Data Structure and Algorithms

Amiya Ranjan Panda

Preliminaries

Topic to be covered...

- Structure
- Union
- Pointer
- Enumerated (enum)

Structure

Objectives:

- Be able to use compound data structures in programs
- Be able to pass compound data structures as function arguments, either by value or by reference
- Be able to do simple bit-vector manipulations

Structure

Contents:

Introduction to Structure

- A structure is
  - a convenient way of grouping several pieces of related information together
  - a collection of variables under a single name

- Problem: – How to group together a collection of data items of different types that are logically related to a particular entity??? (Array)
- Solution: Structure

- Structure:
  - different data types under a single name.
  - The variables are called members of the structure.
  - The structure helps to define a user-defined data type.

Why use structure?

- There are cases where multiple attribute values of an entity need to be stored as a singly unit.
- Challenges:
  - Not necessary that all the information of the entity are one type only.
  - It can have different attributes having different data types.
- Example: An entity *Student* may have its name (string), roll number (int), marks (float), etc.
- To store such type of information for the entity *student*, the following approaches need to be followed:
  - Approach-I: Construct individual arrays for storing names, roll numbers, and marks.
  - Approach-II: Use a special data structure to store the collection of different data types.

Approach-I

```
#include<stdio.h>
int main ()  {
        char name[5][10];
        int roll[5], i;
        float marks[5];
        for(i=0; i<5; i++) {
                printf("Enter the name, roll number, and marks of the student %d: ", i+1);
                scanf("%s %d %f", name[i], &roll[i], &marks[i]);
        }
        printf("Printing the Student details ...\n");
        for(i=0; i<5; i++)
                printf("%s %d %f\n",name[i], roll[i], marks[i]);
        return 0;
}
```

Approach-I

Output

```
Enter the name, roll number, and marks of the student 1: Arun 90 91
Enter the name, roll number, and marks of the student 2: Varun 91 56
Enter the name, roll number, and marks of the student 3: Shyam 89 69
Enter the name, roll number, and marks of the student 4: Amrit 52 63
Enter the name, roll number, and marks of the student 5: Bipin 45 73

Printing the Student details...
Arun 90 91.000000
Varun 91 56.000000
Shyam 89 69.000000
Amrit 52 63.000000
Bipin 45 73.000000
```

Approach-I

- This program may fulfill the requirement of storing the information of a student entity.
- But, the program is very complex, and the complexity increase with the amount of the input.
- The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory.

Approach-II

- structure is an user defined data type available in C that allows to combine data items of different kinds.

Approach-II

- Structures are used to represent a record.
- Example: books in a library. Tack the following attributes about each book:
  - Title
  - Author
  - Subject
  - Book ID

Defining a Structure

- use the struct statement.
  - The *struct* construct defines a new data type, with more than one member.
- The format of the struct statement is as follows:

```
struct [tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

- The tag is optional
- Each member definition is a normal variable definition, such as int i;.
- At the end of the structure's definition, before the final semicolon, can specify one or more structure variables but it is optional.

Defining a Structure

- Example:

```
struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
} book;
```

```
struct Student {
        char name[20];
         int roll_no;
        float marks;
        char gender;
        long int phone_no;
 }st1, st2, st3;
```

use of structure_name is optional

```
struct {
        char name[20];
         int roll_no;
        float marks;
        char gender;
        long int phone_no;
 }st1, st2, st3;
```

Accessing Structure Members

- To access any member of a structure, use the member access operator (.).
- The member access operator is coded as a period between the structure variable name and the structure member.
- Example:

```
#include <stdio.h>
#include <string.h>
struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
};
```

Accessing Structure Members

```
int main() {
        struct Books Book1;  /* Declare Book1 of type Book */
        struct Books Book2;  /* Declare Book2 of type Book */
        strcpy( Book1.title, "C Programming");
        strcpy( Book1.author, "Balguruswami");
        strcpy( Book1.subject, "C Programming Tutorial");
        Book1.book_id = 6495407;

        strcpy( Book2.title, "OOPs ");
        strcpy( Book2.author, "Balguruswami");
        strcpy( Book2.subject, "Object Oriented Programming");
        Book2.book_id = 6495700;
```

Accessing Structure Members

Output:

```
Book 1 title : C Programming
Book 1 author : Balguruswami
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : OOPs
Book 2 author : Balguruswami
Book 2 subject : Object Oriented Programming
Book 2 book_id : 6495700
```

Structure Initialization

- Syntax: struct structure_names tructure_variable={value1, value2, …, valueN};
- There is a one-to-one correspondence between the members and their initializing values.
- Note: C does not allow the initialization of individual structure members within the structure definition template.

```
struct student {
        char name[20];
        int roll_no;
        float marks;
        char gender;
        long int phone_no;
 };
```

```
void main() {
       struct student st1={“ABC", 4, 79.5, 'M', 5010670};
       printf("Name\tRoll No\tMarks\tGender\tPhone No.");
       printf("\n.............................................................\n");
       printf("\n %s\t %d\t %f\t%ct %ld", st1.name, st1.roll_no,
       st1.marks, st1.gender, st1.phone_no);
}
```

Partial Initialization

- Can initialize the first few members and leave the remaining blank.
- However, the uninitialized members should be only at the end of the list.
- The uninitialized members are assigned default values as follows: – Zero for integer and floating pointnumbers.

```
struct student {
        char name[20];
        int roll;
        char remarks;
        float marks;
 };
```

```
void main() {
        struct student s1={“name", 4};
        printf("Name=%s", s1.name);
        printf("\n Roll=%d", s1.roll);
         printf("\n Remarks=%c",s1.remarks);
        printf("\n Marks=%f", s1.marks);
}
```

Copy & Comparison of Structure

- Two variables of the same structure type can be copied in the same way as ordinary variables.
- If student1 and student2 belong to the same structure, then the following statements are valid: student1=student2; student2=student1;
- However, the statements such as: student1==student2 student1!=student2 are not permitted.
- If we need to compare the structure variables, we may do soby comparing members individually.

```
struct student {
      char name[20];
      int roll;
};
```

```
void main() {
      struct student student1={“ABC", 4,};
      struct student student2; student2=student1;
      printf("\nStudent2.name=%s", student2.name);
      printf("\nStudent2.roll=%d", student2.roll);
      if(strcmp(student1.name, student2.name)==0 &&
      (student1.roll==student2.roll)) { printf("\n\n student1 and
      student2 aresame."); }
}
```

How Structure members are stored?

- The elements of a structure are always stored in contiguous memory locations.
- A structure variable reserves number of bytes equal to sum of bytes needed to each of its members.

Array of Structure

- Consider a structure as: struct student { char name[20]; int roll; char remarks; float marks; };
- If we want to keep record of 100 students, we have to make 100 structure variables like st1, st2,…,st100.
- In this situation, array of structure can be used to store the records of 100 students which is easier and efficient to handle (because loops can be used).
- Two ways to declare an array of structure:

```
struct student {
        char name[20];
        int roll;
        char remarks;
        float marks;
 }st[100];
```

```
struct student {
        char name[20];
        int roll;
        char remarks;
        float marks;
};
struct student st[100];
```

Array of Structure

```
struct student {
        int roll_no;
        char f_name[20];
        char l_name[20];
  }st[10];
```

READING VALUES:

```
for(i=0; i<5; i++) {
      printf("\n Enter roll number:");
      scanf("%d", &s[i].roll_no);
      printf("\n Enter first name:");
      scanf("%s", s[i].f_name);
      printf("\n Enter last name:");
      scanf("%s", s[i].l_name);
}
```

WRITING VALUES:

```
for(i=0; i<5; i++) {
      printf("\n Roll number: ");
      printf("%d", s[i].roll_no);
      printf("\n First name: ");
      printf("%s", s[i].f_name);
      printf("\n Last name: ");
      printf("%s", s[i].l_name);
}
```

Structure within a Structure

- Let us consider a structure personal_record to store the information of a person as:

```
struct personal_record {
      char name[20];
      int day_of_birth;
      int month_of_birth;
      int year_of_birth;
      float salary;
}person;
```

Structure within a Structure

- In the structure above, we cangroup all the items related to birthday together and declare them under a substructure as:

```
struct Date {
        int day_of_birth;
        int month_of_birth;
        int year_of_birth;
};

struct personal_record {
        char name[20];
        struct Date birthday;
        float salary;
}person;
```

Structure within a Structure

- Here, the structure personal_record contains a member named birthday which itself is a structure with 3 members. This is called structure within structure.
- The members contained within the inner structure can be accessed as:

  person.birthday.day_of_birth,

  person.birthday.month_of_birth,

  person.birthday. year_of_birth
- The other members within the structure personal_record are accessed asusual:

  person.name,

  person.salary

Structure within a Structure

```
printf("Enter name:\t");
scanf("%s", person.name);
printf("\nEnter day of birthday:\t");
scanf("%d", &person.birthday.day_of_birth);
printf("\nEnter month of birthday:\t");
scanf("%d", &person.birthday.month_of_birth);
printf("\nEnter year of birthday:t");
scanf("%d", &person.birthday.year_of_birth);
printf("\nEnter salary:\t");
scanf("%f", &person.salary);
```

Structure within a Structure

```
struct Date {
      int day;
      int month;
      int year;
};

struct Name {
      char first_name[10];
      char middle_name[10];
      char last_name[10];
};

struct personal_record {
      float salary;
      struct Date birthday;
      struct Name full_name;
};
```

Pointers to Structure

- Astructure type pointer variable canbe declared as:

```
struct Book {
        char name[20];
        int pages;
        float price;
};
struct Book *bptr;
```

- However, this declaration for a pointer to structure does not allocate any memory for a structure.
  - but allocates only for a pointer, so that to access structure’s members through pointer bptr, we must allocate the memory using malloc()function.

Pointers to Structure

- Now, individual structure members are accessed as:

```
bptr->name or (*bptr).name
bptr->pages or (*bptr).pages
bptr->price  or (*bptr).price
```

- Here, -> is called arrow operator and there must be apointer to the structure on the left side of this operator.

```
struct book b, *bptr;
bptr=b;
printf("\n Enter name:\t");
scanf("%s", bptr->name);
printf("\n Enter no. of pages:\t");
scanf("%d", &bptr->pages);
printf("\n Enter price:\t");
scanf("%f", &bptr->price);
```

Pointers to Structure

- Now the members of the structure book can be accessed in 3 ways as:

b.name	bptr->name	(*bptr).name
b.pages	bptr->pages	(*bptr).pages
b.price	bptr-> price	(*bptr).price

Pointers to Array of Structure

- Consider a structure as follows:

```
struct book {
        char name[20];
        int pages;
        float price;
};
struct book b[10], *bptr;
```

- Then the assignment statement bptr=b; assigns the address of the 0th element of b to bptr.

Pointers to Array of Structure

- The members of b[0] can be accessed as:

```
bptr->name
bptr->pages
bptr->price
```

- Similarly members of b[1] can be accessed as:

```
(bptr+1)->name
(bptr+1)->pages
(bptr+1)->price
```

- The following for statement can be used to print all the values of array of structure b as:

```
for(bptr=b; bptr<b+10; bptr++)
        printf("%s %d %f", bptr->name, bptr->pages, bptr- >price);
```

Passing Structure to a Function

- Consider four cases here:
  - Passing the individual members to functions
  - Passing whole structure to functions
  - Passing structure pointer to functions
  - Passing array of structure to functions

Passing Structure to a Function

- Consider four cases here:
  - Passing the individual members to functions
  - Passing whole structure to functions
  - Passing structure pointer to functions
  - Passing array of structure to functions

Passing Structure member to a Function

- Structure members can be passed to functions like ordinary variables.
- Example: Let us consider a structure employee having members name, id and salary and pass these members to a function:

```
struct employee {
    char name[20];
    int id
    float salary;
};
```

- display(emp.name, emp.id, emp.salary);

```
 void display(char e[], int id , float sal) {
        printf("\nName \t\t Id \t\t Salary\n);
        printf("%s\t\t%d\t%.2f", e, id, sal);
  }
```

Problem: Huge number of structure members

Passing the whole Structure to a Function

- Whole structure can be passed to a function by the syntax:
- function_name(structure_variable_name);
- The called function has the form:

  return_type function_name(struct tag_name structure_variable_name)

  { … … … … …; }

- display(emp);

```
void display(struct employee e) {
        printf("\nName\tId\tSalary\n");
        printf("%s\t%d\t%.2f", e.name, e.id, e.salary);
}
```

Structures as Function Arguments

- A structure can be passed as a function argument in the same way as any other variable or pointers are passed.
- Example:

```
#include <stdio.h>
#include <string.h>
struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
};

/* function declaration */
void printBook( struct Books book );
```

Structures as Function Arguments

```
int main() {
    struct Books Book1;  /* Declare Book1 of type Book */
    struct Books Book2;  /* Declare Book2 of type Book */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Balguruswami");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    strcpy( Book2.title, "OOPs ");
    strcpy( Book2.author, "Balguruswami");
    strcpy( Book2.subject, "Object Oriented Programming");
    Book2.book_id = 6495700;

    printBook( Book1 );
    printBook( Book2 );
    return 0;
}
```

Structures as Function Arguments

```
void printBook( struct Books book ) {
	printf( "Book title : %s\n", book.title);
	printf( "Book author : %s\n", book.author);
	printf( "Book subject : %s\n", book.subject);
	printf( "Book book_id : %d\n", book.book_id);
}
```

Output:

```
	Book 1 title : C Programming
	Book 1 author : Balguruswami
	Book 1 subject : C Programming Tutorial
	Book 1 book_id : 6495407
	Book 2 title : OOPs
	Book 2 author : Balguruswami
	Book 2 subject : Object Oriented Programming
	Book 2 book_id : 6495700
```

Passing Structure Pointer to a Function

- In this case, address of structure variable is passed as an actual argument to a function.
- The corresponding formal argument must be a structure type pointer variable.
- Note: Any changes made to the members in the called function are directly reflected in the calling function.
- display(&emp);

```
void display(struct employee *e) {
      printf("\nName\tId\tSalary\n");
      printf("%s\t%d\t%.2f", e->name, e->id, e->salary);
 }
```

Pointers to Structures

- Define pointers to structures in the same way defining pointer to any other variable.
- Example:
  - struct Books *struct_pointer;
- Store the address of a structure variable in this pointer variable.

- To find the address of a structure variable, place the '&'; operator before the structure's name as follows:
  - struct_pointer = &Book1;

- To access the members of a structure using a pointer to that structure, use the → operator as follows:
  - struct_pointer->title;

Pointers to Structures

- Example:

```
#include <stdio.h>
#include <string.h>
struct Books {
        char  title[50];
        char  author[50];
        char  subject[100];
        int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
```

Pointers to Structures

```
int main() {
        struct Books Book1;  /* Declare Book1 of type Book */
        struct Books Book2;  /* Declare Book2 of type Book */
        strcpy( Book1.title, "C Programming");
        strcpy( Book1.author, "Balguruswami");
        strcpy( Book1.subject, "C Programming Tutorial");
        Book1.book_id = 6495407;

        strcpy( Book2.title, "OOPs ");
        strcpy( Book2.author, "Balguruswami");
        strcpy( Book2.subject, "Object Oriented Programming");
        Book2.book_id = 6495700;

        printBook( &Book1 );
        printBook( &Book2 );
        return 0;
}
```

Pointers to Structures

```
void printBook( struct Books *book ) {
	printf( "Book title : %s\n", book->title);
	printf( "Book author : %s\n", book->author);
	printf( "Book subject : %s\n", book->subject);
	printf( "Book book_id : %d\n", book->book_id);
}
```

Output:

```
	Book 1 title : C Programming
	Book 1 author : Balguruswami
	Book 1 subject : C Programming Tutorial
	Book 1 book_id : 6495407
	Book 2 title : OOPs
	Book 2 author : Balguruswami
	Book 2 subject : Object Oriented Programming
	Book 2 book_id : 6495700
```

Passing an Array of Structure Type to a Function

- Passing an array of structure type to a function is similar to passing an array of any type to a function.
- That is, the name of the array of structure is passed by the calling function which is the base address of the array of structure.
- Note: The function prototype comes after the structure definition
- .
- Passing Array of structures to function display(emp);

```
void display(struct employee ee[]) {
      int i;
      printf("\n Name\t\t Id\t Salary\n");
      for(i=0; i<5; i++) {
         printf("%s\t\t%d\t\t%.2f\n", ee[i].name, ee[i].id, ee[i].salary);
 } }
```

typedef Statement

- User Defined Data Types: The C language provides a facility called typedef for creating synonyms for previously defined data type names.
- For example, the declaration:
  - typedef int Length;
  - makes the name Length a synonym (or alias) for the data type int.
- The data "*type*" name Length can now be used in declarations in exactly the same way that the data type int can be used:
  - Length a, b, len ;
  - Length numbers[10] ;

Bit Fields

- C allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow.
- These space-saving structure members are called bit fields, and
  - their width in bits can be explicitly declared.
- A bit field is interpreted as an positive integral type.

Bit Fields

- Bit Fields allow the packing of data in a structure.

```
struct {
    unsigned int f1;
    unsigned int f2;
} status;
```

- This structure requires 8 bytes of memory space.
- If either 0 or 1 in each of the variables need to be stored, then C programming language offers a better way to utilize the memory space.

```
struct {
  unsigned int f1 : 1;
  unsigned int f2 : 1;
} status;
```

- The above structure requires 4 bytes of memory space for variable *status*, but only 2 bits will be used to store the values.

Bit Fields

- If 32 variables each one with a width of 1 bit are used, then also the status structure will use 4 bytes.
- However as soon as there will be 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes.
- Example:

```
#include <stdio.h>
#include <string.h>
struct {
        unsigned int f1;
        unsigned int f2;
} status1;

struct {
        unsigned int f1 : 1;
        unsigned int f2 : 1;
} status2;
```

Bit Fields

```
int main( ) {
        printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
        printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
        return 0;
}
```

- Output

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

- The declaration of a bit-field has the following form inside a structure:

```
struct {
      type [member_name] : width ;
};
```

- type: An integer type that determines how a bit-field's value is interpreted. The type may be *int*, *signed int*, or *unsigned int*.
- member_name: The name of the bit-field.
- width: The number of bits in the bit-field.

Bit Fields

- Example:

```
#include <stdio.h>
#include <string.h>
struct {
        unsigned int age : 3;
} Age;

int main( ) {
        Age.age = 4;
        printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
        printf( "Age.age : %d\n", Age.age );
        Age.age = 7;
        printf( "Age.age : %d\n", Age.age );
        Age.age = 8;
        printf( "Age.age : %d\n", Age.age );
        return 0;
}
```

Output:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Union

- A union is a special data type available in C that allows to store different data types in the same memory location.
- Union can be defined with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

- Use union statement in the same as it is used in defining a structure.
- The union statement defines a new data type with more than one member for the program.
- The syntax of the union statement is as follows:

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

- The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the union's definition, before the final semicolon, one or more union variables can be specified but it is optional.

Union

Example:

```
union Data {
        int i;
        float f;
        char str[20];
} data;
```

- A variable of this union data type can store an integer, a floating-point number, or a string of characters.
- It means a single variable, i.e., same memory location, can be used to store multiple types of data.
- The memory occupied by a union will be large enough to hold the largest member of the union.
- For example, in this union declaration, a variable of this type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

Union: Program

```
#include <stdio.h>
#include <string.h>
union Data {
        int i;
        float f;
        char str[20];
};

int main( ) {
      union Data data;
      printf( "Memory size occupied by data : %d\n", sizeof(data));
      return 0;
}
```

Output:

```
Memory size occupied by data : 20
```

Accessing Union Members

- To access any member of a union, the member access operator (.) is used.
- The member access operator is coded as a period between the union variable name and the union member that need to be accessed.

Union: Program

```
#include <stdio.h>
#include <string.h>
union Data {
        int i;
        float f;
        char str[20];
};
int main( ) {
        union Data data;
        data.i = 10;
        data.f = 220.5;
        strcpy(data.str, "C Programming");
        printf("data.i : %d\n", data.i);
        printf("data.f : %f\n", data.f);
        printf("data.str : %s\n", data.str);
        return 0;
}
```

Output:

```
data.i : 1917853763
data.f :
4122360580327794860452759994368.000000
data.str : C Programming
```

- The values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Union: Program

```
#include <stdio.h>
#include <string.h>
union Data {
        int i;
        float f;
        char str[20];
};
int main( ) {
        union Data data;
        data.i = 10;
        printf( "data.i : %d\n", data.i);
        data.f = 220.5;
        printf( "data.f : %f\n", data.f);
        strcpy( data.str, "C Programming");
        printf( "data.str : %s\n", data.str);
        return 0;
}
```

Output:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Pointers

Pointers: Introduction

- A pointer is a variable that represents the location (rather than the value) of a data item.
- They have a number of useful applications.
  - Enables us to access a variable/ data item that is defined outside the function.
  - Can be used to pass information back and forth between functions.
  - More efficient in handling <u>data tables</u>.
  - Reduces the length and complexity of a program.
  - Sometimes <u>also increases the execution speed</u>.

Pointers: Basic Concept

- Within the computer memory, every stored data item occupies one or more contiguous memory cells/ bytes.
  - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).
- Whenever a variable is declared, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, the <u>first location is considered as address</u>.

Contd.

- Consider the statement

```
int   xyz = 50;
```

  - This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.
  - Suppose that the address location chosen is 1001.

xyz	→	variable
50	→	value
1001	→	address

Contd.

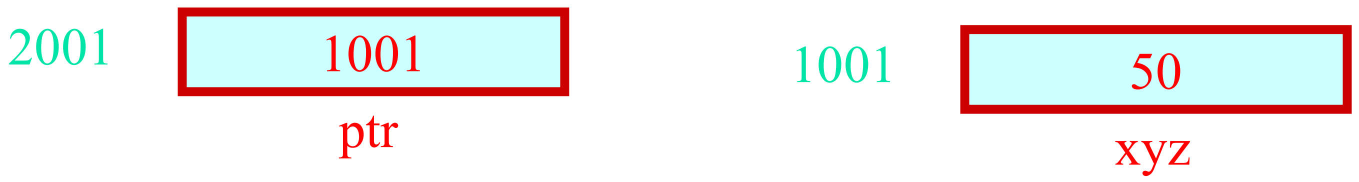
- During execution of the program, the system always associates the name xyz with the address 1001.
  - The value 50 can be accessed by using either the name xyz or the address 1001.
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
  - Such variables that hold memory addresses are called pointers.
  - Since a pointer is a variable, its value is also stored in some memory location.

Contd.

- Suppose we assign the address of xyz to a variable ptr.
  - ptr is said to point to the variable xyz.

Variable	Value	Address
xyz	50	1001
ptr	1001	2001

```
ptr = &xyz;
```



Accessing the Address of a Variable

- The address of a variable can be determined using the '&' operator.
  - The operator '&' immediately preceding a variable returns the address of the variable.
- Example:

```
ptr = &xyz;
```

  - The address of xyz (1001) is assigned to ptr.
- The '&' operator can be used only with a simple variable or an array element.

```
&distance
&x[0]
&x[i-2]
```

Contd.

- Following usages are illegal:

  &235

  - Pointing at constant.

  &(a+b)

  - Pointing at expression.

Example

```
#include  <stdio.h>
int main() {
    int   a;
    float  b, c;
    double  d;
    char  ch;

    a = 10;   b = 2.5;  c = 12.36;  d = 12345.66;  ch = ‘A’;
    printf  (“%d is stored in location %u \n”,  a,  &a) ;
    printf  (“%f is stored in location %u \n”,  b,  &b) ;
    printf  (“%f is stored in location %u \n”,  c,  &c) ;
    printf  (“%ld is stored in location %u \n”,  d,  &d) ;
    printf  (“%c is stored in location %u \n”,  ch,  &ch) ;
    return 1;
}
```

Output:

```
10 is stored in location 3221224908           a
2.500000 is stored in location 3221224904     b
12.360000 is stored in location 3221224900    c
12345.660000 is stored in location 3221224892 d
A is stored in location 3221224891            ch
```

Incidentally variables a,b,c,d and ch are allocated
to contiguous memory locations.

Pointer Declarations

- Pointer variables must be declared before use them.
- General form:

```
data_type  *pointer_name;
```

Three things are specified in the above declaration:

1. The asterisk (*) tells that the variable pointer_name is a pointer variable.
2. pointer_name needs a memory location.
3. pointer_name points to a variable of type data_type.

Contd.

- Example:

```
int     *count;
float  *speed;
```

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```
int      *ptr,  xyz;
:
ptr = &xyz;
```

  - This is called pointer initialization.

Things to Remember

- Pointer variables must always point to a data item of the *same type*.

```
float   x;
int     *ptr;
                    ➔  will result in erroneous output
ptr = &x;
```

- Assigning an absolute address to a pointer variable is prohibited.

```
int   *ptr;
:
ptr = 1005;
```

Accessing a Variable Through its Pointer

- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the indirection operator (*).

```
int   a, b;
int   *ptr;
:
ptr = &a;
b = *ptr;
```

Equivalent to

b = a

Example

```
#include  <stdio.h>
int main()
{
    int   a, b;
    int   c = 5;
    int   *p;

    a  =  4  *  (c  +  5) ;

    p  =  &c;
    b  =  4  *  (*p  +  5) ;
    printf  ("a=%d  b=%d \n",  a, b) ;
    return 0;
}
```

Equivalent

Example

```
#include  <stdio.h>
int main()  {
    int  x, y;
    int  *ptr;

    x = 10 ;
    ptr = &x ;
    y = *ptr ;
    printf  ("%d is stored in location %u \n",  x,  &x) ;
    printf  ("%d is stored in location %u \n",  *&x,  &x) ;
    printf  ("%d is stored in location %u \n",  *ptr,  ptr) ;
    printf  ("%d is stored in location %u \n",  y,  &*ptr) ;
    printf  ("%u is stored in location %u \n",  ptr, &ptr) ;
    printf  ("%d is stored in location %u \n",  y,  &y) ;

   *ptr = 25;
    printf  ("\nNow x = %d \n", x);
    return 0;
}
```

*&x ⇔ x

ptr=&x;
&x ⇔ &*ptr

Output:

```
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location
3221224900
10 is stored in location 3221224904

Now x = 25
```

Address of x: 3221224908

Address of y: 3221224904

Address of ptr: 3221224900

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

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

Contd.

- What are allowed in C?
  - Add an integer to a pointer.
  - Subtract an integer from a pointer.
  - Subtract one pointer from another (related).
    - If p1 and p2 are both pointers to the same array, them p2–p1 gives the number of elements between p1 and p2.
- What are not allowed?
  - Add two pointers.

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

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

Scale Factor

- It has been seen that an integer value can be added to or subtracted from a pointer variable.

```
int   *p1,  *p2 ;
int   i,  j;
:
p1  =  p1  +  1 ;
p2  =  p1  +  j ;
p2++ ;
p2  =  p2  –  (i + j) ;
```

- In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.

Contd.

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

- If p1 is an integer pointer, then

  p1++

  will increment the value of p1 by 4.

Example: to find the scale factors

Returns no. of bytes required for data type representation

```
#include  <stdio.h>
int main()  {
    printf ("Number of bytes occupied by int is %d \n", sizeof(int));
    printf ("Number of bytes occupied by float is %d \n", sizeof(float));
    printf ("Number of bytes occupied by double is %d \n", sizeof(double));
    printf ("Number of bytes occupied by char is %d \n", sizeof(char));
    return 0;
}
```

Output:

```
Number of bytes occupied by int is  4
Number of bytes occupied by float is  4
Number of bytes occupied by double is  8
Number of bytes occupied by char is  1
```

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called call-by-reference (or by address or by location).
- Normally, arguments are passed to a function by value.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

Example: passing arguments by value

```
#include  <stdio.h>
int main()  {
      int  a, b;
      a = 5 ;   b = 20 ;
      swap (a, b) ;
      printf  ("\n a = %d,  b = %d", a, b);
      return 0;
}

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

a and b
do not
swap

x and y swap

Output

a = 5, b = 20

Example: passing arguments by reference

```
#include  <stdio.h>
int main()  {
      int  a, b;
      a = 5 ;   b = 20 ;
      swap (&a, &b) ;
      printf  ("\n a = %d,  b = %d", a, b);
      return 0;
}

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

*(&a) and *(&b) swap

*x and *y swap

Output

a = 20, b = 5

scanf Revisited

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

- What about scanf ?

```
scanf("%d %d %d", &x, &y, x+y) ;                NO

scanf("%d %d", &x, &y) ;                        YES
```

Example: Sort 3 integers

- Three-step algorithm:
  1. Read in three integers x, y and z
  2. Put smallest in x
     - Swap x, y if necessary; then swap x, z if necessary.
  3. Put second smallest in y
     - Swap y, z if necessary.

Contd.

```
#include  <stdio.h>
int main() {
      int  x, y, z ;
      ………..
      scanf  ("%d %d %d", &x, &y, &z) ;
      if  (x > y)   swap (&x, &y);
      if  (x > z)   swap (&x, &z);
      if  (y > z)   swap (&y, &z) ;
      ………..
      return 0;
}
```

sort3 as a function

```
#include  <stdio.h>
int main()  {
      int  x, y, z ;
      ………..
      scanf  (“%d %d %d”, &x, &y, &z) ;
      sort3  (&x, &y, &z) ;
      ………..
      return 0;
}

void   sort3  (int *xp,  int *yp,  int *zp) {
      if  (*xp > *yp)   swap (xp, yp);
      if  (*xp > *zp)   swap (xp, zp);
      if  (*yp > *zp)   swap (yp, zp);
}
```

xp/yp/zp
are
pointers

Contd.

- Why no ‘&’ in swap call?
  - Because xp, yp and zp are already pointers that point to the variables that we want to swap.

Pointers and Arrays

- When an array is declared,
  - The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
  - The base address is the location of the first element (index 0) of the array.
  - The compiler also defines the array name as a constant pointer to the first element.

Example

- Consider the declaration:

```
int  x[5]  =  {1, 2, 3, 4, 5} ;
```

  - Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Contd.

x ⇔ &x[0] ⇔ 2500 ;

- p = x; and p = &x[0]; are equivalent.
- Access successive values of x by using p++ or p- - to move from one element to another.

- Relationship between p and x:

p	=	&x[0]	=	2500
p+1	=	&x[1]	=	2504
p+2	=	&x[2]	=	2508
p+3	=	&x[3]	=	2512
p+4	=	&x[4]	=	2516

*(p+i) gives the value of x[i]

Example: function to find average

int *array

```
#include  <stdio.h>
int main()  {
     int  x[50], k, n ;
     scanf  ("%d", &n) ;
     for  (k=0; k<n; k++)
          scanf  ("%d", &x[k]) ;
     printf  ("\nAverage is %f",
                              avg (x, n));
}
```

```
float  avg  (int array[ ], int  size) {
     int  *p, i , sum = 0;
     p = array ;
     for  (i=0; i<size; i++)
          sum = sum + *(p+i);
     return  ((float) sum / size);
}
```

p[i]

Structures Revisited

- Recall that a structure can be declared as:

```
struct   stud   {
                        int    roll;
                        char  dept_code[25];
                        float  cgpa;
                    };
struct  stud  a, b, c;
```

- And the individual structure elements can be accessed as:

```
a.roll ,  b.roll ,  c.cgpa , etc.
```

Arrays of Structures

- We can define an array of structure records as

```
struct   stud   class[100] ;
```

- The structure elements of the individual records can be accessed as:

```
class[i].roll
class[20].dept_code
class[k++].cgpa
```

Example: Sorting by Roll Numbers

```
#include <stdio.h>
struct  stud  {
     int   roll;
     char  dept_code[25];
     float  cgpa;
 };

int main() {
     struc  stud  class[100], t;
     int  j, k, n;
     scanf  (“%d”, &n);
                    /* no. of students */
```

```
for  (k=0; k<n; k++)
       scanf (“%d %s %f”, &class[k].roll,
            class[k].dept_code, &class[k].cgpa);
    for  (j=0; j<n-1; j++)
       for  (k=j+1; k<n; k++)  {
            if  (class[j].roll > class[k].roll)  {
                  t = class[j] ;
                  class[j] = class[k] ;
                  class[k] = t
            }
       }
    <<<< PRINT THE RECORDS >>>>
    return 0;
}
```

Pointers and Structures

- May recall the name of an array for the address of its zero-th element.
  - Also true for the names of arrays of structure variables.
- Consider the declaration:

```
struct   stud   {
                    int  roll;
                    char  dept_code[25];
                    float  cgpa;
                 }    class[100],  *ptr ;
```

  - The name class represents the address of the zero-th element of the structure array.
  - ptr is a pointer to data objects of the type struct stud.

- The assignment

```
ptr  =  class ;
```

  will assign the address of class[0] to ptr.

- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(stud).
  - It is made to point to the next record.

- Once ptr points to a structure variable, the members can be accessed as:

  ```
  ptr–>roll ;
  ptr–>dept_code ;
  ptr–>cgpa ;
  ```

  - The symbol "–>" is called the arrow operator.

Example

```
#include <stdio.h>
typedef struct {
                float real;
                float imag;
          }Complex;
```

```
swap(Complex *a, Complex *b)  {
   Complex tmp;
   tmp=*a;
   *a=*b;
   *b=tmp;
}
```

```
print(Complex *a) {
  printf("(%f, %f)\n", a->real, a->imag);
}
```

```
int main()  {
  Complex x={10.0, 3.0}, y={-20.0, 4.0};
  print(&x); print(&y);
  swap(&x, &y);
  print(&x); print(&y);
  return 0;
}
```

```
(10.000000, 3.000000)
(-20.000000, 4.000000)
(-20.000000, 4.000000)
(10.000000, 3.000000)
```

A Warning

- When using structure pointers, we should take care of operator precedence.
  - Member operator "." has higher precedence than "*".
    - ptr –> roll and (*ptr).roll mean the same thing.
    - *ptr.roll will lead to error.

  - The operator "–>" enjoys the highest priority among operators.
    - ++ptr –> roll will increment roll, not ptr.
    - (++ptr) –> roll will do the intended thing.

Structures and Functions

- A structure can be passed as argument to a function.
- A function can also return a structure.
- The process shall be illustrated with the help of an example.
    - A function to add two complex numbers.

Example: complex number addition

```
#include  <stdio.h>
struct  complex  {
                    float  re;
                    float  im;
                  };

int main()  {
   struct  complex  a, b, c;
   scanf  ("%f %f", &a.re, &a.im);
   scanf  ("%f %f", &b.re, &b.im);
   c  =  add (a, b) ;
   printf  ("\n %f %f", c.re, c.im);
   return 0;
}
```

```
struct  complex  add (struct complex
x, struct complex  y)  {
    struct  complex  t;
    t.re = x.re + y.re ;
    t.im = x.im + y.im ;
    return (t) ;
}
```

Example: Alternative way using pointers

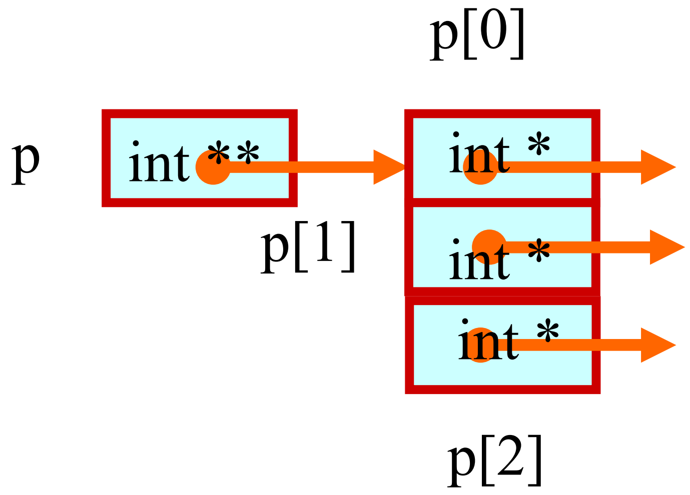
```
#include  <stdio.h>
struct  complex  {
                                float  re;
                                float  im;
                             };

int main()  {
    struct  complex  a, b, c;
    scanf  (“%f %f”, &a.re, &a.im);
    scanf  (“%f %f”, &b.re, &b.im);
    add (&a, &b, &c) ;
    printf  (“\n %f %f”, c.re, c.im);
    return 0;
}
```

```
void  add  (struct complex  *x, struct complex
*y, struct complex  *t)  {
     t->re = x->re + y->re ;
     t->im = x->im + y->im ;
}
```

Pointer to Pointer

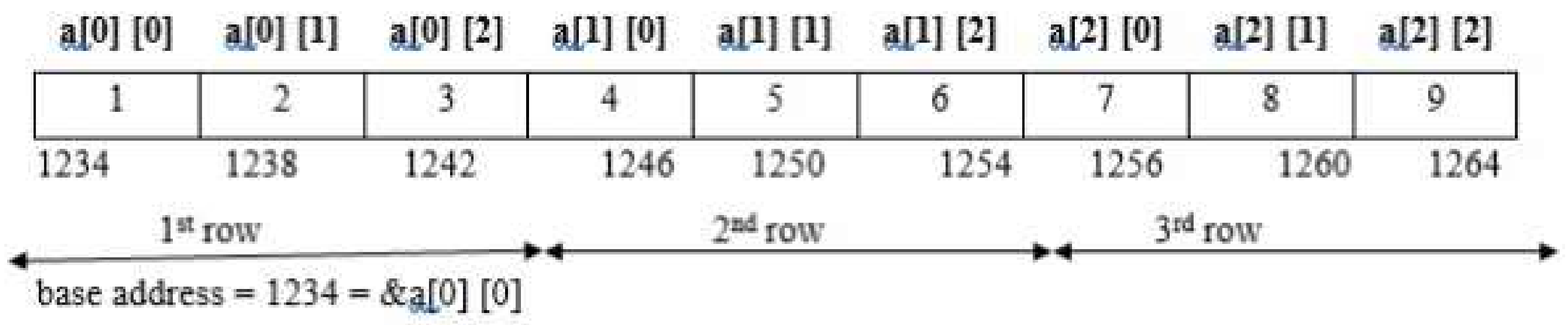
- Example:

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
```



Pointers and 2-D arrays

- Memory allocation for a two-dimensional array is as follows:
  - int a[3] [3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};



```
int *p;
p = &a[0] [0];
(or) p =a
```

$\longrightarrow$ Assigning base address to a pointer

❖ Pointer is used to access the elements of 2 – dimensional array as follows

```
a[i] [j] = *(p+i*columnsize+j)
```

Pointers and 2-D arrays

- Memory allocation for a two-dimensional array is as follows:
  - int a[3] [3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

```
a[1][2] = *(1234 + (1*3+2)*4)
        = *(1234 + (3+2)*4)
        = *(1234 + 5*4) // 4 is Scale factor
        = *(1234+20)
        = *(1254)
a[1][2] = 6
```

Pointers and 2-D arrays

```
#include<stdio.h>
int main()  {
            int a[3][3], i, j, *p;
            printf ("Enter elements of 2-D array");
            for (i=0; i<3; i++)
                        for (j=0; j<3; j++)
                                    scanf ("%d", &a[i] [j]);
            p = &a[0][0];
            printf ("Elements of 2-D array are...\n");
            for (i=0; i<3; i++){
                        for (j=0; j<3; j++)
                                    printf ("%d \t", *(p+i*3+j));
                        printf ("\n");
            }
            return 0;
}
```

Output

```
Enter elements of 2-D array
1 2 3 4 5 6 7 8 9
Elements of 2-D array are...
1 2 3
4 5 6
7 8 9
```

Pointers and 2-D arrays

- int (*p)[10];
  - Here p is a pointer that can point to an array of 10 integers.
  - In this case, the type of p is a pointer to an array of 10 integers.

- Note that parentheses around p are necessary:
- int *p[10];
  - here p is an array of 10 integer pointers.

- A pointer that points to the 0th element of an array and a pointer that points to the whole array are totally different.

Pointers and 2-D arrays

```
#include<stdio.h>
int main() {
    int *p;
    int (*parr)[5]; // pointer to an array of 5 integers
    int my_arr[5];
    p = my_arr;
    parr = my_arr;
    printf("Address of p = %u\n", p);
    printf("Address of parr = %u\n", parr );
    p++;
    parr++;
    printf("\nAfter incrementing p and parr by 1 \n\n");
    printf("Address of p = %u\n", p );
    printf("Address of parr = %u\n", parr );
    printf("Address of parr = %u\n", *parr );
    return 0;
}
```

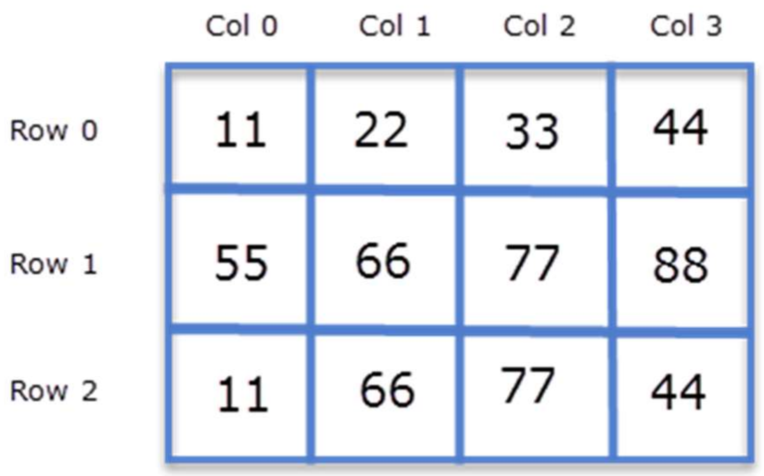
Output

```
Address of p = 2293296
Address of parr = 2293296

After incrementing p and parr
by 1

Address of p = 2293300
Address of parr = 2293316
```

Pointers and 2-D arrays

```
int arr[3][4] = {
                {11, 22, 33, 44},
                {55, 66, 77, 88},
                {11, 66, 77, 44}
        };
```

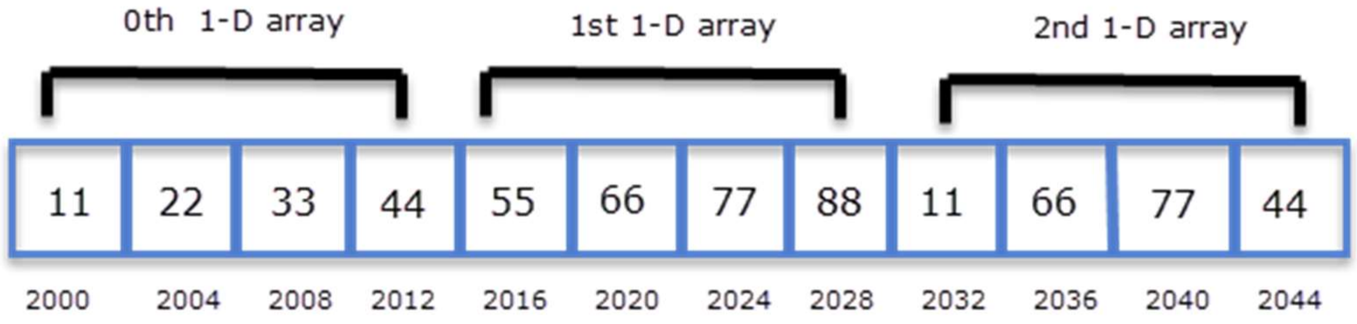
The 2-D array can be visualized as following:



	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

Pointers and 2-D arrays

- This shows how a 2-D array is stored in the memory:



- A 2-D array is actually a 1-D array in which each element is itself a 1-D array.
- So *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

Pointers and 2-D arrays

- We know that the name of a 1-D array is a constant pointer to the 0th element.
- In the case, of a 2-D array, 0th element is a 1-D array.
  - Hence the type or base type of arr is a pointer to an array of 4 integers.
- Since pointer arithmetic is performed relative to the base size of the pointer.
  - In the case of arr, if arr points to address 2000 then arr + 1 points to address 2016 (i.e 2000 + 4*4).

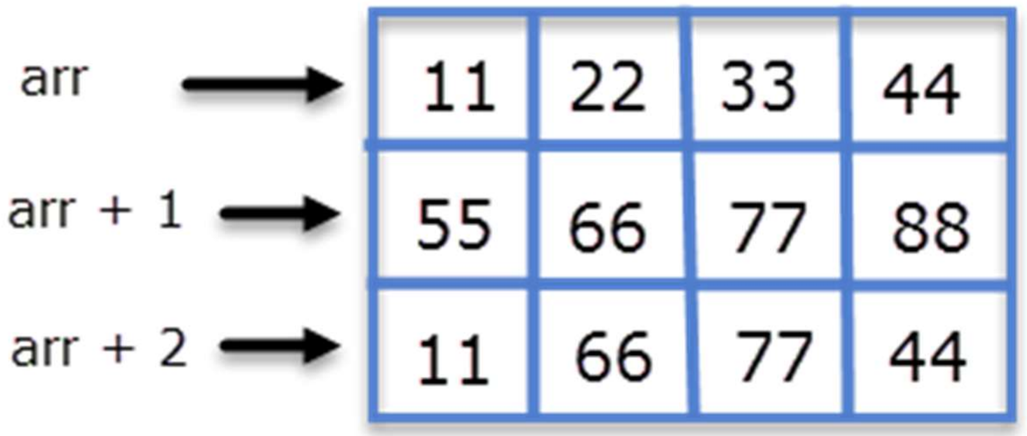
Pointers and 2-D arrays

- We know that the name of the array is a constant pointer that points to the 0th element of the array.
- In the case of a 2-D array, 0th element is a 1-D array.
  - So the name of the array in case of a 2-D array represents a pointer to the 0th 1-D array.
- Therefore in this case *arr* is a pointer to an array of 4 elements.
  - If the address of the 0th 1-D is 2000, then according to pointer arithmetic (arr + 1) will represent the address 2016,
  - similarly (arr + 2) will represent the address 2032.

Pointers and 2-D arrays

we can conclude that:

- arr points to 0th 1-D array.
- (arr + 1) points to 1st 1-D array.
- (arr + 2) points to 2nd 1-D array.



Pointers and 2-D arrays

In general

- (arr + i) points to ith 1-D array.
    - As we know that dereferencing a pointer to an array gives the base address of the array.
    - So dereferencing arr we will get *arr,
        - base type of *arr is (int*).
    - Similarly, on dereferencing arr+1 we will get *(arr+1).

- We can say that
    - *(arr+i) points to the base address of the ith 1-D array.

- Again it is important to note that type (arr + i) and *(arr+i) points to same address but their base types are completely different.
    - The base type of (arr + i) is a pointer to an array of 4 integers, while the base type of *(arr + i) is a pointer to int or (int*).

Pointers and 2-D arrays

- Use *arr* to access individual elements of a 2-D array:
  - Since *(*arr* + i) points to the base address of ith 1-D array and it is of base type pointer to int, by using pointer arithmetic we should able to access elements of ith 1-D array.

  - *(*arr* + i) points to the address of the 0th element of the 1-D array. So,
  - *(*arr* + i) + 1 points to the address of the 1st element of the 1-D array
  - *(*arr* + i) + 2 points to the address of the 2nd element of the 1-D array
- Hence
  - *(arr + i) + j points to the base address of jth element of ith 1-D array.
  - On dereferencing *(*arr* + i) + j, we will get the value of jth element of ith 1-D array: *(*(*arr* + i) + j)
  - By using this expression we can find the value of jth element of ith 1-D array.
  - Furthermore, the pointer notation *(*(*arr* + i) + j) is equivalent to the subscript notation.

Pointers and 2-D arrays

```
#include<stdio.h>
int main()  {
    int arr[3][4] = {
                    {11, 22, 33, 44},
                    {55, 66, 77, 88},
                    {11, 66, 77, 44}
                 };
    int i, j;
    for(i = 0; i < 3; i++)  {
        printf("Address of %d th array %u \n",i , *(arr + i));
        for(j = 0; j < 4; j++)
            printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
        printf("\n\n");
    }
    return 0;
}
```

Output

```
Address of 0 th array 2686736
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44

Address of 1 st array 2686752
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88

Address of 2 nd array 2686768
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44
```

Dynamic Memory Allocation

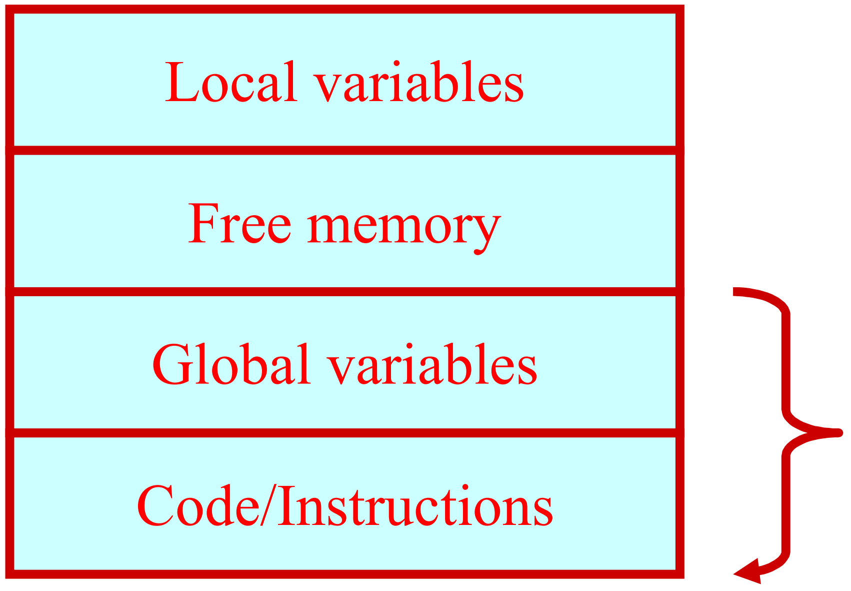
Basic Idea

- Many a time it has been observed data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.

Contd.

- C language requires the number of elements in an array to be specified at compile time.
  - Often leads to wastage of memory space or program failure.
- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library functions.

Memory Allocation Process in C



Stack

Heap

Permanent storage area

Contd.

- The program instructions and the global variables are stored in a region known as permanent storage area.
- The local variables are stored in another area called stack.
- The memory space between these two areas is available for dynamic allocation during execution of the program.
  - This free region is called the heap.
  - The size of the heap keeps changing

Memory Allocation Functions

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

Allocating a Block of Memory: malloc

- A block of memory can be allocated using the function malloc.
  - Reserves a block of memory of specified size and returns a pointer of type void.
  - The return pointer can be assigned to any pointer type.
- General format:

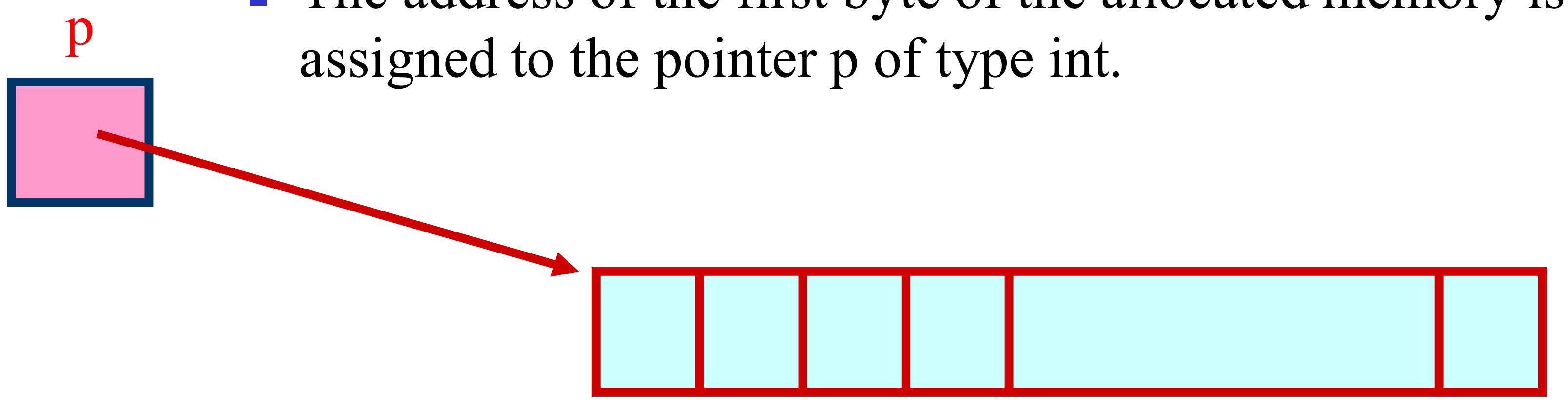
```
ptr  =  (type *)  malloc (byte_size) ;
```

Contd.

- Examples

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

  - A memory space equivalent to "100 times the size of an int" bytes is reserved.
  - The address of the first byte of the allocated memory is assigned to the pointer p of type int.



400 bytes of space

Contd.

```
cptr  =  (char *)  malloc (20) ;
```

- Allocates 20 bytes of space for the pointer cptr of type char.

```
sptr  =  (struct stud *)  malloc (10 *  sizeof (struct stud));
```

Points to Note

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

Example

```
#include <stdio.h>
int main()  {
  int i, n;
  float *height;
  float sum=0,avg;

  printf("Input the number of students. \n");
  scanf("%d", &n);

  height=(float *) malloc(n * sizeof(float));
printf("Input heights for %d students \n", n);
  for(i=0; i<n; i++)
     scanf("%f", &height[i]);

  for(i=0; i< n ;i++)
      sum+=height[i];

  avg=sum/(float) n;

 printf("Average height= %f \n", avg);
 return 0;
}
```

```
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000
```

Program

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);
  ptr = (int*) malloc(n * sizeof(int));
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);
  // deallocating the memory
  free(ptr);
  return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

Program

Using an array of pointers

```
int main() {
   int row = 3, col = 4, i, j, count;
   int* arr[row];
   for (i = 0; i < row; i++)
      arr[i] = (int*)malloc(col * sizeof(int));
   // Note that arr[i][j] is same as *(*(arr+i)+j)
   count = 0;
   for (i = 0; i < row; i++)
      for (j = 0; j < col; j++)
         arr[i][j] = ++count;
            // Or *(*(arr+i)+j) = ++count
   for (i = 0; i < row; i++)
      for (j = 0; j < col; j++)
         printf("%d ", arr[i][j]);
   // Free the dynamically allocated memory
   for (int i = 0; i < row; i++)
      free(arr[i]);
   return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Program

Using pointer to a pointer

```
int main() {
    int row = 3, col = 4, i, j, count;

    int **arr = (int**) malloc(row * sizeof(int*));
    for (i = 0; i < row; i++)
        arr[i] = (int*) malloc(col * sizeof(int));
    count = 0;
    for (i = 0; i < row; i++)
        for (j = 0; j < col; j++)
            arr[i][j] = ++count;
                // OR *(*(arr+i)+j) = ++count
    for (i = 0; i < row; i++)
        for (j = 0; j < col; j++)
            printf("%d ", arr[i][j]);
    for (int i = 0; i < r; i++)
        free(arr[i]);
    free(arr);
    return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Allocating a Block of Memory: calloc()

- A block of memory can be allocated using the function calloc().
  - Reserves multiple blocks of memory of specified size and returns a pointer of type void.
  - The return pointer can be assigned to any pointer type.
  - The calloc() function requires two arguments instead of one as used by malloc().
  - It is initialised to zero.
- General format:

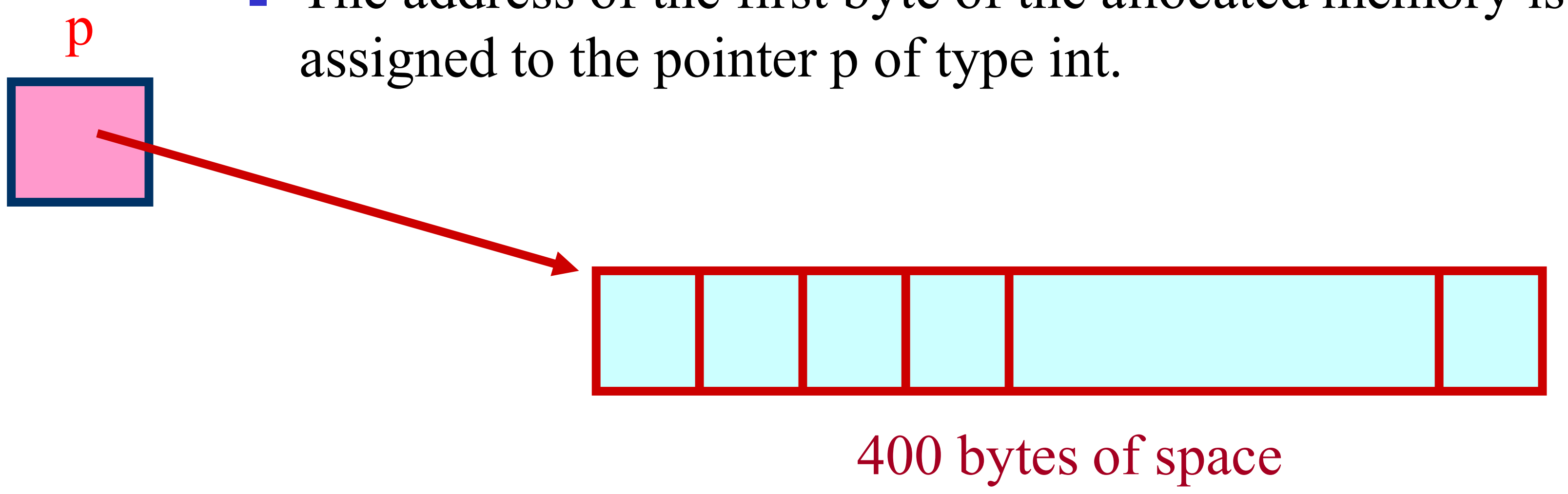
```
ptr  =  (type *)  calloc(n, size) ;
```

Contd.

- Examples

  p = (int *) calloc (100, sizeof (int)) ;

  - A memory space equivalent to "100 times the size of an int" bytes is reserved.
  - The address of the first byte of the allocated memory is assigned to the pointer p of type int.



Contd.

```
cptr  =  (char *)  calloc (20, sizeof(char)) ;
```

- Allocates 10 bytes of space for the pointer cptr of type char.

```
sptr  =  (struct stud *)  calloc(10, sizeof (struct stud));
```

Example

```
#include <stdio.h>
int main()  {
  int i, n;
  float *height;
  float sum=0,avg;

  printf("Input the number of students. \n");
  scanf("%d", &n);

  height=(float *) calloc(n, sizeof(float));
```

```
printf("Input heights for %d students \n", n);
  for(i=0; i<n; i++)
     scanf("%f", &height[i]);

  for(i=0; i< n ;i++)
      sum+=height[i];

  avg=sum/(float) n;

 printf("Average height= %f \n", avg);
 return 0;
}
```

```
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000
```

Releasing the Used Space

- When we no longer need the data stored in a block of memory, we may release the block for future use.
- How?
  - By using the free function.
- General format:

```
free (ptr) ;
```

where ptr is a pointer to a memory block which has been already created using malloc.

2-D Array Allocation

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

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

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

Allocate array of pointers

Allocate array of integers for each row

```
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]);
  }
```

Elements accessed like 2-D array elements.

2-D Array: Contd.

```
void print(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");
      }
}
```

```
int main()  {
  int **p;
  int m, n;
  printf("Enter m and n value:\n");
  scanf("%d%d", &m, &n);
  p=allocate(m, n);
  read_data(p, m, n);
  printf("\n The array read as \n");
  print(p, m, n);
}
```

```
Enter m and n value:
3 3
1 2 3
4 5 6
7 8 9

 The array read as
    1     2     3
    4     5     6
    7     8     9
```

Difference: malloc() & calloc()

Description:

malloc() is an abbreviation for memory allocation. It is a predefined function defined in the *stdlib.h* header file.

calloc() is an abbreviation for contiguous allocation. It is an advancement over malloc() function.

Use:

malloc() is used to allocate memory during the runtime of a program.

calloc() is used to allocate multiple blocks of memory of the same size dynamically.

Memory Initialization:

Memory allocated is uninitialized that means it has garbage values.

Memory is initialized with zero.

Number Of Arguments:

malloc() accommodates a single argument at a time that is, number of byte.

The calloc() takes two arguments. The first argument is the number of blocks of memory to be allocated and the second argument is used to define the size of blocks in terms of bytes.

Difference: malloc() & calloc()

Memory Allocation:

if the memory allocation is successful, it returns a void pointer to the beginning of the allocated memory. This void pointer can be type-casted to any type of pointer. If the memory allocation fails due to reasons like insufficient memory, the malloc () function returns a NULL pointer.

if the memory allocation is successful, it returns a void pointer to the beginning of the allocated memory. This void pointer can be type-casted to any type of pointer. If the memory allocation fails due to reasons like insufficient memory, the calloc() function returns NULL pointer.

Purpose:

malloc() is used for creating structures.

calloc() is for creating dynamic arrays.

Memory Allocation Speed:

malloc() is relatively faster than calloc().

calloc() takes time to allocate multiple blocks of memory, because of the extra step of initializing the allocated memory by zero.

Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed.
  - Memory allocated is larger than necessary.
- How?
  - By using the realloc() function.
- If the original allocation is done by the statement

```
        ptr = malloc(size) ;
```

then reallocation of space may be done as

```
        ptr = realloc (ptr, newsize) ;
```

Contd.

- The new memory block may or may not begin at the same place as the old one.
  - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns NULL and frees the original block.