_{10}
$=$ 0000 0000 0000 0000 0001 1101 0110 1001_2
$=$ $00001D69_{16}$ $=$ 0x00001D69 = 0x1D69

-7529_{10}
$=$ 1111 1111 1111 1111 1110 0010 1001 0111_2
$=$ 0xFFFFE297

We shall discuss about this representation afterward.

Binary to Hex

`float` Data

7529.0_{10}
$= \texttt{0 1000 1011 110 1011 0100 1000 0000 0000}_2$

-7529.0_{10}
$= \texttt{1 1000 1011 110 1011 0100 1000 0000 0000}_2$

This representations are different from that of 7529 or −7529.

Binary to Hex

char Data

A $= 0100\ 0001_2 =$ 0x41

1 $= 0011\ 0001_2 =$ 0x31

char 1 is not same as int 1 or float 1.0.

Few Other Built-in Types of C

- `unsigned int (unsigned)`:
  32-bit unsigned binary,
  0 to $2^{32} - 1 = 4294967295$.
- `long int`: same as `int`.
- `long long int`:
  64-bit signed binary,
  $-2^{63} = 9223372036854775808$ to
  $2^{63} - 1 = 9223372036854775807$.
- `double`:
  64-bit IEEE 754 double-precision format.

Constants of Primitive Types

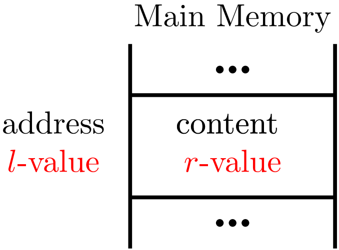
- `int`: `123, −123`
- `float`: `1.23, -1.23e-02`
- `char`: `A, 5, %`

A floating-point constant is often taken in double-precision format.

A Variable and Its Memory Location

Either the compiler generates code to allocate memory or it is allocated when the process image (e.g., `a.out`) is loaded.

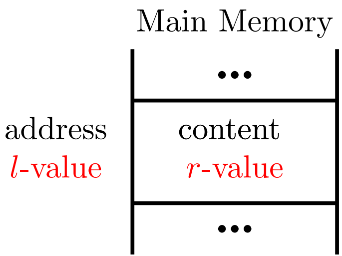
The allocated memory location has an **address** or *l*-**value** and a **content** or *r*-**value**.



A Variable and Its Memory Location

Either the compiler generates code to allocate memory or it is allocated when the process image (e.g., `a.out`) is loaded.

The allocated memory location has an **address** or ***l*-value** and a **content** or ***r*-value**.



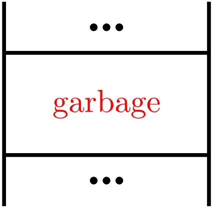
A Variable and Its Memory Location

The allocated space is of **fixed size** to store the data of the specified type; e.g., 4 bytes for `int`.

Unless initialized, the **content** or the *r-value* is `undefined` after the declaration.

`int count;`

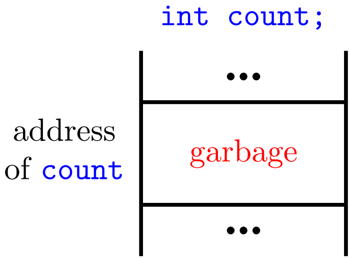
address of `count`



A Variable and Its Memory Location

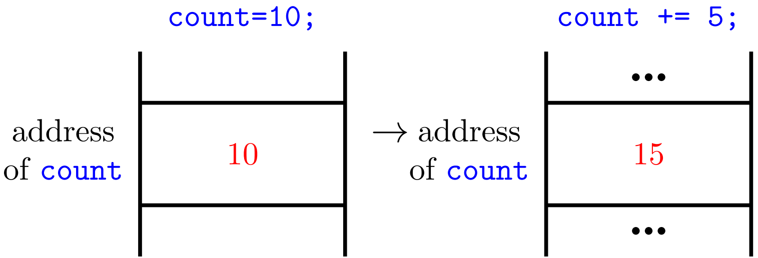
The allocated space is of **fixed size** to store the data of the specified type; e.g., 4 bytes for `int`.

Unless initialized, the **content** or the *r-value* is `undefined` after the declaration.



A Variable and Its Memory Location

The *r-value* can be initialized or updated.



Pointer

- The **address** or ***l*-value** of a variable can be extracted using the **unary operator** '`&`'.
- This ***l*-value** can be stored in another variable of type `int *` known as **pointer type**.

```
int count = 10, *cP;
cP = &count;
```

Memory Locations for Other Types

```
float cgpa;
char grade;
```

- Memory allocations are similar for other data types, e.g., `float` and `char`.
- The only difference is the `size` of the allocated space.

Constant: `const` (1)

A declaration can be qualified to define a name of a constant.

```
const double pi = 3.14159265358979323846
```

In this case we cannot modify `pi`; its value is stored in the **read-only** memory segment.

Constant: `const`


```
#include <stdio.h>
int main() {
  const double pi = 3.141592653589793 2;
  pi = pi + 1; return 0;
}//const.c
```

```
$ cc const.c
const.c: In function 'main':
const.c:4: error: assignment of
read-only variable 'pi'
$
```

Reading char Data (1)

A program expected to read two characters from two lines.

```
#include <stdio.h>
int main() {
  char c, d;
  printf("Enter two characters: ");
  scanf("%c", &c);
  scanf("%c", &d);
  printf("%c..%c\n", c, d);
  return 0;
} // charRead.c
```

Reading char Data (2)

```
$ cc charRead.c
$ a.out
Enter two characters:  1
1..
$
```

Why? It does not read the second character. The reason is that pressing of *Enter key* injects a *non-printable character* `\n` (newline) in the input stream.

Reading `char` Data

Replace: `printf("%c..%c\n", c, d);`
by: `printf("%c..%d\n", c, d);`

```
$ cc charRead.c
$ a.out
Enter two characters: 1
1..10
$
```

Why `10!?` It's the ASCII value of `\n` (newline).

Reading char Data

To read proper input,
Replace: `scanf("%c", &d);`
by: `scanf(" %c", &d);` ← **A gap before %c**

```
$ cc charRead.c
$ a.out
Enter two characters: 1
2
1..2
```

How? The **gap** is matched with `\n`.

Basic Assignment and Arithmetic Operators

Assignment Operator =

Assignment Operator =

```
int count;
count = 10;
```

- The first line declares the variable `count`.
- In the second line, the **assignment operator** (=) is used to store 10 in the location of `count`.
- In C language, `count = 10` is called an **expression**.
  Value of the expression here is 10.
- The semicolon converts the expression to a **statement**.

Assignment Operator =

```
int count;
count = 10;
```

- The first line declares the variable `count`.
- In the second line, the **assignment operator** (`=`) is used to store `10` in the location of `count`.
- In C language, `count = 10` is called an **expression**.
  Value of the expression here is `10`.
- The semicolon converts the expression to a **statement**.

Assignment Operator =

```
int count;
count = 10;
```

- The first line declares the variable `count`.
- In the second line, the **assignment operator** (`=`) is used to store `10` in the location of `count`.
- In C language, `count = 10` is called an **expression**.
  Value of the expression here is `10`.
- The semicolon converts the expression to a **statement**.

Assignment Operator =

```
int count;
count = 10;
```

- The first line declares the variable `count`.
- In the second line, the **assignment operator** (`=`) is used to store `10` in the location of `count`.
- In C language, `count = 10` is called an **expression**.
  Value of the expression here is `10`.
- The semicolon converts the expression to a **statement**.

C expression

C expression

`count = 2*count + 5;`

- Here the variable `count` is used on both sides of the assignment operator. There are two constants: `2` and `5`, and three operators: `=` (assignment), `*` (multiplication) and `+` (addition).
- `count = 2*count + 5` is an expression and `count = 2*count + 5;` is a statement.

Type Casting

Type Casting

- A `float` data can be assigned to an `int` variable:
  `int count = (int)7.5;`
  But there may be loss of precision.
- An `int` data can also be assigned to a variable of type `float`:
  `float cgpa = (float)2147483647;`
  But here also there may be loss of information.
- This process is called **type casting**.

Type Casting

- A `float` data can be assigned to an `int` variable:
  `int count = (int)7.5;`
  But there may be loss of precision.
- An `int` data can also be assigned to a variable of type `float`:
  `float cgpa = (float)2147483647;`
  But here also there may be loss of information.
- This process is called **type casting**.

Type Casting

- A `float` data can be assigned to an `int` variable:
  `int count = (int)7.5;`
  But there may be loss of precision.
- An `int` data can also be assigned to a variable of type `float`:
  `float cgpa = (float)2147483647;`
  But here also there may be loss of information.
- This process is called **type casting**.

Type Casting Error


```
#include <stdio.h>
int main() {
  int count = (int)7.5;
  float cgpa = (float)2147483647;
  printf("count: %d\n", count);
  printf("cgpa: %e\n", cgpa);
  return 0; }
```

```
$ cc temp.c
$ ./a.out
count:  7
cgpa:  2.147484e+09
```

Type Casting Error (2)

```
#include <stdio.h>
int main() { // floatEq.c
  float a = 1.3;
  if (a == 1.3) printf("1. Equal\n");
  else printf("1. Not equal\n");
  if (a == (float)1.3) printf("2. Equal\n");
  else printf("2. Not equal\n");
  return 0;}
```

Type Casting Error (3)

```
$ cc floatEq.c
$ ./a.out
1.  Not equal
2.  Equal
```

Why Type Casting Error (1)

- The assignment of a floating-point data to an `int` variable or vice versa is not a simple operation due to the difference in their internal representations.
- For `int count = (int)7.5`, the fractional part is removed and `7` is stored in 32-bit integer representation (2's complement form).

Why Type Casting Error (2)

- For `float cgpa = (float)2147483647`, the integer `2147483647` is converted to floating-point form in IEEE 754 single-precision format. In this format, a lesser number of bits (23 bits) are available for storing the significant digits, resulting to a loss of precision.

Arithmetic Operators

Five Basic Arithmetic Operators

+ (addition), - (subtraction), * (multiplication), / (division), % (modulo or mod).

a%b produces the remainder when a is divided by b. Here, the first operand a should be a non-negative integer and the second operand b should be a positive integer.

Operation mod (%)

```
#include <stdio.h>
int main() {
  printf("0%10 = %d\n", 0%10);
  printf("10%4 = %d\n", 10%4);
  printf("-10%4 = %d\n", -10%4);
  return 0; }
```

```
$ cc temp2.c
$ ./a.out
0%10 = 0
10%4 = 2
-10%4 = -2
```

Caution: The operator % does not extract the remainder correctly for negative operands.

Pre- and Post-Increments/Decrements (1)

```
int count = 10, total = 10;
++count;
total++;
```

- `++count` implies *pre-increment* and `total++` implies *post-increment*.
- After execution of the corresponding statements, the value of each location is `11`.
- But the value of the expression `++count` is `11` and that of `total++` is `10`.

Pre- and Post-Increments/Decrements (2)

Similarly we have pre- and post-decrement operators:

```
int count = 10, total = 10;
--count;
total--;
```

More Assignment Operators

```
int count = 10, total = 10;
count += 5*total;
```

The meaning of the expression:

`count += 5*total` is

`count = count + 5*total`.

Operator Overloading

The first four operators ($+, -, *, /$) can be used for `int`, `float`, and `char` data[2]. But the fifth operator (`%`) cannot be used on `float` data.

[2] The actual operations of addition, subtraction, etc. on `int` and `float` data are quite different due to the difference in their representations.

Mixed Mode Operations (1)

- Mixed mode operations among `int`, `float`, and `char` data are permitted.
- If one operand is of type `float` and the other one is of type `int`, then the `int` data will be converted to the closest `float` representation before performing the operation.

Mixed Mode Operations (2)

```
int n = 4;
float a = 2.5;
char c = 'a'; // ASCII value 97
printf("%d*%f = %f\n", n, a, n*a);
printf("%d*%f+%c = %f\n",n,a,c,n*a+c);
```

```
$ ./a.out
4*2.500000 = 10.000000
4*2.500000+a = 107.000000
```

Mixed Mode Operations

Caution: Error may creep in during *division* on `int` data.

Examples:

```
printf("1/3*30.0=%f\n", 1/3*30.0);
```
⇒ `1/3*10.0=0.000000`

```
printf("10.0*1/3=%f\n", 10.0*1/3);
```
⇒ `10.0*1/3=3.333333`

```
printf("10.0*(1/3)=%f\n", 10.0*(1/3));
```
⇒ `10.0*(1/3)=0.000000`

Precedence and Associativity

Precedence and Associativity (1)

- +, -, *, / have **left-to-right** associativity.
- *, /, % have the same precedence, and it is higher than + and -, which also have the same precedence.

Precedence and Associativity (2)

= is Right Associative

```
int count = 10, n ;
n = count = 2*count + 5;
```

The variable `n` gets the updated value of `count`, i.e., 25.

Precedence of =

The precedence of assignment operator(s) is lower than every other operator except the comma (`,`) operator.

Precedence and Associativity

Unary ++ and --

The unary `++` and `--` have higher precedence than `*`, `/`, `%`.

Errors in Computer Arithmetic

Overflow Problem

$2147483647 + 1 = -2147483648 \leftarrow$ **range overflow**

Range Overflow Problem

An example: $2147483647 + 1 = -2147483648$

```
#include <stdio.h>
int main() { // intOverflow.c
  int n = 2147483647;
  printf("n+1: %d\n", n+1);
  return 0; }
```

```
$ cc intOverflow.c
$ ./a.out
n+1:  -2147483648
```

Precision Loss (1)

An example: $10^5 + 10^{-5} = 10^5$

```
#include <stdio.h>
int main() { // lossPreci.c
  float a = 1.0e-40, b = 1.0e+5, c;
  c = a+b;
  printf("%e + %e = %e\n", a, b, c);
  if(b == a+b) printf("Equal\n");
  else printf("not Equal\n");
  return 0;}
```

Precision Loss


```
$ cc lossPreci.c
$ a.out
9.999946e-41 + 1.000000e+05 = 1.000000e+05
Equal
```

Law of Associativity Fails! (1)

An example: $0.3 \times 10^{-14} + (0.3 \times 10^{-14} + 10^5) \neq (0.3 \times 10^{-14} + 0.3 \times 10^{-14}) + 10^5$

```
#include <stdio.h>
int main() { // lawAsso.c
  float a=0.3e-14, b=0.3e-14, c=1.0e+5;
  if(a+(b+c) == (a+b)+c) printf("Equal\n");
  else printf("not Equal\n");
  return 0; }
```

Law of Associativity Fails! (2)

```
$ cc lawAsso.c
$ a.out
not Equal
$
```

Division by Zero

Division of `int` data by zero gives error at run time.

```
#include <stdio.h>
int main(){ //divIntZero.c
  int n = 10, m;
  printf("Enter an integer: ");
  scanf("%d", &m);
  printf("n/m: %d\n", n/m);
  return 0;}
```

Division by Zero


```
$ cc divIntZero.c
$ ./a.out
Enter an integer:  0
Floating point exception
```

Division by Zero

Division of `float` or `double` data by zero does not generate any error at run time. The result is `inf`, which can be used if needed.

```
#include <stdio.h>
#include <math.h>
int main(){ //divFloatZero.c
  float n = 10.0, m, r;
  printf("Enter a number: ");
  scanf("%f", &m);
  printf("n/m= %f\n", r = n/m);
  printf("atan(%f) = %f\n", r, atan(r));
  return 0;}
```

Division by Zero


```
$ cc divFloatZero.c -lm
$ a.out
Enter a number:  0
n/m= inf
atan(inf) = 1.570796
```

Integer ↔ Character (1)

- If a `char` data (8 bits) is assigned to an `int` type variable (32 bits), then the ASCII value of the `char` data is stored in the location of the `int` type variable.
- But if an `int` data is assigned to a `char` type variable, then the *least significant 8 bits* of the `int` data are stored in the location of `char` type variable.

Integer ↔ Character (2)

```
#include <stdio.h>
int main(){ // int2char.c
  int count='C'; char grade=1345;
  printf("count=%d, grade=%c\n",count,grade);
  return 0;}
```

```
$ cc int2char.c
int2char.c:  In function 'main':
int2char.c:3:  warning:  overflow in
implicit constant conversion
$ ./a.out
count=67, grade=A
```

Integer ↔ Character (3)

Reasons

- **Why** `int count='C'` **gives** `count=67`:
  ASCII value of `C` is `67`, which is stored in the location of `count`.
- **Why** `char grade=1345` **gives** `grade=A`:
  Binary representation of `1345` is `0000 0000 0000 0000 0000 0101 0100 0001`.
  The decimal value of the least significant byte (8 bits) is `65`, which is the ASCII value of `A`.

Expression

&

Statement

Expression & Statement (1)

- A **pure expression** has a **value**, e.g., 2, -2, $-a$, $-2 * a + b$.
- A **command** or **statement** changes the content of a location but does not have a value.
- In C language, many expressions are **impure** and cause **side effects** by changing values of locations, e.g. `++count`, `n = 2*m + 4`.

Expression & Statement (2)

- Any **expression** in C (with or without any side effect) can be converted to a **statement** by putting a semicolon at the end. These are called **expression statements**.
- This blurs the distinction between an expression and a command in C language.
- A semicolon in C language, unlike Algol or Pascal languages, does not compose two statements to form a new statement. Rather, it forms or terminate a statement.

Expression & Statement

- A semicolon itself may be viewed as **null statement** (no operation).

Compound Statement (1)

- A sequence of statements within a pair of **curly braces** forms a single **compound statement** or **block**.
- Variables can be declared within a block and are local to the block.
- A name clash is resolved in favor of the local object/block.

Compound Statement (2)

```
#include <stdio.h>
int main(){ //blockVar.c
  int a = 10, b = 20, c = 30;
  { int b = 200, c = 300;
    { int c = 3000;
      printf("L3> a=%d, b=%d, c=%d\n",
                  a, b, c);}
    printf("L2> a=%d, b=%d, c=%d\n",
                  a, b, c);}
  printf("L3> a=%d, b=%d, c=%d\n",
                  a, b, c);
  return 0;}
```

Compound Statement


```
$ cc blockVar.c
$ ./a.out
L3> a=10, b=200, c=3000
L2> a=10, b=200, c=300
L3> a=10, b=20, c=30
```

Change in Control Flow

- Depending on data, it may be necessary to perform different sets of operations in a program.
- This calls for **control flow** to make data-dependent choice of the execution statements.

Example. Write a C Program that reads two `int` data from the keyboard, finds the larger among them, and prints it on the VDU (screen).

Change in Control Flow


```
#include <stdio.h>
int main(){ //findLarger.c
  int a, b, larger;
  printf("Enter two integer data: ");
  scanf("%d%d", &a, &b);
  if (a > b) larger = a;
  else larger = b;
  printf("\nlarger=%d\n",larger);
  return 0;}
```

if Statement

We use a command called **if-statement** for controlling the execution sequence in `findLarger.c`.

Structure of if-statement:

if (*expression*) $statement_1$ else $statement_2$

if (*expression*) $statement_1$

In `findLarger.c`, we use the first type:

expression: `a > b`

$statement_1$: `larger = a;`

$statement_2$: `larger = b;`

Relational and Boolean Expressions (1)

Two new types of expressions are used in `if`-statement and other control-flow constructs of C language. They are called **relational** and **boolean** expressions.

C language does not have distinct truth values (TRUE and FALSE). Rather, the value **zero (0)** is treated as FALSE and any **non-zero** value is treated as TRUE.

Relational and Boolean Expressions (2)

```
#include <stdio.h>
int main(){ //TrueFalse.c
    int a;
    scanf("%d", &a);
    if (a) printf("non-zero\n");
    else printf("zero\n");
    return 0;}
```

Relational and Boolean Expressions (3)

```
$ cc TrueFalse.c
$ ./a.out
0
zero
$ ./a.out
-1
non-zero
$ ./a.out
1
non-zero
```

Relational Operators (1)

Following are the relational operators with their usual meaning.
`==` (equal to), `!=` (not-equal to), `<` (less than) `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to).

The usual operands of relational operators are `int`, `float`, `char`, etc. Their values are `boolean`.

Logical Operators (1)

`&&` (logical and), `~` (logical not), `||` (logical or).

The operands and values of logical operators are boolean values. Find out the precedence and associativity of these operators from the book.

if-else Statement (1)

To find the largest among three integers.

Version 1

```
#include <stdio.h>
int main(){
  int a, b, c, largest;
  printf("Enter three integers: ");
  scanf("%d%d%d", &a, &b, &c);
  if (a > b) largest = a; else largest = b;
  if (c > largest) largest = c;
  printf("\nlargest = %d\n", largest);
  return 0;}
```

if-else Statement (2)

Version 2

```
#include <stdio.h>
int main(){
  int a, b, largest;
  printf("Enter three integers: ");
  scanf("%d%d%d", &largest, &a, &b);
  if (a > largest) largest = a;
  if (b > largest) largest = b;
  printf("\nlargest = %d\n", largest);
  return 0;}
```

We use three variables but one input data may be lost at the end.

if-else Statement (3)

Version 3

```
#include <stdio.h>
int main(){
  int a, largest;
  printf("Enter three integers: ");
  scanf("%d%d", &largest, &a);
  if (a > largest) largest = a;
  scanf("%d", &a);
  if (a > largest) largest = a;
  printf("\nlargest = %d\n", largest);
  return 0;}
```

We use two variables but two input data may be lost at the end.

if-else Statement (4)

Version 4

```
#include <stdio.h>
int main(){
  int a, b, c, largest;
  printf("Enter three integers: ");
  scanf("%d%d%d", &a, &b, &c);
  if (a > b)
    if (a > c) largest = a;
    else largest = c;
  else if (b > c) largest = b;
  else largest = c;
  printf("\nlargest = %d\n", largest);
  return 0;}
```

`if-else` Statement (5)

This is an example of **nested `if` statement**. No input data is lost in this case.

Note

Statements within the `if` and the `else` parts may be *compound statements*.

$$\texttt{if}\ (expression)\ \{\quad statement_1 \quad \ldots \quad statement_k \quad \}$$

`if-else` Statement


```
if (expression) {
        statement_1
        ...
        statement_k
    }
else    {
        statement_1
        ...
        statement_m
    }
```

`if-else` Statement (7)

Proper bracing

`if` and `else-if` statements can be nested.
The `else` part will be associated to the nearest `if`.
It is better to use **curly braces** to disambiguate the association.

if-else Statement

The following code needs no bracing for `if` and `else-if` statements.

```
#include <stdio.h>
int main(){
  int data;
  printf("Enter an integer: ");
  scanf("%d", &data);
  if (data<0) printf("-ve\n");
  else if (data == 0) printf("zero\n");
     else printf("+ve\n");
  return 0;}
```

if-else Statement

The following code needs bracing for `if` and `else-if` statements.

```
#include <stdio.h>
int main(){
  int data;
  printf("Enter an integer: ");
  scanf("%d", &data);
  if (data>0)
    if (data%5) printf("not divisible by 5\n");
  else printf("-ve data\n"); // incorrect association
  return 0;
}
```

if-else Statement

```
$ cc temp23.c
temp23.c:  In function ‘main’:
temp23.c:7:  warning:  suggest explicit
braces to avoid ambiguous ‘else’
$ ./a.out
Enter an integer:  -3
$ ./a.out
Enter an integer:  3
not divisible by 5
$ ./a.out
Enter an integer:  10
-ve data
```

if-else Statement


```
#include <stdio.h>
int main(){
    int data;
    printf("Enter an integer: ");
    scanf("%d", &data);
    if (data>0){
        if (data%5) printf("not divisible by 5\n");
    }
    else printf("-ve data\n");
    return 0;}
```

if-else Statement


```
$ cc temp23a.c
$ ./a.out
Enter an integer:  -3
-ve data
$ ./a.out
Enter an integer:  3
not divisible by 5
$ ./a.out
Enter an integer:  10
$
```

switch Statement (1)

`switch` Statement (2)

- C language uses `switch` statement to take *multi-way decision*.
- The decision is taken by matching the value of an expression to a value from a finite set of constants.
- Based on the decision, the *control of execution* is transfered.

`switch` Statement (3)

```
switch (expression) {
        case const-exp1: statement1
        case const-exp2: statement2
        . . .
        case const-expk: statementk
        default: statementk+1
        }
```

switch Statement (4)

Example Read a non-negative integer and take different actions depending on the remainders obtained by dividing the data by 5.

```
#include <stdio.h>
int main() { // switchNoBreak.c
  int data;
  printf("Enter a +ve integer: ");
  scanf("%d", &data);
  switch(data%5){
    case 0: printf("remainder = 0\n");
    case 1: printf("remainder = 1\n");
    case 2: printf("remainder = 2\n");
    case 3: printf("remainder = 3\n");
```

switch Statement


```
    default: printf("remainder = 4\n");
  }
  return 0;
}
```

```
$ cc switchNoBreak.c
$ ./a.out
Enter a +ve integer:  27
remainder = 2
remainder = 3
remainder = 4
```

The control is falling through. It is to be transfered out of the switch statement.

`switch` Statement (6)

Always use `break` statement to avoid the fall-through.

It forces the control out of the `switch` statement.

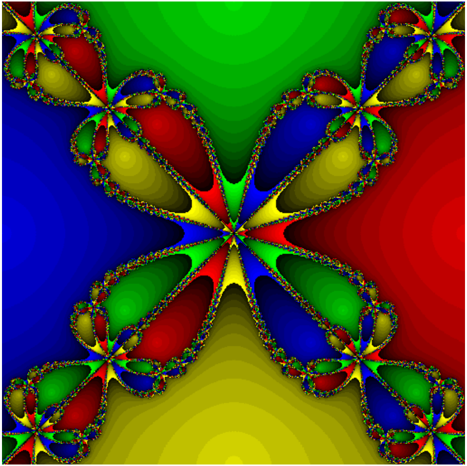
switch Statement


```
#include <stdio.h>
int main(){\\ switchBreak.c
  int data;
  printf("Enter a +ve integer: ");
  scanf("%d", &data);
  switch(data%5){
    case 0: printf("remainder 0\n"); break;
    case 1: printf("remainder 1\n"); break;
    case 2: printf("remainder 2\n"); break;
    case 3: printf("remainder 3\n"); break;
    default: printf("remainder 4\n");
  }
  return 0;}
```

switch Statement (8)

```
$ cc switchBreak.c
$ ./a.out
Enter a +ve integer:  27
remainder 2
```

Iteration in C



Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value.

Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Why Iteration

It is often necessary to execute a sequence of statements *repeatedly* to compute certain value. Every *imperative programming language* provides different constructs (statements) to perform this iterative computation.

Example problems:

1. Compute $n!$
2. Compute GCD(m, n)
3. Find the product of two or more matrices
4. Sort a list of integers in non-decreasing order

and so on, and so many!

Example

Write a program to compute the following sum:

$$S_n = 1 + 2 + 3 + \cdots + n,$$

where n is the input.

The best way to do it is to use the **closed form** or the **formula**:

$$S_n = \frac{n(n+1)}{2}.$$

Example

And we can write a program to do this.

```
#include <stdio.h>
int main(){
  int n;
  printf("Enter a +ve integer: ");
  scanf("%d", &n);
  printf("1+ ...+%d = %d\n",
          n, n*(n+1)/2);
  return 0;}
```

Example


```
$ cc -Wall temp26a.c
$ ./a.out
Enter a +ve integer:  5
1+ ...+5 = 15
```

Example

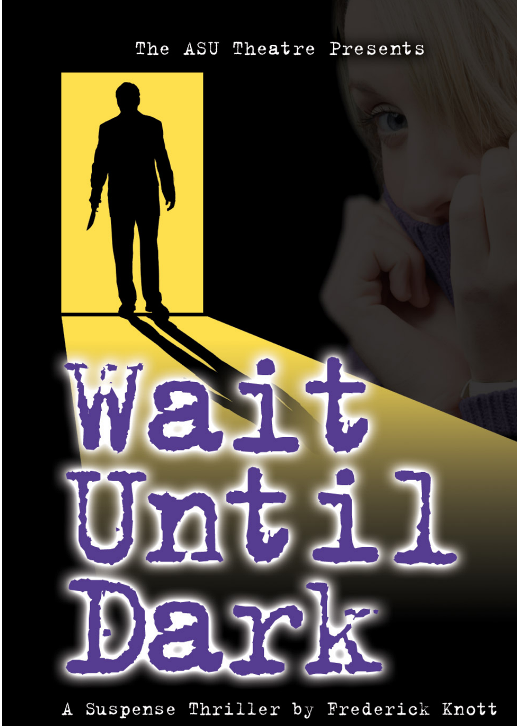
An alternate way using while loop

```
#include <stdio.h>
int main(){
    int n, sum = 0;
    printf("Enter a +ve integer: ");
    scanf("%d", &n);
    while(n > 0) {
        sum = n + sum;
        --n; }
    printf("sum: %d\n", sum);
    return 0;}
```

Example


```
$ cc -Wall temp26.c
$ ./a.out
Enter a +ve integer:  5
sum:  15
```

`while` Loop


The ASU Theatre Presents
Wait Until Dark
A Suspense Thriller by Frederick Knott

`while` Statement (1)

- `while` statement in C is one of the constructs used for iterative computation.
- The structure or syntax of `while` is:
  `while` (*expression*) {*statement*(*s*)}
- The `while` loop will not be entered if the loop-control *expression* evaluates to FALSE (zero) even before the first iteration.
- `break` statement can be used to come out of the `while` loop.

`while` Statement (2)

- The previous `while` program destroys the input data. That can be avoided by introducing a third variable where the value of `n` can be copied.
- To compute the following sum:

$$S_n^{(c)} = 1^c + 2^c + \cdots + n^c = \sum_{i=1}^{n} i^c,$$

where c is another input data (+ve int), we use **nested `while` loops**.

while Statement (3)

```
#include <stdio.h>
int main(){
  int n, c, sum = 0, m;
  printf("Enter the number of terms: ");
  scanf("%d", &n);
  printf("Enter the power: ");
  scanf("%d", &c);
  m = n; // save the input data
  while(n > 0){      // outer while
    int i=0, p=1;    // local to the block
    while(i++ < c) p *= n; // inner while
    sum += p; --n;} // end of outer while
  printf("sum = %d\n", sum);
  return 0;}
```

while Statement (4)

do-while Loop (1)

Useful for things that want to loop at least once. The structure is

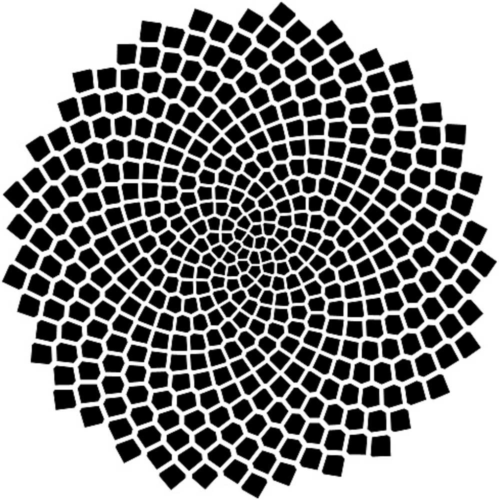
```
do {
  statement
} while(condition);
```

- The *condition* is tested at the end of the block instead of the beginning; so the statement block will be executed at least once.

do-while Loop (2)

- If the condition is true, it jumps back to the beginning of the block and executes the (simple or compound) statement again.
- `do-while` loop is essentially the same as `while` loop except that the loop body (statement block) is guaranteed to execute at least once.

`for` Loop



`for` Statement

It is an *iterative* construct in C language.
The structure or syntax of this statement is:

`for` (exp_1; exp_2; exp_3) { *statement* }

- exp_1: for initialization before entering the loop.
- exp_2: to decide whether to enter or continue the loop.

for Statement (2)

- *exp*$_3$: executed after execution of the *statement* part of the loop; it is used essentially to update the loop control condition.
- All three expressions can be omitted.
  If *exp*$_2$ is omitted, then the condition is TRUE.
  Ex: `for(i=1; ;i++)`
- A *statement* may be simple or compound, as well as null (`;`).
  Ex: `for(i=1;i>0;i++);`

Example on for Loop (1)

A program to compute the sum of first n natural numbers.

```
#include <stdio.h>
int main(){
  int n, i, sum = 0;
  printf("Enter a +ve integer: ");
  scanf("%d", &n);
  for(i=1, sum=0; i<=n; ++i)
      sum += i;
  printf("0+ ... + %d = %d\n", n, sum);
  return 0;}
```

Example on `for` Loop (2)

Here exp_1 is `i=1, sum=0`.
We have used the comma operator to join two simple statements.
The resultant compound statement is evaluated left to right and has the lowest precedence among the operators.

`while` and `for` Loops (1)

A `while` statement can be simulated by a `for` statement.

`while(`*exp*`)` *stmt* $\equiv$ `for(;` *exp*`;)` *stmt*

while and for Loops (2)

Similarly, a `for` statement can be simulated by a `while` statement and expression statements.

`for(`*exp1*`;` *exp2*`;` *exp3*`)` *stmt*
$\equiv$ *exp1*`;` `while(`*exp2*`)` {*stmt* *exp3*;}

Caution! This equivalence is not true if there is a `continue` statement in *stmt*.

Another example on `for` Loop (1)

Read n `int` data and print the largest among them.

The first input is the number of data, n, and subsequent inputs are a sequence of n `int` data.

Another example on `for` Loop (2)

Inductive Definition

$$\text{lrgst}(d_1, d_2, \cdots, d_n)$$

$$= \begin{cases} d_1 & \text{if } n = 1, \\ max(d_1, \text{lrgst}(d_2, \cdots, d_n)) & \text{if } n > 1. \end{cases}$$

Another example on `for` Loop (3)

```
#include <stdio.h>
int main(){
    int n, largest, i=1;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter %d data: ", n);
    scanf("%d", &largest);
    for(i=2; i<=n; ++i){
        int temp; //local to block
        scanf("%d", &temp);
        if (temp > largest) largest = temp;}
    printf("Largest: %d\n", largest);
    return 0;}
```

Another example on `for` Loop (4)

Special Termination

- It is not necessary to know the number of data *a priori*.
- We can use `EOF (end-of-file)` (defined in `stdio.h`) to terminate the input.
- Every call to `scanf()` returns the *number of data read*.
- If `Ctrl+D` is pressed from the keyboard, then `scanf()` returns `EOF`.

Another example on `for` Loop


```
#include <stdio.h>
int main() {
  int largest, count = 0, temp;
  printf("Enter integer data\n");
  printf("and terminate by Ctrl+D\n");
  scanf("%d", &largest); // at least one data
  ++count;
  for(; scanf("%d", &temp)!= EOF;){
      printf("%d ", temp);
      if (temp > largest) largest = temp;
      ++count;}
  printf("\nLargest among %d data: %d\n",
      count, largest);
  return 0;}
```