Programming & Data Structure

CS 11002

Partha Bhowmick

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

CSE Department
IIT Kharagpur

Spring 2012-2013

Pointers

Arrays and Pointers

A pointer is a variable that represents the *location* of a data item.

Example

`int *a;` means `a` is a pointer to integer.

See also earlier slides on similar declaration of pointers.

Arrays and Pointers

Why?

- To access a variable defined outside the function.
- To pass information back and forth between a function and its reference point.
- To efficiently navigate through data tables.
- To reduce the length and complexity of a program.
- Sometimes also increases the execution speed.

Arrays and Pointers


```
int i, a[10];
```

In `a[i]`, the array name `a` represents a pointer. The value of `a` is the *address* of the *0th* location, i.e.,

$$\mathtt{a} \equiv \&\mathtt{a}[0] \text{ and } *\mathtt{a} \equiv \mathtt{a}[0].$$

Arrays and Pointers (4)

In general, for $\texttt{i} = 0, 1, \ldots, 9$,

$$\texttt{a} + \texttt{i} \equiv \&\texttt{a}[\texttt{i}] \text{ and } *(\texttt{a} + \texttt{i}) \equiv \texttt{a}[\texttt{i}].$$

So,

$$\&(*(\texttt{a} + \texttt{i})) \equiv \&\texttt{a}[\texttt{i}] \equiv \texttt{a} + \texttt{i}$$

and

$$*(\&\texttt{a}[\texttt{i}]) \equiv *(\texttt{a} + \texttt{i}) \equiv \texttt{a}[\texttt{i}].$$

Note

The operators `*` and `&` are inverse to each other.

Pointer Arithmetic


```
#include <stdio.h>
int main(){
  char c = 'A';
  int i = 10;
  float f = 3.14;
  double d = 31.459;
  printf("%6c --- location %u \n", c,  (unsigned)&c);
  printf("%6d --- location %u \n", i,  (unsigned)&i);
  printf("%6.3f --- location %u \n", f,(unsigned)&f);
  printf("%6.3f --- location %u \n", d,(unsigned)&d);
  return 0;
} //address.c
```

Pointer Arithmetic (2)

Output

```
     A --- location 3218444239
    10 --- location 3218444232
 3.140 --- location 3218444228
31.459 --- location 3218444216
```

Pointer Arithmetic (3)

```
#include <stdio.h>
int main(){
  char *c, c1= 'A';
  int *i, i1 = 10;
  float *f, f1 = 3.14;
  double *d, d1 = 31.459;
  c = &c1; i = &i1; f = &f1; d = &d1;
  printf("%6c --- location %u \n", *c,   (unsigned)c);
  printf("%6d --- location %u \n", *i,   (unsigned)i);
  printf("%6.3f --- location %u \n", *f, (unsigned)f);
  printf("%6.3f --- location %u \n", *d, (unsigned)d);
  return 0;
} //pointer3.c
```

Pointer Arithmetic

Output

```
     A --- location 3216152175
    10 --- location 3216152168
 3.140 --- location 3216152164
31.459 --- location 3216152152
```

Pointer Arithmetic (5)

Note

- `char *c, c1= 'A';`
  `...`
  `c = &c1;` → is the *pointer initialization*.
- Pointer variables must point to a data of the *same type*.
  ```
  char *c, c1= 'A';
  int *i;
  i = &c1;
  ```
  → throws compilation warning and results to erroneous output.

Pointer Arithmetic

- Assigning an absolute address to a pointer variable is prohibited.
- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the *indirection operator* (`*`).

```
char *c, c1= 'A', c2;
c = &c2;
```

`c2 = *c;` → equivalent to `c2 = c1;`

Pointer Arithmetic

- Like other variables, *contents* of pointer variables can be used in *expressions*.

Example

```
int *p1, *p2, sum, prod;
...
sum = *p1 + *p2;
prod = *p1 * *p2;
prod = (*p1) * (*p2);
*p1 = *p1 + 2;
x = *p1 / *p2 + 5;
```

Pointer Arithmetic

- The following are allowed in C.
  - Add an integer to a pointer.

```
int i, a[5];
for (i=0; i<5; i++)
  printf("%u\n", (unsigned)(a+i));
```

Output:

```
3214950148
3214950152
3214950156
3214950160
3214950164
```

Pointer Arithmetic

- Subtract an integer from a pointer.
- Subtract one pointer from another of the same type.
  Note: If `p1` and `p2` are both pointers to the same array, then `p2-p1` gives the number of elements between `p1` and `p2`.

Pointer Arithmetic

- The following are *not allowed* in C.
  - Add two pointers.

    ```
    p1 = p1 + p2;
    ```
  - Multiply / divide a pointer in an expression.

    ```
    p1 = p2 / 5;
    p1 = p1 - p2 * 10;
    ```

Scale Factor

When an integer `i` is added to or subtracted from an integer pointer variable `pi`, the scale factor `sizeof(int)` times the value of `i` that actually gets added with `pi`.

Example

`pi = pi + 10;` implies `pi` increases by `sizeof(int)*10`.

Scale Factor (2)

```
#include <stdio.h>
int main(){
  int i=1, *pi; char c='A', *pc;
  pi = &i, pc = &c;
  printf("pi = %u, pc = %u \n",
         (unsigned)pi, (unsigned)pc);
  pi = pi+10; pc = pc+10;
  printf("pi = %u, pc = %u \n",
         (unsigned)pi, (unsigned)pc);
  return 0;} //pointer4.c
```

Output

```
pi = 3214711672, pc = 3214711679
pi = 3214711712, pc = 3214711689
```

Scale Factor

Data Type	Scale Factor
`char`	1
`int`	4
`float`	4
`double`	8

- To verify:
  `printf("#bytes for int = %d.\n", sizeof(int));`
- For a `struct newDataType`, the scale factor is given by `sizeof(newDataType)`.

Passing Pointers to Functions (1)

Q: Why are pointers passed to functions as arguments?

A: A pointer to a data item within the *caller function* allows the *callee function* to access and alter the data for further usage by the *caller function*.

Passing Pointers to Functions (2)

Example

Incorrect passing of arguments

```
void f(int i){i=10*i;}
int main(){
  int i=1;
  f(i);
  printf("i = %d\n", i);
  return 0;} //pointer5.c
```

Output: `i = 1`

Passing Pointers to Functions (3)

Example

Correct passing of arguments

```
void f(int *pi){*pi = 10*(*pi);}
int main(){
  int i=1;
  f(&i);
  printf("i = %d\n", i);
  return 0;} //pointer6.c
```

Output: `i = 10`

Passing Pointers to Functions (4)

Q: What is actually meant by passing a pointer to a function?

A: It means passing the value stored in the pointer variable, i.e., the address of the data that the pointer points to.

Q: Is it *pass by reference* or *pass by value*?

A: In C, *address of a data* is passed.
So, *pass by reference* means passing the address of the data.

Passing Pointers to Functions (5)

Example

`swap`—arguments passed by *value*—**wrong!**

```
main(){
   int  a=5, b=10;
   swap (a, b);
   printf ("\n a=%d, b=%d", a, b);}

void  swap (int x, int y){
   int  t;
   t = x; x = y; y = t;}
```

Output: `a=5, b=10`

Passing Pointers to Functions (6)

Example

`swap`—arguments passed by *reference*—**right**

```
main(){
   int  a=5, b=10;
   swap (&a, &b);
   printf ("\n a=%d, b=%d", a, b);}

void swap (int *x, int *y){
   int  t;
   t = *x; *x = *y; *y = t;}
```

Output: `a=10, b=5`

Passing Pointers to Functions (7)

Example

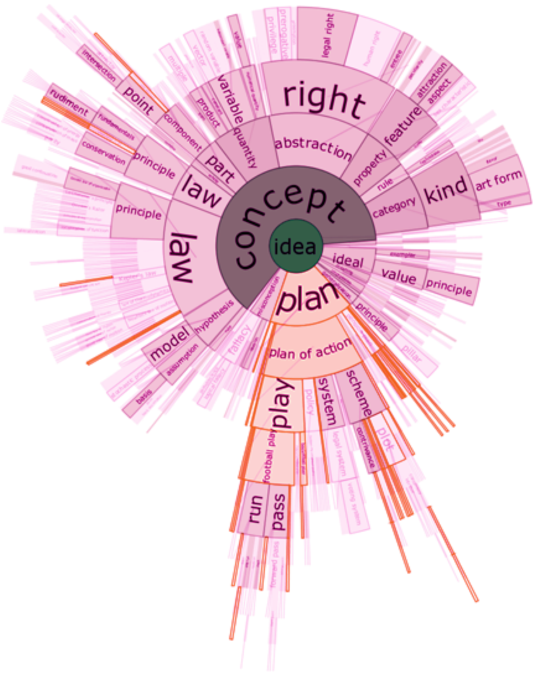
`scanf()` versus `printf()`

```
int x, y;
scanf ("%d %d", &x, &y);
...
printf ("%d %d %d", x, y, x+y);
```

Q: Why to use '`&`' in `scanf()` but not in `printf()`?

A: The function `printf()` needs only a value in order to output it. But `scanf()` stores a value, and hence the address of a place for its storage. The address is passed through the pointer.

Dynamic Memory Allocation



Dynamic Memory Allocation

Why?

Many a time we face situations where data is dynamic in nature due to following reasons.

- Amount of data cannot be predicted beforehand.
- Number of data items keeps changing during program execution.

Such situations can be handled more easily and effectively using *dynamic memory management techniques*.

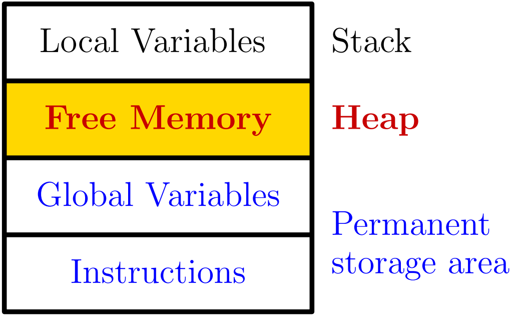
Dynamic Memory Allocation (2)

Why not Array?

C language requires the number of elements in an array to be specified at *compile time*. This often leads to *over-allocation* causing wastage of memory space or *under-allocation* resulting to program failure.

Through *dynamic memory allocation*, memory space required can be specified at *execution time*. C supports allocating and freeing memory dynamically using library routines.

Dynamic Memory Allocation



MEMORY

Dynamic allocation is done from the free memory space or *heap* during run time.

Dynamic Memory Allocation

Memory Allocation Functions

`malloc` Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.

```
int *p;
p = (int *)malloc(100 * sizeof(int));
```

`calloc` Allocates space for an array of elements, initializes them to zero, and then returns a pointer to the memory.

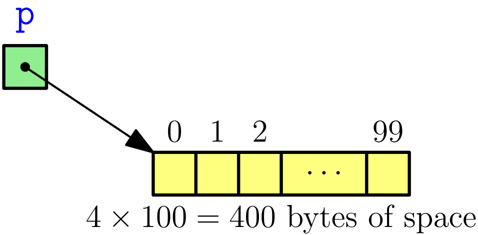
`free` Frees previously allocated space.

`realloc` Modifies the size of previously allocated space.

These are defined in `stdlib.h`. *Use* `man` *to see further details.*

Dynamic Memory Allocation


```
p = (int *) malloc(100 * sizeof(int));
```



A memory space of 100 times the size of `int` = 400 bytes is reserved. The address of the *first byte* of the allocated memory is assigned to the pointer `p`.

Dynamic Memory Allocation

`malloc()` always allocates a block of contiguous bytes.
The allocation can fail if sufficient contiguous memory space is not available. If it fails, then `malloc` returns `NULL`.

```
if ((p = (int *) malloc(100 * sizeof(int)))
     == NULL){
  printf ("\nMemory cannot be allocated");
  exit();
}
```

Dynamic Memory Allocation

Releasing the allocated space

When we no longer need the data stored in a block of memory, we should release the block for future use.

How? By using the free function: `free(p);`

Dynamic Memory Allocation

Ways of declaration of 2D arrays

```
#define MAXROW 4
#define MAXCOL 5
...
int A[MAXROW][MAXCOL];
int (*B)[MAXCOL];
int *C[MAXROW];
int **D;
```

Dynamic Memory Allocation

`int A[MAXROW][MAXCOL];`

⇒ `A` is a *statically allocated array*.

`int (*B)[MAXCOL];`

⇒ `B` is a *pointer* to an array of `MAXCOL` integers.

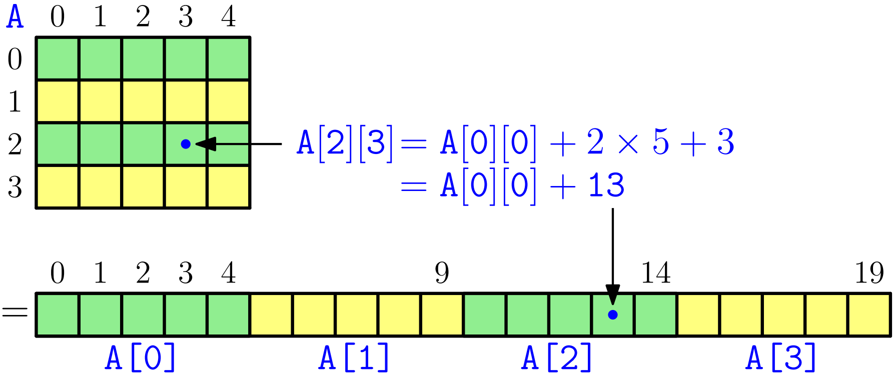
`int *C[MAXROW];`

⇒ `C` is an array of `MAXROW` `int` *pointers*.

`int **D;`

⇒ `D` is a *pointer to an* `int` *pointer*.

Dynamic Memory Allocation



$$\texttt{A[2][3]} = \texttt{A[0][0]} + 2 \times 5 + 3$$
$$= \texttt{A[0][0]} + \texttt{13}$$

Note: $\texttt{A[i][j]} \equiv \texttt{A[0][0]} + \texttt{MAXCOL} \times \texttt{i} + \texttt{j}$

Dynamic Memory Allocation

- All these are essentially different in terms of memory management.
- Except `A`, the three other arrays support *dynamic memory allocation*.
- When properly allocated memory, each of these can be used to represent a `MAXROW`-by-`MAXCOL` array.
- In all the four cases, the (i, j)th entry is accessed as `array_name[i][j]`.
- `A` and `B` are pointers to arrays, whereas `C` and `D` are arrays of pointers.

Dynamic Memory Allocation


```
int (*B)[MAXCOL];
```

`B` is a pointer to an array of `COLSIZE` integers. So it can be allocated `ROWSIZE` rows in the following way:

```
B = (int (*)[COLSIZE])malloc(ROWSIZE *
sizeof(int[COLSIZE]));
```

Dynamic Memory Allocation


```
int *C[MAXROW];
```

`C` is a static array of `ROWSIZE` int pointers.
Therefore, `C` itself cannot be allocated memory.
The individual rows of C should be allocated memory.

```
int i;
for (i=0; i<ROWSIZE; ++i)
  C[i] = (int *)malloc(COLSIZE * sizeof(int));
```

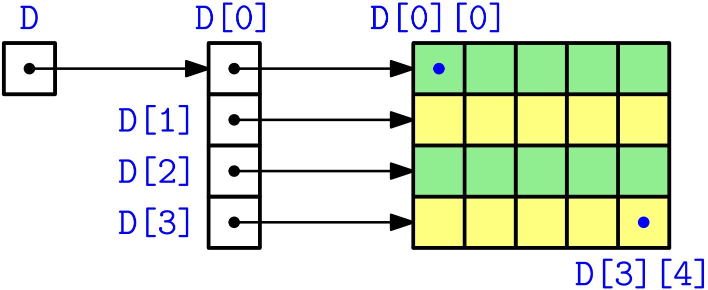
Dynamic Memory Allocation


```
int **D;
```

D is dynamic in both directions. First, it should be allocated memory to store ROWSIZE int pointers, each meant for a row of the 2D array. Each row pointer, in turn, should be allocated memory for COLSIZE int data.

```
int i;
D = (int **)malloc(ROWSIZE * sizeof(int *));
for (i=0; i<ROWSIZE; ++i)
  D[i] = (int *)malloc(COLSIZE * sizeof(int));
```

Dynamic Memory Allocation



Dynamic Memory Allocation


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

int **allocate (int h, int w){
  int **p, i;

  //p = (int **) malloc(h * sizeof (int *));
  p = (int **) calloc(h, sizeof (int *));
  for (i=0;i<h;i++)
    // p[i] = (int *) malloc(w * sizeof (int));
    p[i] = (int *) calloc(w,sizeof (int));
  return(p);
}
```

Dynamic Memory Allocation


```
void read_data (int **p, int h, int w){
  int i, j;
  for (i=0;i<h;i++)
    for (j=0;j<w;j++)
      scanf ("%d", &p[i][j]);
}
void print_data (int **p, int h, int w){
  int i, j;
  for (i=0;i<h;i++){
  for (j=0;j<w;j++)
    printf ("%5d ", p[i][j]);
   printf ("\n");}
}
```

Dynamic Memory Allocation


```
int main(){
  int **p, M, N;

  printf ("Give M and N: \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  read_data (p, M, N);
  printf ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
} //calloc.c
```

Dynamic Memory Allocation


```
$ cc -Wall calloc.c -o calloc.out
$ ./calloc.out
Give M and N:
3 4
Enter the elements:
1 2 3 4
5 6 7 10
11 21 31 41

The array read as
    1     2     3     4
    5     6     7    10
   11    21    31    41
```

Ex 1

How many bytes you require to represent the following structure?

```
struct RECORD {
  char name[80];
  int age;
  float height;
  char address[5][80];
} rec;
```

Ex 1

(2)

```
printf("rec size in bytes = %d\n",
        sizeof(rec.name)  +
        sizeof(rec.age)   +
        sizeof(rec.height)+
        sizeof(rec.address));
printf("rec size in bytes = %d\n",
        sizeof(rec));
```

```
rec size in bytes = 488
rec size in bytes = 488
```

Ex 2

Output?

```
char try[]="1234567890\n";
printf("strlen: %d, sizeof: %d\n",
        strlen(try),sizeof(try));
```

```
strlen: 11, sizeof: 12
```

Ex 2

Output?

```
char try[]="1234567890\n";
printf("strlen: %d, sizeof: %d\n",
         strlen(try),sizeof(try));
```

```
strlen:  11, sizeof:  12
```

Ex 3

Fill in the gap to compute the length of a string.

```
int length(char str[]){
  int i=0;
  while(____________________);
  return(___________________);
}
```

```
int length(char str[]){
  int i=0;
  while(str[i++]!='\0');
  return(--i);
}
```

Ex 3

Fill in the gap to compute the length of a string.

```
int length(char str[]){
  int i=0;
  while(_____________________);
  return(____________________);
}
```

```
int length(char str[]){
  int i=0;
  while(str[i++]!='\0');
  return(--i);
}
```