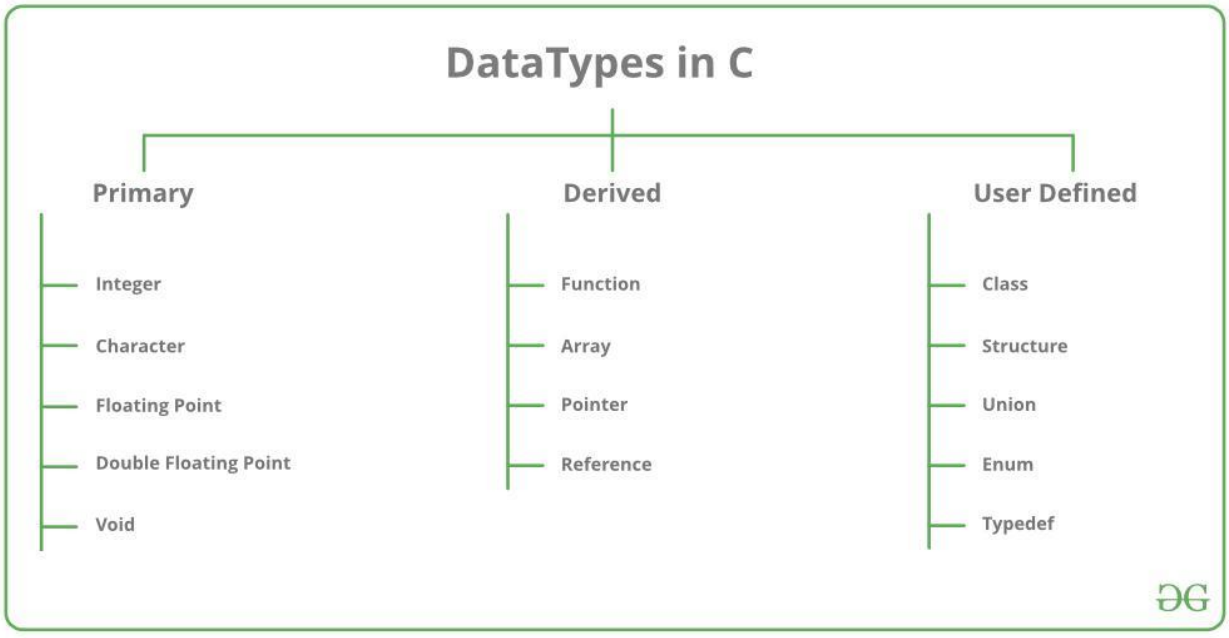
Data Types in C

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

The data types in C can be classified as follows:

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.



Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the ***32-bit GCC compiler***.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c

Data Type	Size (bytes)	Range	Format Specifier
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Note: *The l**ong, short, signed and unsigned** are datatype modifier that can be used with some primitive data types to change the size or length of the datatype.*

The following are some main primitive data types in C:

Integer Data Type

The integer datatype in C is used to store the whole numbers without decimal values. Octal values, hexadecimal values, and decimal values can be stored in int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Syntax of Integer

We use **int keyword** to declare the integer variable:

```
int var_name;
```

The integer data type can also be used as

1. **unsigned int:** Unsigned int data type in C is used to store the data values from zero to positive numbers but it can't store negative values like signed int.
2. **short int:** It is lesser in size than the int by 2 bytes so can only store values from –32,768 to 32,767.
3. **long int:** Larger version of the int datatype so can store values greater than int.
4. **unsigned short int:** Similar in relationship with short int as unsigned int with int.

Note: *The size of an integer data type is compiler-dependent. We can use sizeof operator to check the actual size of any data type.*

Example of int

- C

```c
// C program to print Integer data types.

#include <stdio.h>

int main()

{

    // Integer value with positive data.

    int a = 9;



    // integer value with negative data.

    int b = -9;



    // U or u is Used for Unsigned int in C.

    int c = 89U;



    // L or l is used for long int in C.

    long int d = 99998L;
```

```
    printf("Integer value with positive data: %d\n", a);

    printf("Integer value with negative data: %d\n", b);

    printf("Integer value with an unsigned int data: %u\n",

           c);

    printf("Integer value with an long int data: %ld", d);



    return 0;

}
```

Output

```
Integer value with positive data: 9

Integer value with negative data: -9

Integer value with an unsigned int data: 89

Integer value with an long int data: 99998
```

Character Data Type

Character data type allows its variable to store only a single character. The size of the character is 1 byte. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Syntax of char

The **char keyword** is used to declare the variable of character type:

```
char var_name;
```

Example of char

- C

```c
// C program to print Integer data types.
#include <stdio.h>

int main()
{
    char a = 'a';
    char c;

    printf("Value of a: %c\n", a);

    a++;
    printf("Value of a after increment is: %c\n", a);

    // c is assigned ASCII values
    // which corresponds to the
    // character 'c'
    // a-->97 b-->98 c-->99
    // here c will be printed
    c = 99;

    printf("Value of c: %c", c);
```

```
    return 0;

}
```

Output

```
Value of a: a

Value of a after increment is: b

Value of c: c
```

Float Data Type

In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Syntax of float

The **float keyword** is used to declare the variable as a floating point:

```
float var_name;
```

Example of Float

- C

```
// C Program to demonstrate use

// of Floating types

#include <stdio.h>



int main()

{
```

```
    float a = 9.0f;

    float b = 2.5f;



    // 2x10^-4

    float c = 2E-4f;

    printf("%f\n", a);

    printf("%f\n", b);

    printf("%f", c);



    return 0;

}
```

Output

```
9.000000

2.500000

0.000200
```

Double Data Type

A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C.
The double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points. Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf

Syntax of Double

The variable can be declared as double precision floating point using the **double keyword:**
double *var_name;*

Example of Double

- C

```
// C Program to demonstrate

// use of double data type

#include <stdio.h>



int main()

{

    double a = 123123123.00;

    double b = 12.293123;

    double c = 2312312312.123123;



    printf("%lf\n", a);



    printf("%lf\n", b);



    printf("%lf", c);



    return 0;
```

```
}
```

Output

```
123123123.000000

12.293123

2312312312.123123
```

Void Data Type

The void data type in C is used to specify that no value is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

Syntax:

```
// function return type void


void exit(int check);

// Function without any parameter can accept void.

int print(void);

// memory allocation function which
// returns a pointer to void.
void *malloc (size_t size);
```

Example of Void

- C

```
// C program to demonstrate

// use of void pointers

#include <stdio.h>



int main()
```

```
{

    int val = 30;

    void* ptr = &val;

    printf("%d", *(int*)ptr);

    return 0;

}
```

Output
30

Size of Data Types in C

The size of the data types in C is dependent on the size of the architecture, so we cannot define the universal size of the data types. For that, the C language provides the sizeof() operator to check the size of the data types.

Example

- C

```
// C Program to print size of

// different data type in C

#include <stdio.h>



int main()

{

    int size_of_int = sizeof(int);

    int size_of_char = sizeof(char);
```

```
    int size_of_float = sizeof(float);

    int size_of_double = sizeof(double);



    printf("The size of int data type : %d\n", size_of_int);

    printf("The size of char data type : %d\n",

           size_of_char);

    printf("The size of float data type : %d\n",

           size_of_float);

    printf("The size of double data type : %d",

           size_of_double);



    return 0;

}
```

Output

```
The size of int data type : 4

The size of char data type : 1

The size of float data type : 4

The size of double data type : 8
```

Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
```

```
   float weight;
   char name[30];
};

int main()
{
   struct person *ptr;
   int i, n;

   printf("Enter the number of persons: ");
   scanf("%d", &n);

   // allocating memory for n numbers of struct person
   ptr = (struct person*) malloc(n * sizeof(struct person));

   for(i = 0; i < n; ++i)
   {
       printf("Enter first name and age respectively: ");

       // To access members of 1st struct person,
       // ptr->name and ptr->age is used

       // To access members of 2nd struct person,
       // (ptr+1)->name and (ptr+1)->age is used
       scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
   }

     printf("Displaying Information:\n");
   for(i = 0; i < n; ++i)
       printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);

   return 0;
}
```

Run Code

When you run the program, the output will be:

```
Enter the number of persons:  2
Enter first name and age respectively:  Harry 24
Enter first name and age respectively:  Gary 32
Displaying Information:
Name: Harry     Age: 24
Name: Gary      Age: 32
```

C Program to Multiply Two Matrices Using Multi-dimensional Arrays

```
include <stdio.h>

// function to get matrix elements entered by the user
void getMatrixElements(int matrix[][10], int row, int column) {

   printf("\nEnter elements: \n");

   for (int i = 0; i < row; ++i) {
      for (int j = 0; j < column; ++j) {
         printf("Enter a%d%d: ", i + 1, j + 1);
         scanf("%d", &matrix[i][j]);
      }
   }
}

// function to multiply two matrices
void multiplyMatrices(int first[][10],
                      int second[][10],
                      int result[][10],
                      int r1, int c1, int r2, int c2) {

   // Initializing elements of matrix mult to 0.
   for (int i = 0; i < r1; ++i) {
      for (int j = 0; j < c2; ++j) {
         result[i][j] = 0;
      }
   }
```

```
   // Multiplying first and second matrices and storing it in result
   for (int i = 0; i < r1; ++i) {
      for (int j = 0; j < c2; ++j) {
         for (int k = 0; k < c1; ++k) {
            result[i][j] += first[i][k] * second[k][j];
         }
      }
   }
}

// function to display the matrix
void display(int result[][10], int row, int column) {

   printf("\nOutput Matrix:\n");
   for (int i = 0; i < row; ++i) {
      for (int j = 0; j < column; ++j) {
         printf("%d  ", result[i][j]);
         if (j == column - 1)
            printf("\n");
      }
   }
}

int main() {
   int first[10][10], second[10][10], result[10][10], r1, c1, r2, c2;
   printf("Enter rows and column for the first matrix: ");
   scanf("%d %d", &r1, &c1);
   printf("Enter rows and column for the second matrix: ");
   scanf("%d %d", &r2, &c2);

   // Taking input until
   // 1st matrix columns is not equal to 2nd matrix row
   while (c1 != r2) {
      printf("Error! Enter rows and columns again.\n");
      printf("Enter rows and columns for the first matrix: ");
      scanf("%d%d", &r1, &c1);
      printf("Enter rows and columns for the second matrix: ");
      scanf("%d%d", &r2, &c2);
   }

   // get elements of the first matrix
   getMatrixElements(first, r1, c1);

   // get elements of the second matrix
   getMatrixElements(second, r2, c2);
```

```
    // multiply two matrices.
    multiplyMatrices(first, second, result, r1, c1, r2, c2);

    // display the result
    display(result, r1, c2);

    return 0;
}
```

POINTERS

Definition

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

POINTERS

- Syntax

  Datatype *pointervariable;

- Syntax Example

```
1. int    *ip;    /* pointer to an integer */
2. double *dp;    /* pointer to a double */
3. float  *fp;    /* pointer to a float */
4. char   *ch     /* pointer to a character */
```

POINTERS

- Example

```
int  var = 20;        /* actual variable declaration */

int  *ip;             /* pointer variable declaration */

ip = &var;            /* store address of var in pointer
                         variable*/
```

POINTERS

Reference operator (&) and Dereference operator (*)

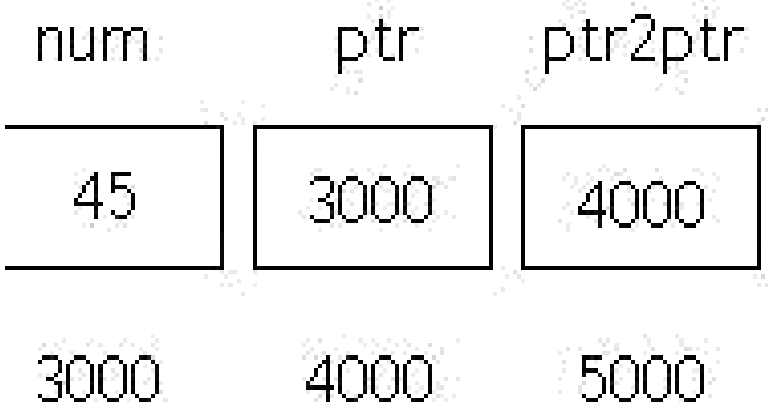
1. & is called reference operator. It gives the address of a variable.

2. * is called dereference operator. It gives the value from the address

Pointer to Pointer

Double (**) is used to denote the double pointer.

Double Pointer Stores the address of the Pointer Variable. Conceptually we can have Triple ….. n pointers.



Example 1

```
int main()
{
int num = 45 , *ptr , **ptr2ptr ;
ptr     = &num;  //3000
ptr2ptr = &ptr;  //4000
printf("%d",**ptr2ptr);
return(0);
}
```

Output 45

Pointer to Constant Objects

These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

```
const datatype *pointername;
```

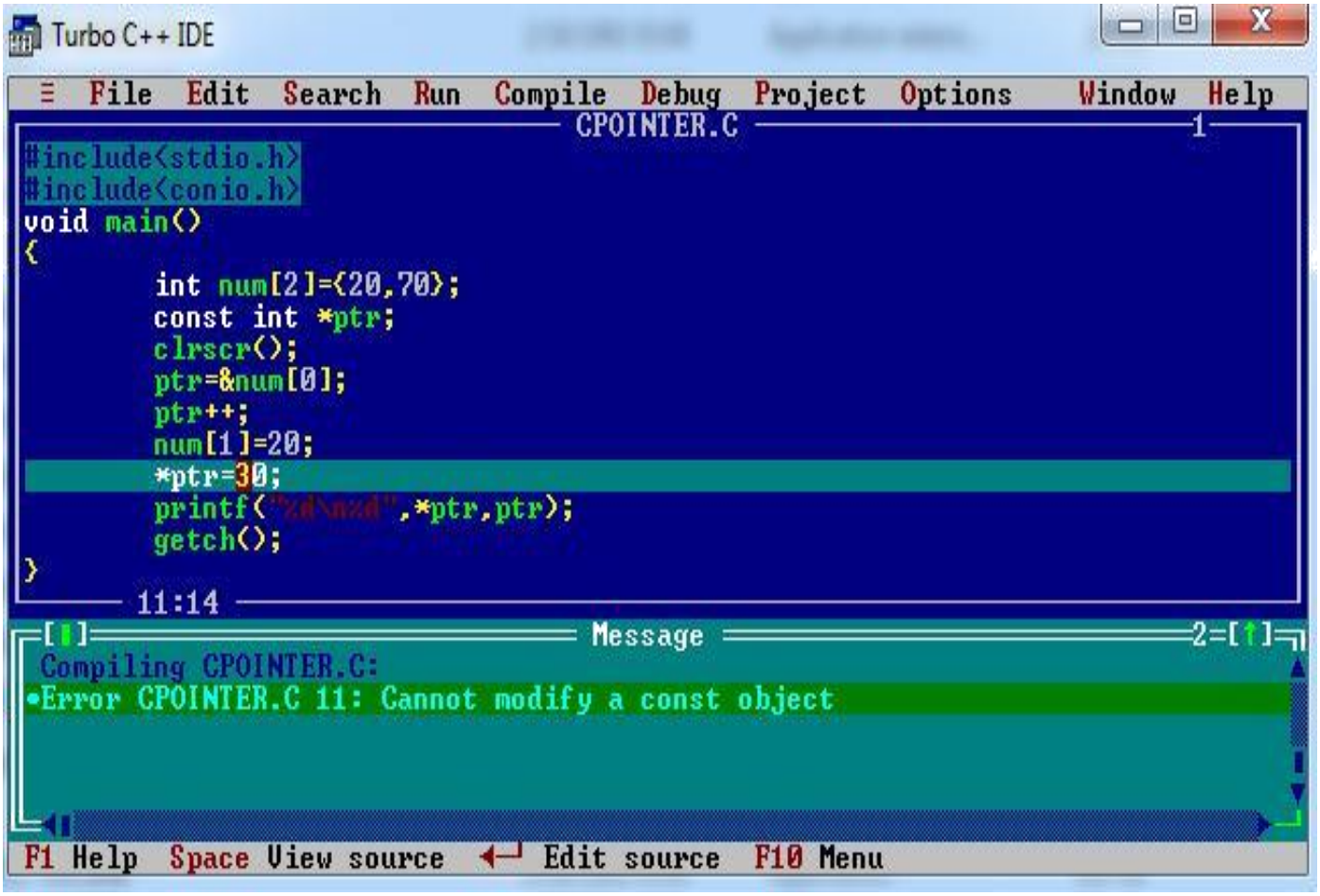
(or)

```
datatype const *pointername;
```

Example

The pointer variable is declared as a const. We can change address of such pointer so that it will point to new memory location, but pointer to such object cannot be modified (*ptr).

Example


Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
CPOINTER.C
#include<stdio.h>
#include<conio.h>
void main()
{
int num[2]={20,70};
const int *ptr;
clrscr();
ptr=&num[0];
ptr++;
num[1]=20;
*ptr=30;
printf("%d\n%d",*ptr,ptr);
getch();
}
11:14
Message
Compiling CPOINTER.C:
•Error CPOINTER.C 11: Cannot modify a const object
F1 Help Space View source Edit source F10 Menu

Constant Pointers

Constant pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). If we try to point the pointer to some other variable, then it is not possible.

```
int* const ptr=&variable;
```

(or)

```
int *const ptr=&variable // ptr is a constant pointer to int
```

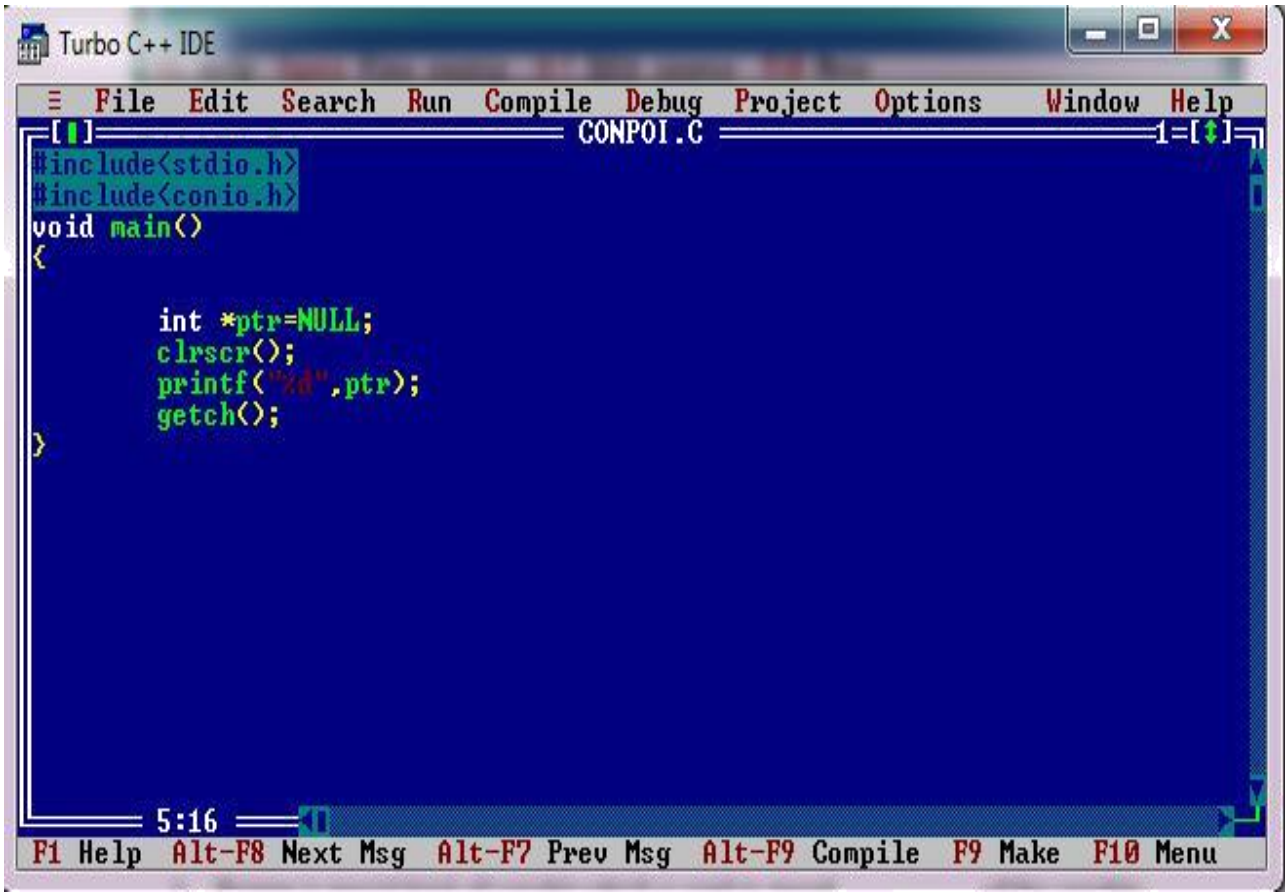
Null pointer

- NULL Pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.

```
datatype *pointer_variable=0;
```

```
datatype *pointer_variable=NULL;
```

Example


Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
CONPOI.C
#include<stdio.h>
#include<conio.h>
void main()
{
int *ptr=NULL;
clrscr();
printf("%d",ptr);
getch();
}
5:16
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

Pointer Arithmetic

- C allows you to perform some arithmetic operations on pointers.
- **Incrementing a pointer**

  Incrementing a pointer is one which increases the number of bytes of its data type.

```
int *ptr;
int a[]={1,2,3};
ptr=&a;
ptr++;
ptr=&a;
ptr=ptr+1;
```

Pointer Arithmetic

- **Decrementing a Pointer**

  Decrementing a pointer is one which decreases the number of bytes of its data type.

  Using Unary Operator

```
int *ptr;
int a[]={1,2,3};
ptr=&a;
ptr--;
```

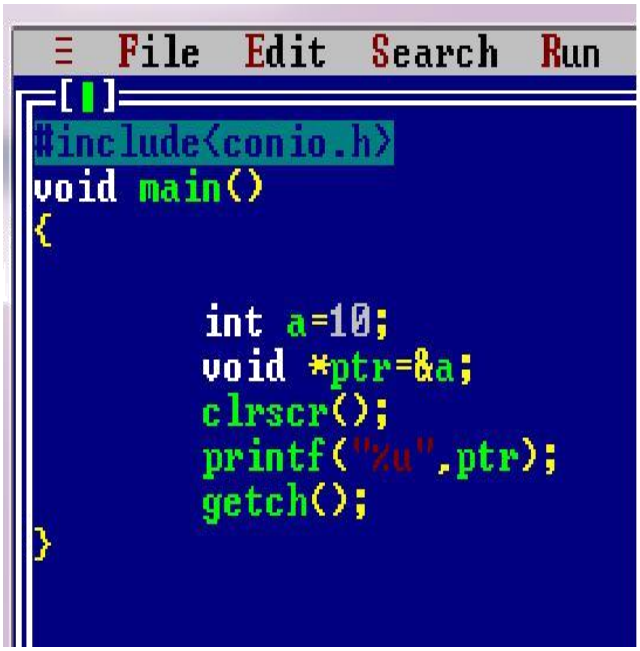
Limitations of Pointer Arithmetic

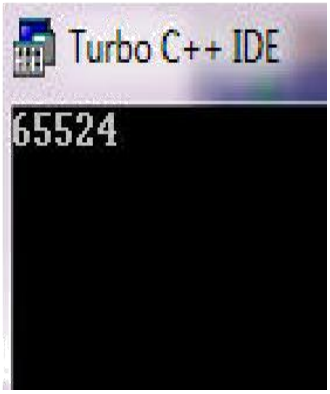
- Addition of 2 pointers is not allowed
- Addition of a pointer and an integer is commutative ptr+5ó 5+ptr
- Subtraction of 2 pointers is applicable.
- subtraction of a pointer and an integer is not commutative ptr-**5≠ 5**-ptr.
- Only integers can be added to pointer. It is not valid to add a float or double value to a pointer.
- A pointer variable cannot be assigned a non address value except zero.
- Multiplication and division Operators cannot be applied on pointers.
- Bitwise operators cannot be applied on pointers.
- A pointer and an integer can be subtracted.
- A pointer and an integer can be added.

Void pointer

1. Void pointer is a generic pointer and can point to any type of object. The type of object can be char, int, float or any other type.

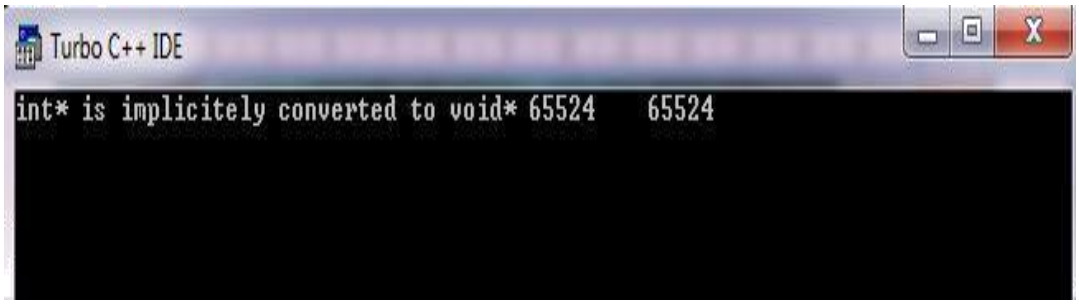
- **Example**



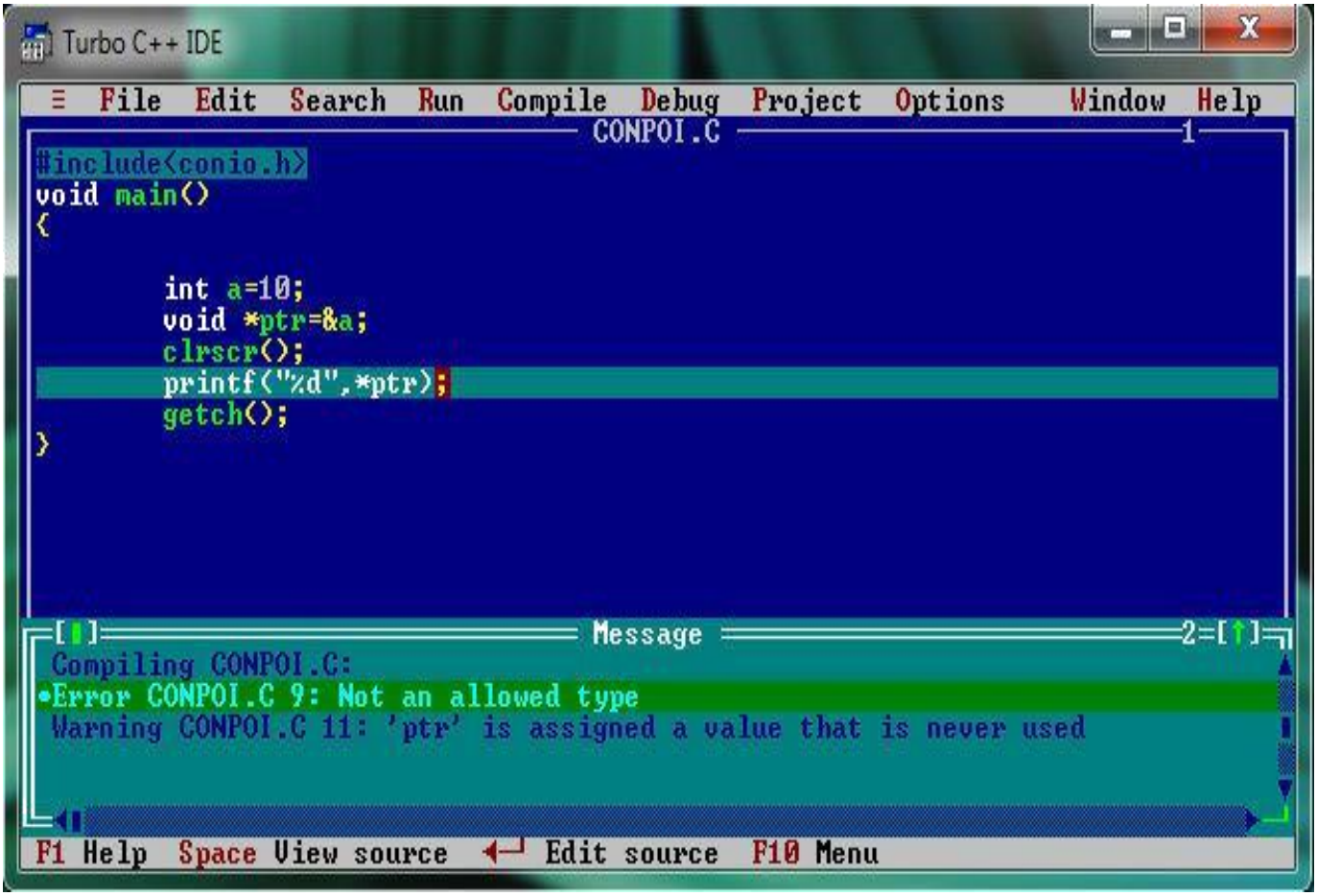


2.A pointer to any type of object can be assigned to a void pointer.

OUTPUT



3. A void pointer cannot be dereferenced

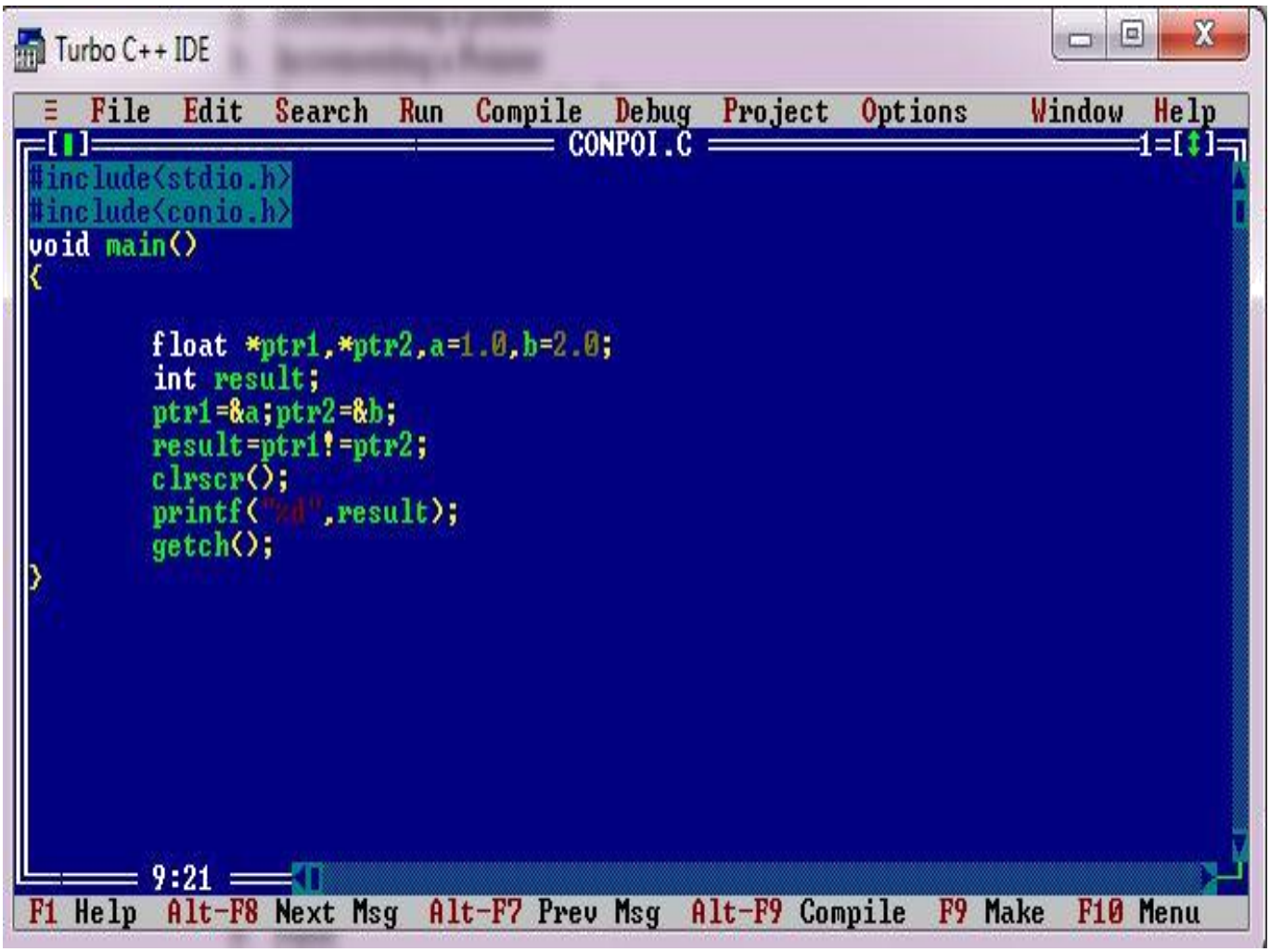


Relational operations

- A pointer can be compared with a pointer of same type or zero.
- Various relational operators are ==.,!=,<,<=,>,>=
- Ex: float a=1.0,b=2.0,*fptr1,*fptr2;
- fptr1=&a;fptr2=&b;
- int result;

  result=fptr1!=fptr2;

Example


Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Window Help
CONPOI.C
#include<stdio.h>
#include<conio.h>
void main()
{
float *ptr1,*ptr2,a=1.0,b=2.0;
int result;
ptr1=&a;ptr2=&b;
result=ptr1!=ptr2;
clrscr();
printf("%d",result);
getch();
}
9:21
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

Pointers and Arrays

- Pointers and arrays are closely related , An array variable is actually just a pointer to the first element in the array.
- Accessing array elements using pointers is efficient way than using array notation.
- When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array.
- Base address i.e address of the first element of the array is also allocated by the compiler.

Pointers and Arrays Cont...

Suppose we declare an array **arr**,

int arr[5]={ 1, 2, 3, 4, 5 };Assuming that the base address of **arr** is 1000 and each integer requires two bytes, the five elements will be stored as follows

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Pointers and Arrays Cont...

- Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**.
- Therefore **arr** is containing the address of **arr[0]** i.e 1000. In short, arr has two purpose- it is the name of an array and it acts as a pointer pointing towards the first element in the array.

Pointers and Arrays Cont...

int *p;

p = arr;

or

p = &arr[0];

Now we can access every element of array **arr** using **p++** to move from one element to another.

Pointers and Arrays Cont...

- a[0] is the same as *a
- a[1] is the same as *(a + 1)
- a[2] is the same as *(a + 2)
- If pa points to a particular element of an array, (pa + 1) always points to the next element, (pa + i) points i elements after pa and (pa - i) points i elements before.

Example 1

```
#include<stdio.h>
void main()
{
int a[10],i,n;
printf("Enter n");
scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n;i++)
printf("a[%d]=%d\n",i,*(a+i));
}
```

Output

```
Enter n 5
1 2 3 4 5
a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5
```

INTRODUCTION TO STRUCTURE

- 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

- A Structure is a collection of variables of different data types under a single name.
- The variables are called members of the structure.
- The structure is also called a user-defined data type.

Defining a Structure

- Syntax:

```
struct structure_name
        {
        data_type member_variable1;  data_type
        member_variable2;
        ……..;  data_type member_variableN;
        };
```

Once structure_name is declared as new data type, then variables of that type can be declared as:

```
struct structure_name structure_variable;
```

Note: The members of a structure do not occupy memory until they are associated with a structure_variable.

- Example

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

- Multiple variables of *struct student* type can be declared as:

```
struct student st1, st2, st3;
```

Defining a structure…

- Each variable of structure has its own copy of member variables.
- The member variables are accessed using the dot (.) operator or member operator.
- For example: *st1.name* is member variable *name* of *st1* structure variable while *st3.gender* is member variable *gender* of *st3* structure variable.

Defining a structure…

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

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

Structure initialization

- Syntax:

struct structure_name structure_variable={value1, value2, …., valueN};

- 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};
clrscr();
printf("Name\t\t\tRoll No.\tMarks\t\tGender\tPhone No.");
printf("\n.....\n");
printf("\n %s\t\t %d\t\t %f\t%c\t %ld", st1.name, st1.roll_no, st1.marks,
    st1.gender, st1.phone_no);
getch();
}
```

Partial Initialization

- We 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 point numbers.
  - '\0' for characters and strings.

```
struct student
    {
    char name[20];
    int roll;
    char remarks;
    float marks;
    };
void main()
{
struct student s1={“name", 4};
clrscr();
printf("Name=%s", s1.name);
printf("\n Roll=%d", s1.roll);
printf("\n Remarks=%c", s1.remarks);
printf("\n Marks=%f", s1.marks);
getch();
}
```

Accessing member of structure/ Processing a structure

- By using dot (.) operator or period operator or member operator.
- Syntax:

  structure_variable.member

- Here, *structure_variable* refers to the name of a *struct* type variable and *member* refers to the name of a member within the structure.

Question

- Create a structure named *student* that has *name*, *roll and mark* as members. Assume appropriate types and size of member. Write a program using structure to read and display the data entered by the user.

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

void main()
{
struct student s;
clrscr();
printf("Enter name:\t");
gets(s.name);
printf("\n Enter roll:\t");
scanf("%d", &s.roll);
printf("\n Enter marks:\t");
scanf("%f", &s.mark);
printf("\n Name \t Roll \t Mark\n");
printf("\n.....\n");
printf("\n%s\t%d\t%f", s.name, s.roll, s.mark);
getch();
}
```

Copying and Comparing Structure Variables

- 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 so by comparing members individually.

Here, structure has been declared global i.e. outside of main() function. Now, any function can access it and create a structure variable.

```
struct student
    {
 char name[20];
 int roll;
 };
void main()
{
struct student student1={"ABC", 4, };
struct student student2;
clrscr();
student2=student1;
printf("\nStudent2.name=%s",
student2.name);
printf("\nStudent2.roll=%d",
student2.roll);
if(strcmp(student1.name,student2.na
me)==0 &&
    (student1.roll==student2.roll))
    {
    printf("\n\n student1 and student2
    are same.");
    }
getch();
}
```

How structure elements 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.
- Computer stores structures using the concept of "word boundary". In a computer with two bytes word boundary, the structure variables are stored left aligned and consecutively one after the other (with at most one byte unoccupied in between them called slack byte).

How structure elements are stored?

- When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined.
- Due to this, even if the members of two variables are equal, their structures do not necessarily compare.
- That's why C does not permit comparison of structures.

Array of structure

- Let us consider we have 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 we can use array of structure to store the records of 100 students which is easier and efficient to handle (because loops can be used).

Array of structure…

- 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];
```

- Write a program that takes roll_no, fname lname of 5 students and prints the same records in ascending order on the basis of roll_no

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);
    }
```

Sorting values

```
for(i=0; i<5; i++)
   {
        for(j=i+1; j<5; j++)
        {
             if(s[i].roll_no<s[j].roll_no)
             {
                 temp = s[i].roll_no;
                 s[i].roll_no=s[j].roll_no;
                 s[j].roll_no=temp;
             }
        }
    }
```

Question

- Define a structure of employee having data members name, address, age and salary. Take the data for n employees in an array and find the average salary.
- Write a program to read the *name*, *address*, and *salary* of 5 employees using array of structure. Display information of each employee in alphabetical order of their name.

Array within Structure

- We can use single or multi dimensional arrays of type *int* or *float*.
- Eg.

```
struct student
{
char name[20];
int roll;
float marks[6];
};
struct student s[100];
```

- Here, the member *marks* contains six elements, *marks[0]*, *marks[1]*, .., *marks[5]* indicating marks obtained in six different subjects.
- These elements can be accessed using appropriate subscripts.
- For example, *s[25].marks[3]* refers to the marks obtained in the fourth subject by the 26th student.

Array within structure…

Reading Values

```
    for(i=0;i<n;i++)
{
    printf("\n Enter information about student%d",i+1);  printf("\n
    Name:\t");
    scanf(" %s",  s[i].name);  printf("\n
    Class:\t");  scanf("%d", &s[i]._class);
    printf("\n Section:");  scanf(" %c",
    &s[i].section);
    printf("\n Input marks of 6subjects:\t");  for(j=0;j<6;j++)
    {
        scanf("%f", &temp);
        s[i].marks[j]=temp;
}   }
```

Structure within another Structure (Nested 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 another Structure (Nested Structure)…

- In the structure above, we can group 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 another Structure (Nested 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 as usual:

  person.name person.salary

```
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 another Structure (Nested Structure)…

- ***Note:- More than one type of structures can be nested…***

```
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,deathday;
      struct name full_name;
      };
```

Assignment

- Create a structure named ***date*** that has ***day***, ***month*** and ***year*** as its members. Include this structure as a member in another structure named ***employee*** which has ***name***, ***id*** and ***salary*** as other members. Use this structure to read and display employee's name, id, date of birthday and salary.

Pointer to Structure

- A structure type pointer variable can be 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.
- Now, individual structure members are accessed as:

bptr->name ***bptr->pages*** ***bptr->price***

OR

(*bptr).name ***(*bptr).pages*** ***(*bptr).price***

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

```
struct book *bptr;

bptr=(struct book *)malloc(sizeof(struct book));

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=temp)
```

Pointer to Structure…

- Also, the address of a structure type variable can be stored in a structure type pointer variable as follows:

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

- Here, the base address of *b* is assigned to *bptr* pointer.

Pointer 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*

Pointer to array of structure

- Let we have 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 zeroth element of *b* to *bptr*.

Pointer 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);
```

Problem

- Define a structure of employee having data members name, address, age and salary. Take data for n employee in an array **dynamically** and find the average salary.
- Define a structure of student having data members name, address, marks in C language, and marks in information system. Take data for n students in an array dynamically and find the total marks obtained.