SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

18CSS101J – Programming for Problem Solving
Unit IV

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI

UNIT IV

INTRODUCTION

Passing Array Element to Function-Formal and Actual Parameters-Advantages of using Functions-Processor Directives and #define Directives-Nested Preprocessor Macro-Advantages of using Functions-Functions-Pointers and address operator-Size of Pointer Variable and Pointer-Operator-Pointer Declaration and dereferencing-pointers-Void Pointers and size of Void Pointers-Arithmetic Operations-Incrementing Pointers-Pointers-Constant Pointers-Pointers to array elements and strings-Function Pointers-Array of Function Pointers-Accessing Array of Function Pointers-Null Pointers-Pointers

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI

INTRODUCTION

- Arrays can be passed as a parameter to a function.
- When it is being passed, the name of the array is enough as an argument instead passing it as an entire array.
- Since the name of the array itself holds the address, no need to include & in the function call.
- Since it passes address , in case of any modification occurs in called function automatically reflects back in the calling function.

Introduction

- Multidimensional arrays are also allowed to be passed as arguments to functions.
- The first dimension is omitted when a multidimensional array is used as a formal parameter in a function.

Formal and Actual Parameters One-Dimensional Array

- **Actual Parameters:**

```
int a[10];
```

Function call:

```
add(a);
```

here,

a is the base address of the array

add is the function

Formal and Actual Parameters One-Dimensional Array

- **Formal Parameters:**

  within the function body

```
void add(int a[])
{
        ….
}
```

  int a[] - is a formal parameter, since it is a on dimensional array size doesn’t matter.

Formal and Actual Parameters Two-Dimensional Array

- **Actual Parameters:**

```
int a[10][10];
```

Function call:

```
add(a);
```

here,

a is the base address of the array 'a'

b is the base address of the array 'b'

add is the function

Formal and Actual Parameters One-Dimensional Array

- **Formal Parameters:**

  within the function body

```
void add(int a[][10])
{
        ....
}
```

  int a[][10] - is a formal parameter, the first dimension is omitted only column value is taken into the account.

Passing Single element

- **Example**

  int a[5]={0,1,2,3,4};

  Function call:

  add(a[1]);

  Here,

  add is the function name,

  a[1] is the second element of the array, value 1 is passed to the array.

Adding Constant five to the given array using functions(One-Dimensional)

- **Main Function**

```
int main()
{
        int a[50],i,n;
        scanf("%d",&n);
        for(i=1;i<=n;i++)
                scanf("%d",&a[i]);
        add(a,n);
        for(i=1;i<=n;i++)
                printf("%d\n",a[i]);
        return 0;
}
```

Adding Constant five to the given array using functions(One-Dimensional)

- **Add Function:**

```
void add(int a[],int n)
{
        int  r;
        for(r=1;r<=n;r++)
        {
                a[r]=a[r]+5;
        }
}
```

Output

- **Input:**

```
5
2 4 6 8 10
```

- **Output:**

```
7
9
11
13
15
```

Main Function

```
int main()
   {
         int num[2][2], i, j;
         printf("Enter 4 numbers:\n");
         for (i = 0; i < 2; ++i)
         {
               for (j = 0; j < 2; ++j)
               {
                     scanf("%d", &num[i][j]);
               }
         }
// passing multi-dimensional array to displayNumbers function
         displayNumbers(num);
         return 0;
   }
```

Display Numbers Function

```
void displayNumbers(int num[2][2])
  {
        // Instead of the above line,
        // void displayNumbers(int num[][2]) is also valid
        int i, j;
        printf("Displaying:\n");
        for (i = 0; i < 2; ++i)
        {
              for (j = 0; j < 2; ++j)
              {
                    printf("%d\n", num[i][j]);
              }
        }
}
```

The advantages of using functions are:

- Divide and conquer
  - Manageable program development
- Software reusability
  - Use existing functions as building blocks for new programs
  - Abstraction - hide internal details (library functions)
- Avoid code repetition

Preprocessor Directives MACROS

- The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation.
- A macro is a segment of code which is replaced by the value of macro.
- Macro is defined by #define directive.
- Preprocessing directives are lines in your program that start with #.
- The # is followed by an identifier that is the directive name.
- For example, #define is the directive that defines a macro.
- Whitespace is also allowed before and after the #.

List of preprocessor directives

#include
#define
#undef
#ifdef
#ifndef
#if
#else
#elif
#endif
#error
#pragma

#include

- The #include preprocessor directive is used to paste code of given file into current file.
- It is used include system-defined and user-defined header files.
- If included file is not found, compiler renders error. It has three variants:

#include <file>

- This variant is used for system header files.
- It searches for a file named file in a list of directories specified by us, then in a standard list of system directories.

#include "file"

- This variant is used for header files of your own program.
- It searches for a file named file first in the current directory, then in the same directories used for system header files.
- The current directory is the directory of the current input file.

#include anything else

- This variant is called a computed #include.
- Any #include directive whose argument does not fit the above two forms is a computed include.

Macro's (#define)

#define token value There are two types of macros:

1.Object-like Macros

2. Function-like Macros

Object-like Macros

- The object-like macro is an identifier that is replaced by value.
- It is widely used to represent numeric constants.

For example:

```
#include <stdio.h>
#define PI 3.1415
main()
{
printf("%f",PI);
}
```

Output:

```
3.14000
```

Function-like Macros

- The function-like macro looks like function call.
- For example:`#define MIN(a,b) ((a)<(b)?(a):(b))`
- Here, MIN is the macro name.

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:

```
Minimum between 10 and 20 is: 10
```

#undef

- To undefine a macro means to cancel its definition. This is done with the #undef directive.

Syntax:

```
#undef token
```

define and undefine example

```
#include <stdio.h>
#define PI 3.1415
#undef PI
 main()
{    printf("%f",PI); }
```

Output

```
Compile Time Error: 'PI' undeclared
```

#ifdef

The #ifdef preprocessor directive checks if macro is defined by #define.
If yes, it executes the code.

Syntax:

```
#ifdef MACRO //code #endif
```

#ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code.

Syntax:

```
#ifndef MACRO //code #endif
```

#if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code.

Syntax:

```
#if expression //code #endif
```

#else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression //if code #else //else code #endif
```

Syntax with #elif

```
#if expression //if code #elif expression //elif code #else
//else code #endif
```

Example

```
#include <stdio.h>
#include <conio.h>
 #define NUMBER 1
void main()
 {
 #if NUMBER==0
 printf("Value of Number is: %d",NUMBER);
#else print("Value of Number is non-zero");
#endif getch();
 }
```

Output

```
Value of Number is non-zero
```

#error

- The #error preprocessor directive indicates error.
- The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
float a; a=sqrt(7);
printf("%f",a);
 }
 #endif
```

#pragma

- The #pragma preprocessor directive is used to provide additional information to the compiler.
- The #pragma directive is used by the compiler to offer machine or operating-system feature.
- Different compilers can provide different usage of #pragma directive.

Syntax:

```
#pragma token
```

Example

```
#include<stdio.h>
#include<conio.h>
void func() ;
#pragma startup func
#pragma exit func
void main()
{
printf("\nI am in main");
getch(); }
void func(){
printf("\nI am in func");
getch();
}
```

Output

```
I am in func
I am in main
I am in func
```

Advantages of Macros

- Time efficiency.
- Not need to pass arguments like function.
- It's preprocessed.
- Easier to Read

Disadvantages of Macros

- Very hard to debug in large code.
- Take more memory compare to function

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 (*)

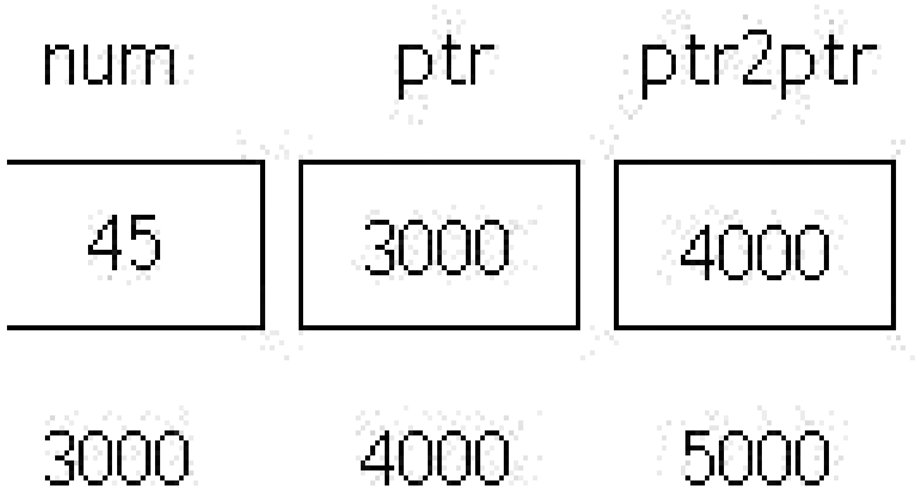
1. & is called reference operator. It gives the address of a variable.

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

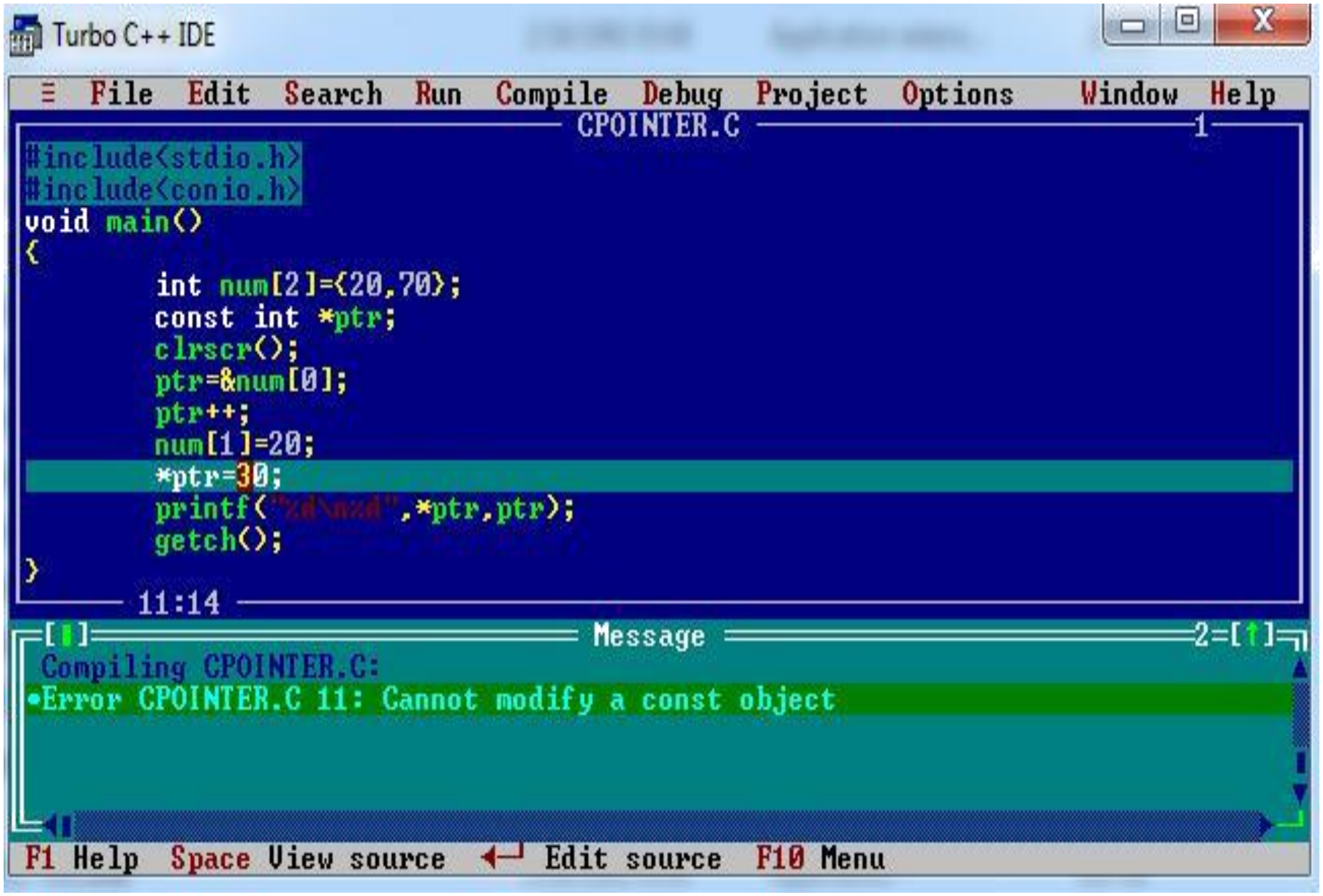
These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

```
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
1
11:14
Message
2
F1 Help Space View source ↵ Edit source F10 Menu

```
#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();
}
```

```
Compiling CPOINTER.C:
•Error CPOINTER.C 11: Cannot modify a const object
```

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.

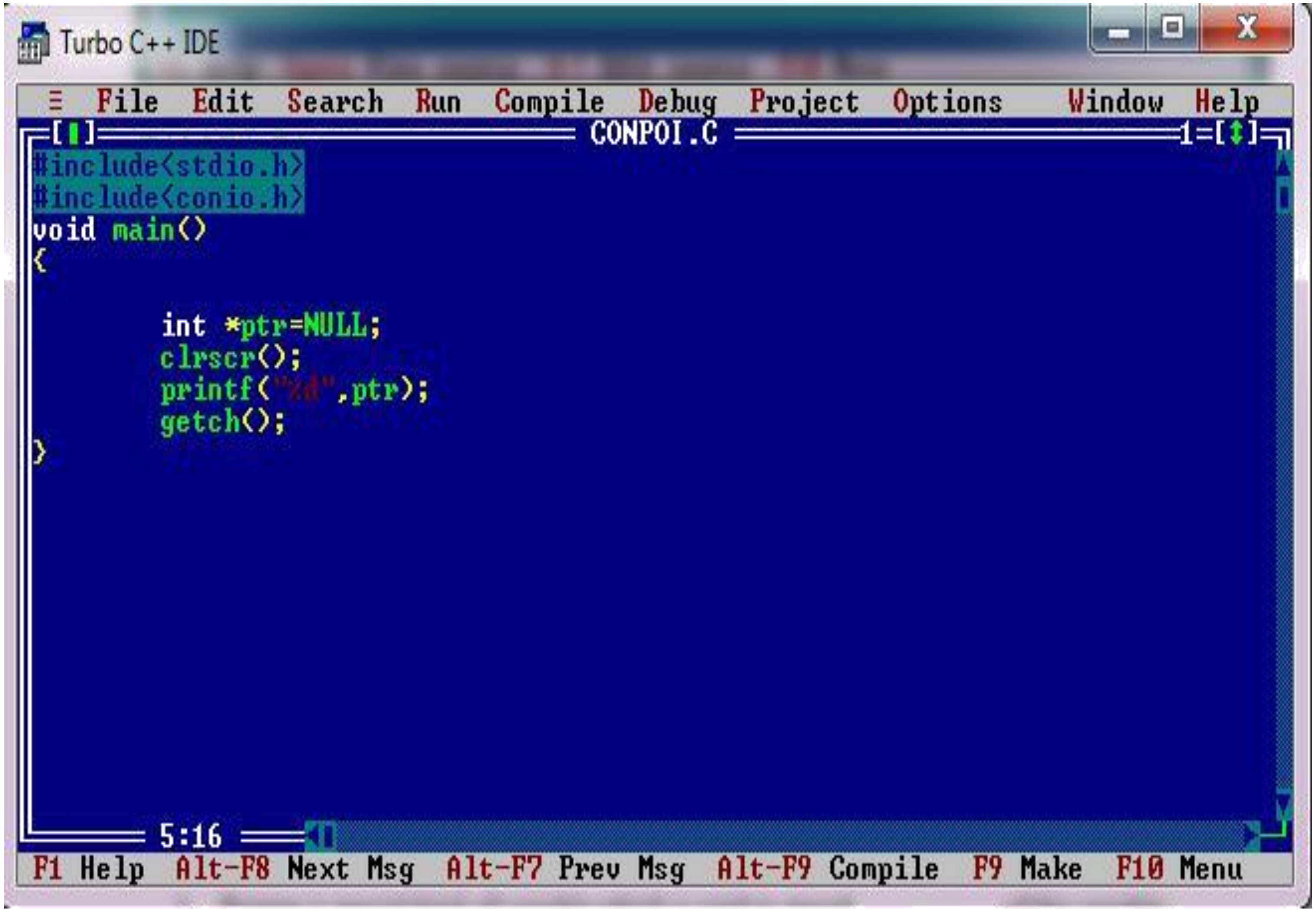
```
int* const ptr=&variable;
        (or)
int *const ptr=&variable // ptr is a constant pointer to int
```

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
1
5:16
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
#include<stdio.h>
#include<conio.h>
void main()
{

        int *ptr=NULL;
        clrscr();
        printf("%d",ptr);
        getch();
}
```

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--;
  ```

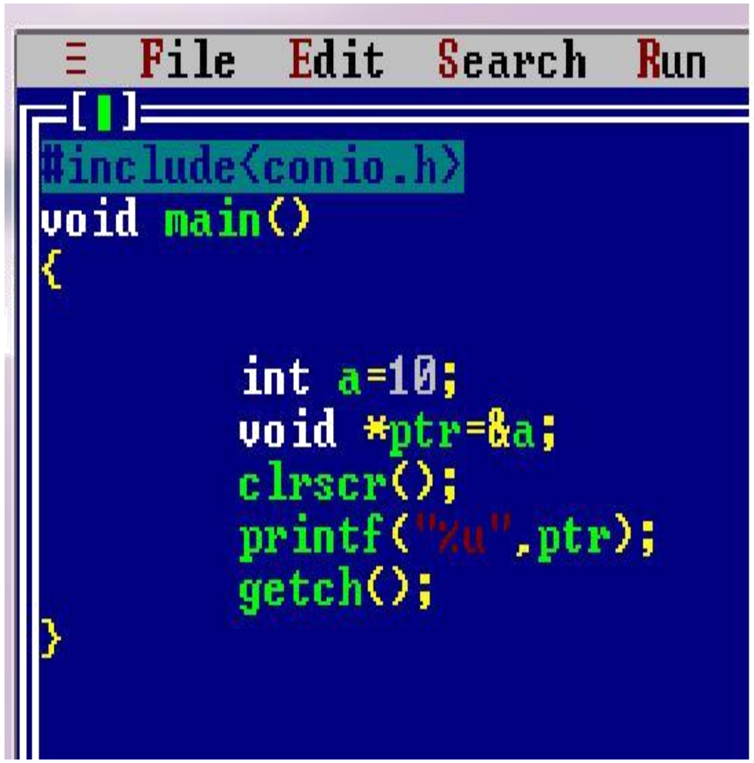
Limitations of Pointer Arithmetic

- 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.

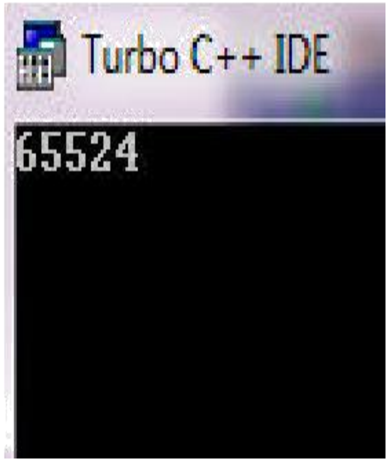
Void pointer

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**

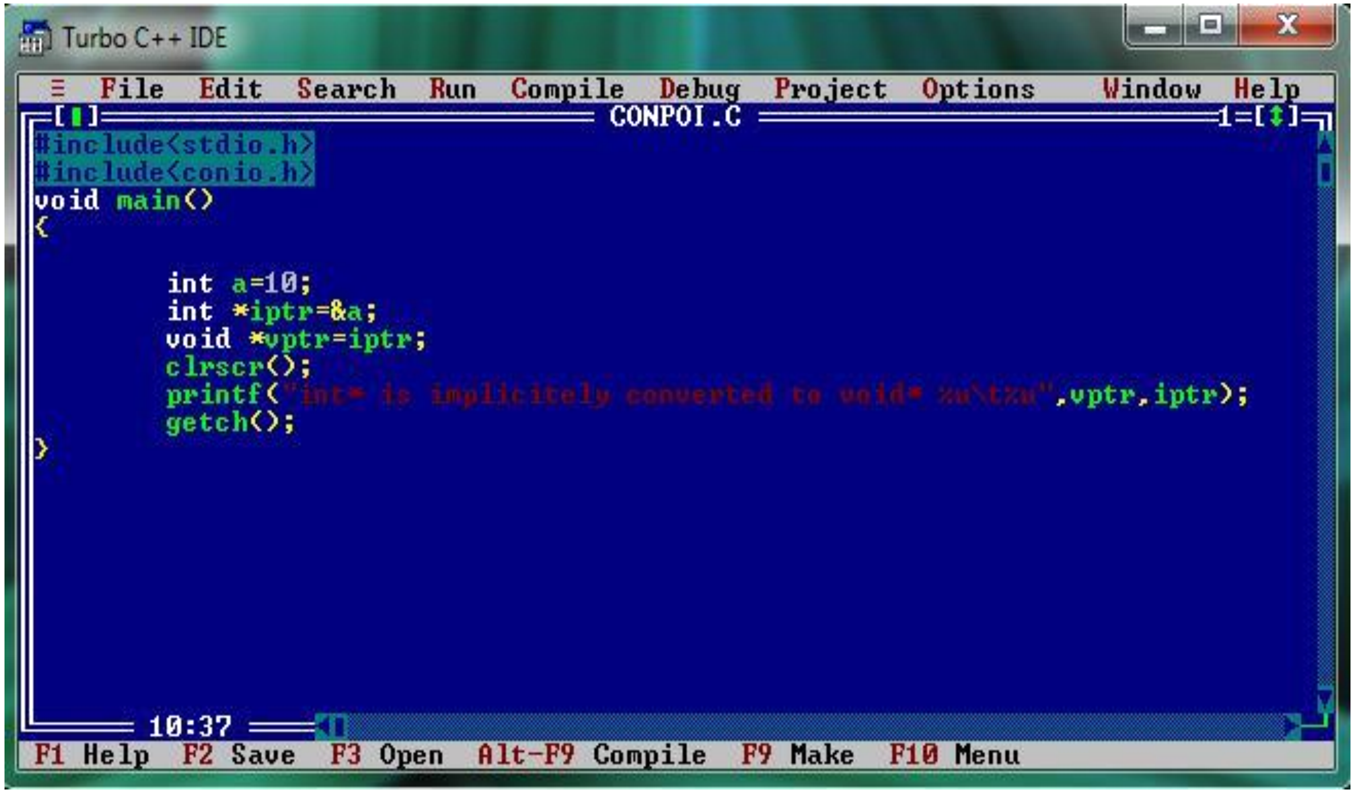


```
#include<conio.h>
void main()
{

        int a=10;
        void *ptr=&a;
        clrscr();
        printf("%u",ptr);
        getch();
}
```



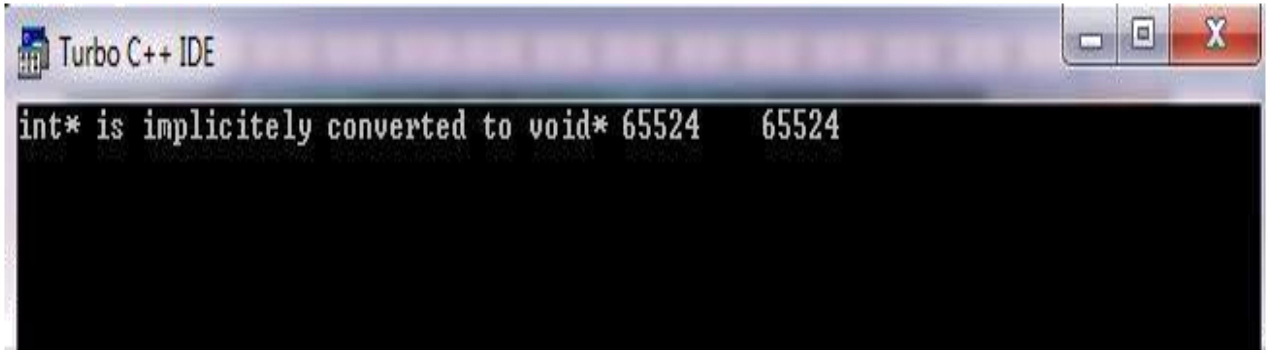
```
65524
```

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


Turbo C++ IDE
≡ File Edit Search Run Compile Debug Project Options Window Help
CONPOI.C
10:37
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

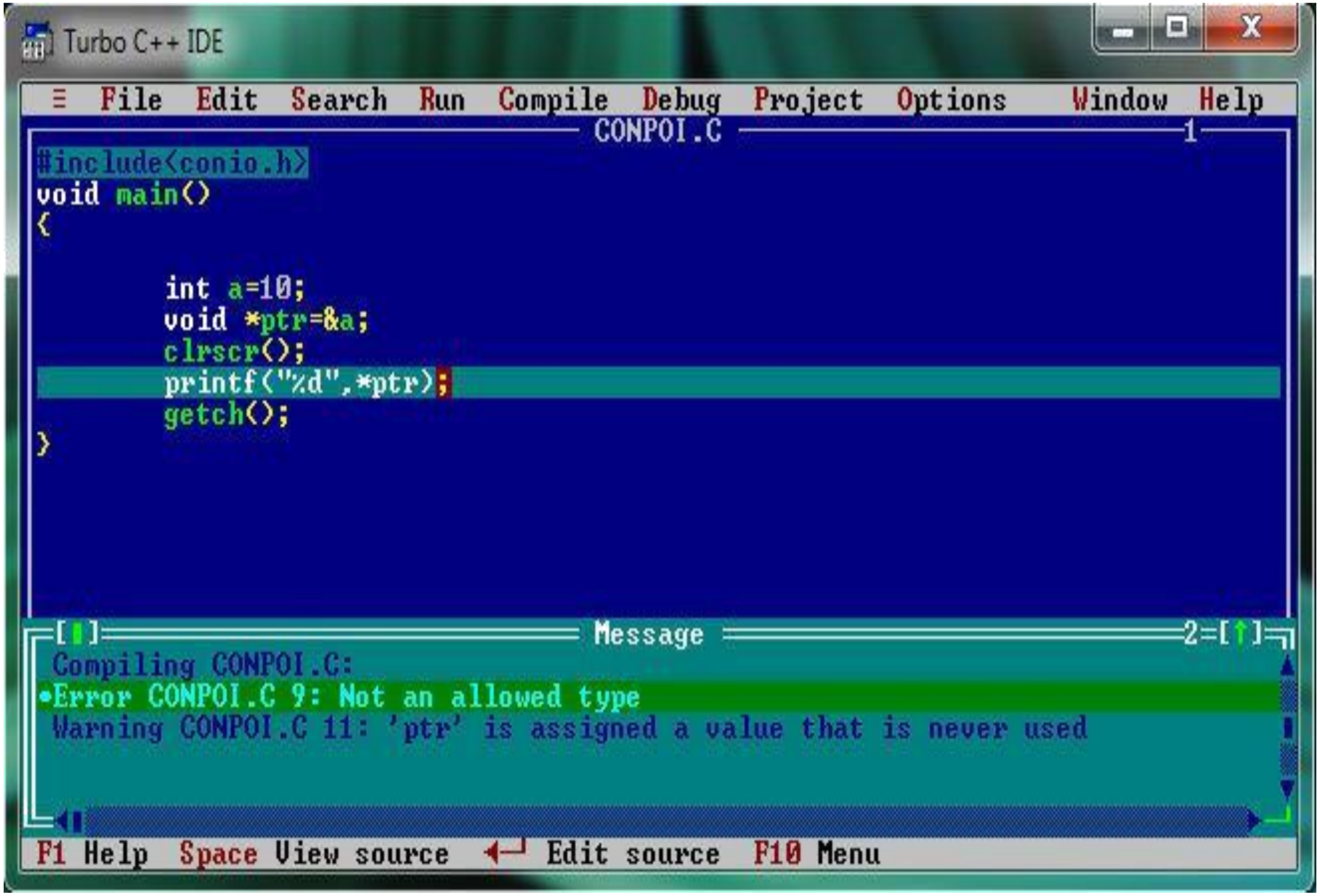
```
#include<stdio.h>
#include<conio.h>
void main()
{

        int a=10;
        int *iptr=&a;
        void *vptr=iptr;
        clrscr();
        printf("int* is implicitely converted to void* %u\t%u",vptr,iptr);
        getch();
}
```

OUTPUT



```
int* is implicitely converted to void* 65524    65524
```

3. A void pointer cannot be dereferenced


Turbo C++ IDE
≡ File Edit Search Run Compile Debug Project Options Window Help
CONPOI.C
1
Message
2
F1 Help Space View source ←┘ Edit source F10 Menu

```
#include<conio.h>
void main()
{

        int a=10;
        void *ptr=&a;
        clrscr();
        printf("%d",*ptr);
        getch();
}
```

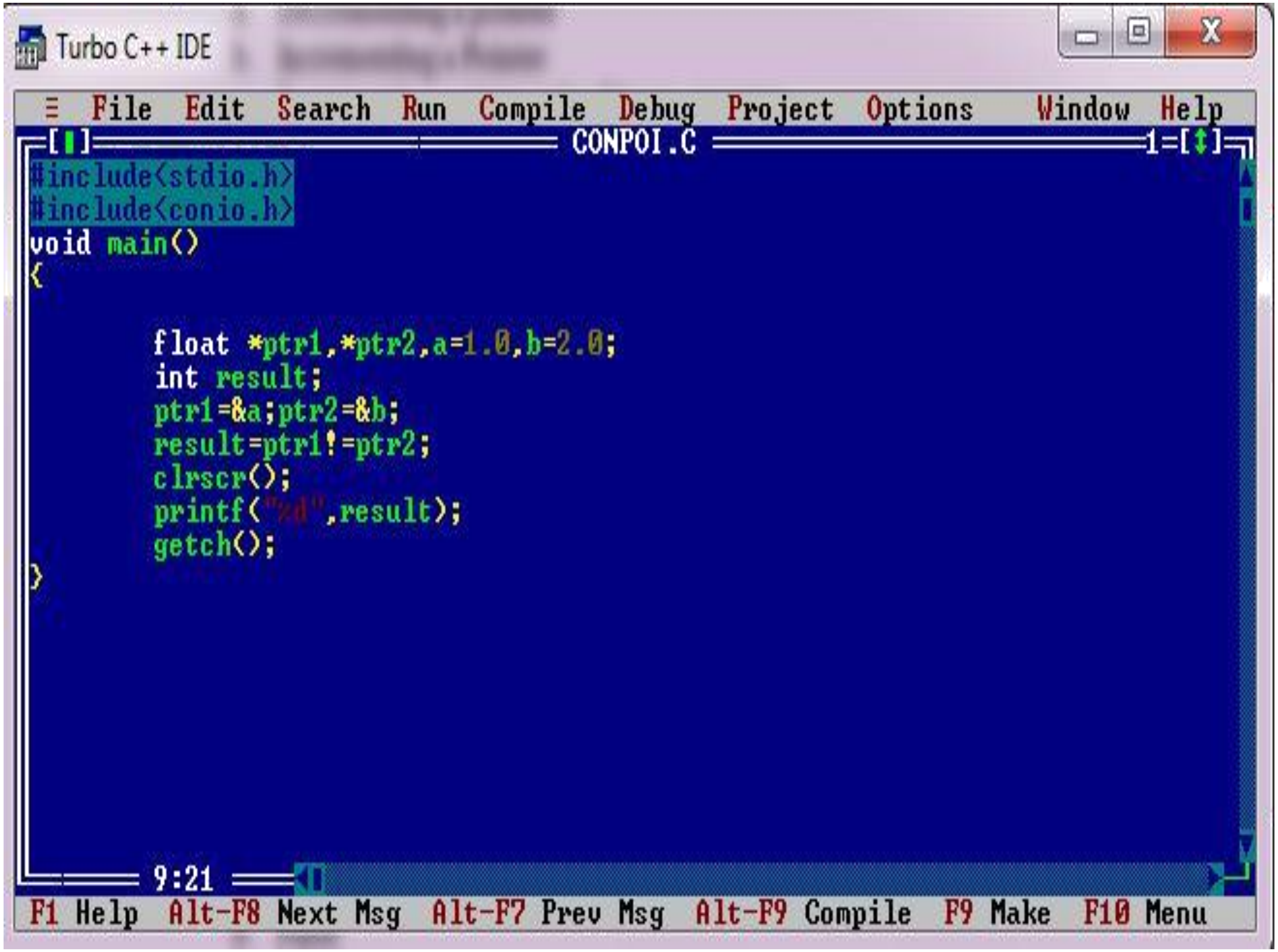
```
Compiling CONPOI.C:
•Error CONPOI.C 9: Not an allowed type
 Warning CONPOI.C 11: 'ptr' is assigned a value that is never used
```

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
1
9:21
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
#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();
}
```

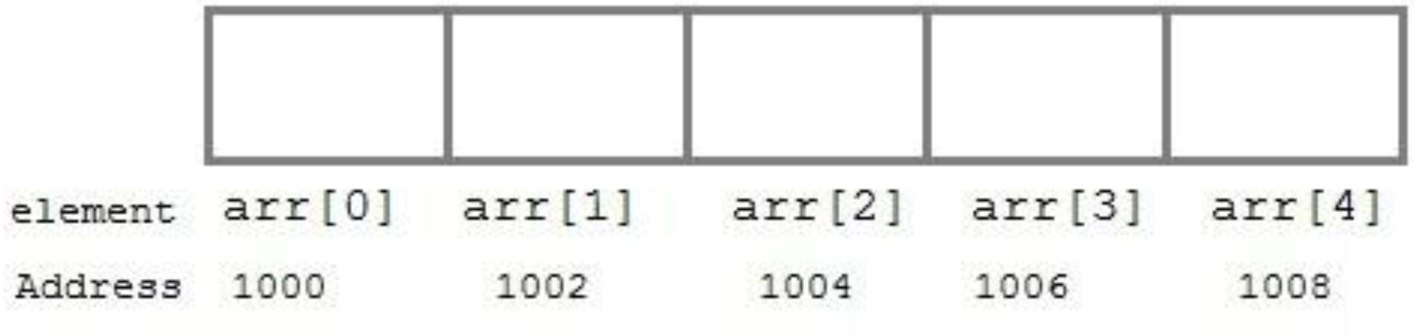
Pointers and Arrays

- 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



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
```

Pointer to Multidimensional Array

- A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array.
- As we know now, name of the array gives its base address. In a[i][j], **a** will give the base address of this array, even a+0+0 will also give the base address, that is the address of **a[0][0]** element.
- Here is the generalized form for using pointer with multidimensional arrays.
- ***(*(a + i) + j)** is same as a[i][j].

Example 2

```
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
   int i,j;
   // declare 4x3 array
   int matrix[ROWS][COLS] = {{1, 2, 3},
                              {4, 5, 6},
                              {7, 8, 9},
                              {10, 11, 12}};
   for (i = 0; i < ROWS; i++)
   {
      for (j = 0; j < COLS; j++)
      {
         printf("%d\t", matrix[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

Output

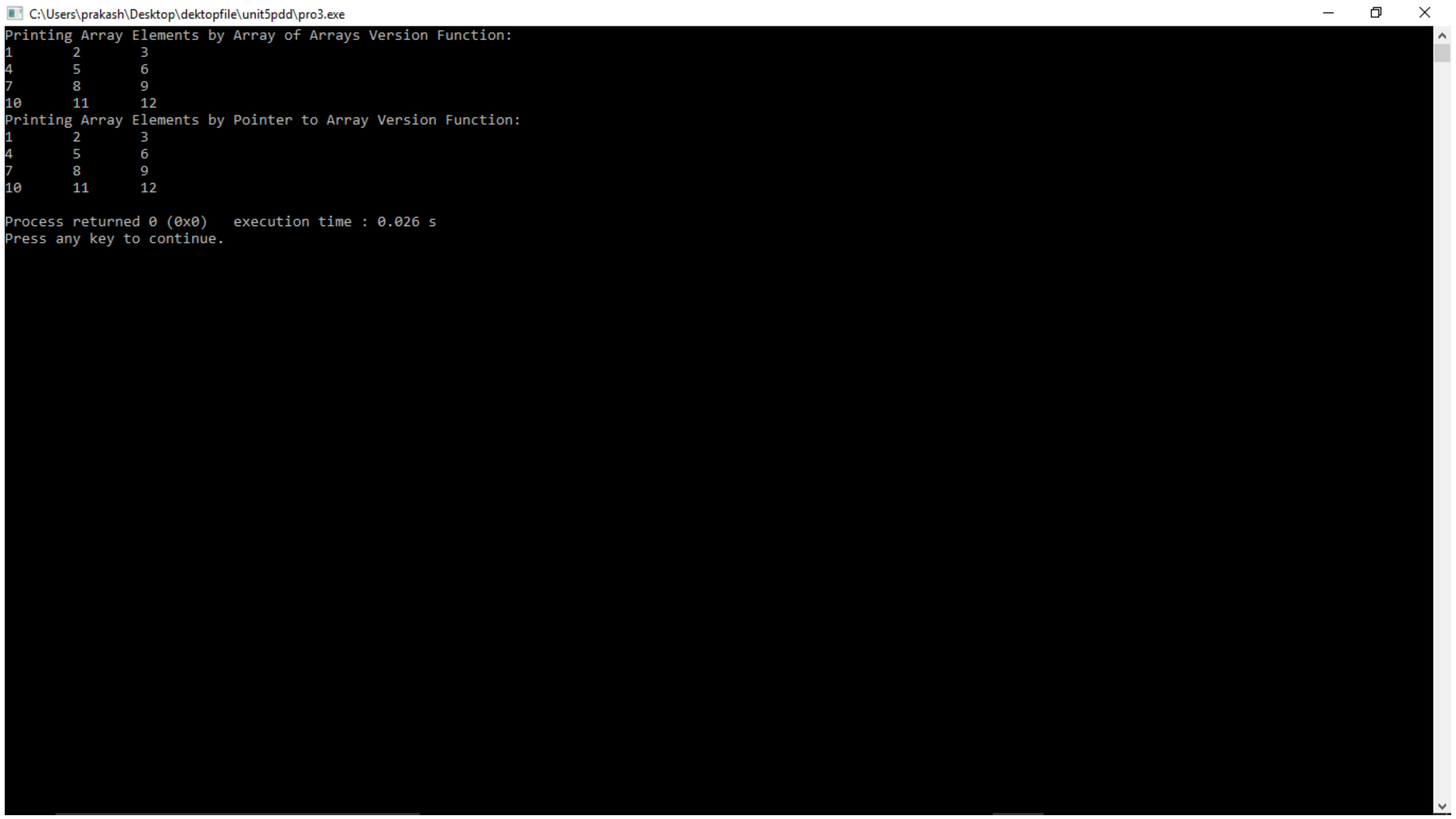
```
1       2       3

4       5       6

7       8       9

10      11      12
```

```
matrix[0][0] = *(*(matrix))

matrix[i][j] = *((*(matrix)) + (i * COLS + j))

matrix[i][j] = *(*(matrix + i) + j)

matrix[i][j] = *(matrix[i] + j)

matrix[i][j] = (*(matrix + i))[j]

&matrix[i][j] = ((*(matrix)) + (i * COLS + j))
```

Example 3

```
void array_of_arrays_ver(int arr[][COLS])
{
   int i, j;
   for (i = 0; i < ROWS; i++)
   {
      for (j = 0; j < COLS; j++)
      {
         printf("%d\t", arr[i][j]);
      }
      printf("\n");
   }
}
void ptr_to_array_ver(int (*arr)[COLS])
{
   int i, j;
   for (i = 0; i < ROWS; i++)
   {
      for (j = 0; j < COLS; j++)
      {
         printf("%d\t", (*arr)[j]);
      }
      arr++;
      printf("\n");
   }
}
```

Output


C:\Users\prakash\Desktop\dektopfile\unit5pdd\pro3.exe

```
Printing Array Elements by Array of Arrays Version Function:
1       2       3
4       5       6
7       8       9
10      11      12
Printing Array Elements by Pointer to Array Version Function:
1       2       3
4       5       6
7       8       9
10      11      12

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Double Pointer

```
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
   // matrix of 4 rows and 3 columns
   int matrix[ROWS][COLS] = {{1, 2, 3},
                             {4, 5, 6},
                             {7, 8, 9},
                             {10, 11, 12}};
   int** pmat = (int **)matrix;
   printf("&matrix[0][0] = %u\n", &matrix[0][0]);
   printf("&pmat[0][0] = %u\n", &pmat[0][0]);
   return 0;
}
```

Output

```
&matrix[0][0] = 2675498000

&pmat[0][0] = 1
```

Passing an array to a function

- Single element of an array can be passed in similar manner as passing variable to a function
- **C program to pass a single element of an array to function**

```
#include <stdio.h>
void display(int age)
{
    printf("%d", age);
}

int main()
{
    int ageArray[] = { 2, 3, 4 };
    display(ageArray[2]); //Passing array element ageArray[2] only.
    return 0;
}
```

```
OUTPUT
4
```

Passing an entire one-dimensional array to a function

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

Output

```
Enter n

5

1

2

3

4

5

Sum=15
```

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example 4

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
   int num[2][2], i, j;
   printf("Enter 4 numbers:\n");
   for (i = 0; i < 2; ++i)
      for (j = 0; j < 2; ++j)
         scanf("%d", &num[i][j]);
   // passing multi-dimensional array to displayNumbers function
   displayNumbers(num);
   return 0;
}
```

```
void displayNumbers(int num[2][2])
{
 // Instead of the above line,
  // void displayNumbers(int num[][2]) is also valid
   int i, j;
   printf("Displaying:\n");
   for (i = 0; i < 2; ++i)
      for (j = 0; j < 2; ++j)
         printf("%d\n", num[i][j]);
}
```

Output

```
Enter 4 numbers:
1
2
3
4
Displaying:
1
2
3
4
```

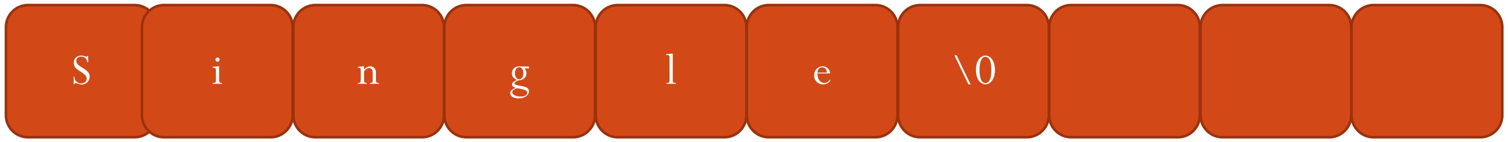
Pointers and Strings

- **Strings as arrays:**In C, the abstract idea of a string is implemented with just an array of characters. For example, here is a string:
- char label[] = "Single"; What this array looks like in memory is the following:

S	i	n	g	l	e	\0

Pointers and Strings cont…

- where the beginning of the array is at some location in computer memory, for example, location 1000.
- A character array can have more characters than the *abstract string* held in it, as below:
- char label[10] = "Single"; giving an array that looks like:



- we can access each character in the array using subscript notation, as in:

```
printf("Third char is: %c\n", label[2]);
```

- which prints out the third character, **n**.

Disadvantage Of Creating Strings Using The Character Array

- A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

```
char label[] = "Single"; /* 7 characters */
char label[10] = "Single";
```

- Thus, you must specify the maximum number of characters you will ever need to store in an array.
- This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

String as Pointers

- Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.
- However, pointers only hold an address, they cannot hold all the characters in a character array.
- This means that when we use a char * to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

```
char label[] = "Single";
char label2[10] = "Married";
char *labelPtr;
labelPtr = label;
```

Example1

```
#include <stdio.h>
int main()
{
 char *sample = "From whence cometh my help?n";
 while(putchar(*sample++))
 ;
 return(0);
}
```

Output:

```
From whence cometh my help?n
```

Passing Strings

Below is the definition of a function that prints a label and a call to that function:

```
void PrintLabel(char the_label[])
{
printf("Label: %s\n", the_label);
 }
…
int main(void)
{
char label[] = "Single";
 …
PrintLabel(label);
 …
}
```

Passing Strings Cont...

- Since label is a character array, and the function PrintLabel() expects a character array, the above makes sense.
- However, if we have a pointer to the character array label, as in:

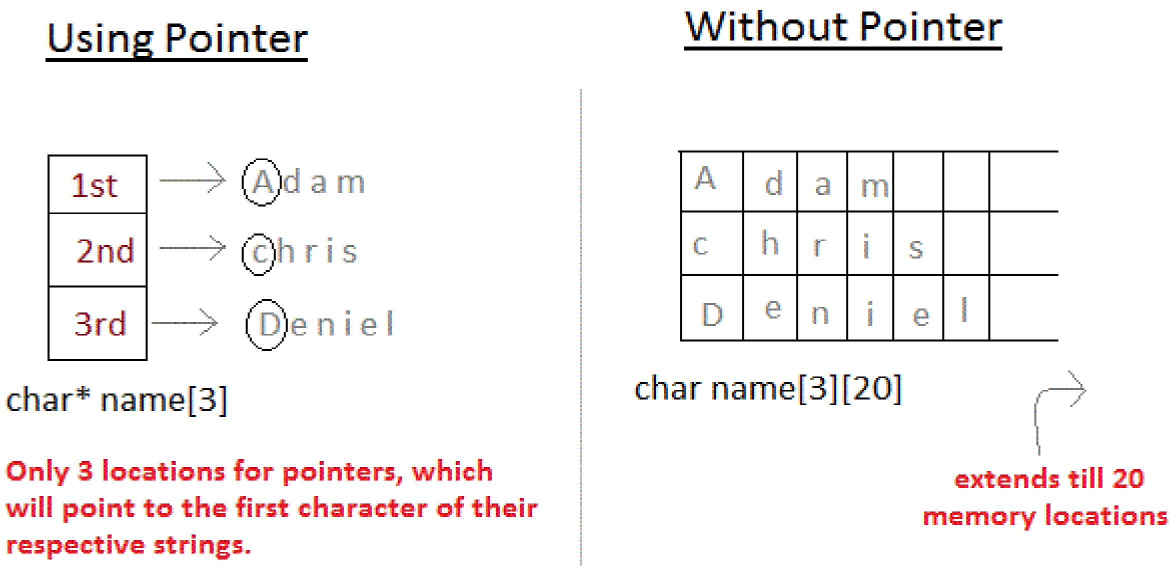
  char *labelPtr = label;

  then we can also pass the pointer to the function, as in:
- PrintLabel(labelPtr);

Array of Pointers

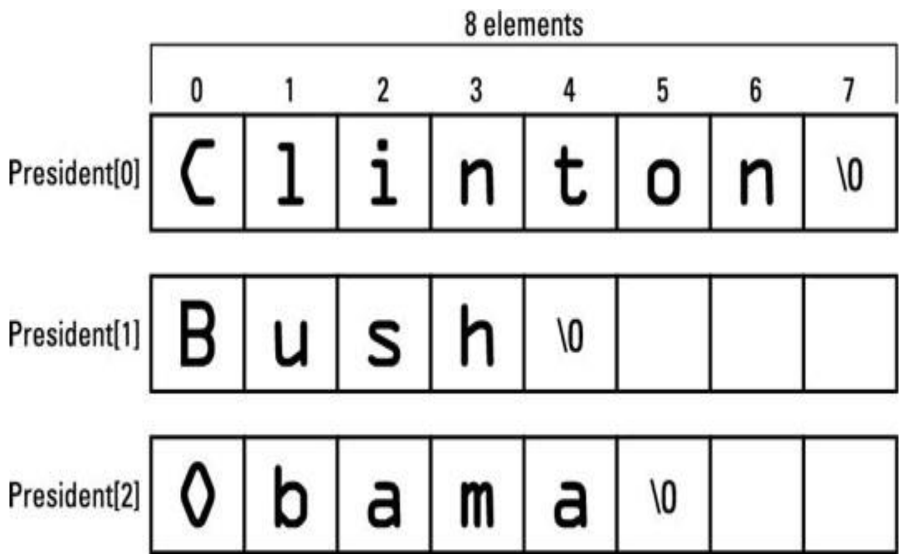
- Pointers are very helpful in handling character array with rows of varying length.

  char *name[3]={ "Adam", "chris", "Deniel" }; *//Now see same array without using pointer* char name[3][20]= { "Adam", "chris", "Deniel" };


Using Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

Example1

```
#include <stdio.h>
#define SIZE 3
int main()
{
 char president[SIZE][8] = {
 "Clinton",
 "Bush",
 "Obama"
 };
 int x,index;
 for(x=0;x<SIZE;x++)
 {
 index = 0;
 while(president[x][index] != '\0')
 {
 putchar(president[x][index]);
 index++;
 }
 putchar('n');
 }
 return(0);
}
```



Output:

```
ClintonnBushnObaman
```

Function Pointers

- In C, like normal data pointers (int *, char *, etc), we can have pointers to functions.
- Initialization

  return_type function_pointer(argu)=&function_name

  void (*fun_ptr)(int) = &fun;
- Function Definition

```
void fun(int a)
{
    printf("Value of a is %d\n", a);
}
```

Example1

```
#include<stdio.h>
void fun(int a)
{
   printf("a=%d\n",a);
}
void main()
{
   void (*fun1)(int)=&fun;
   (*fun1)(15);
}
```

Output:

```
a=15
```

Function Pointers Cont…

If we remove bracket, then the expression

"void (*fun_ptr)(int)"

becomes

"void *fun_ptr(int)"

which is declaration of a function that returns void pointer.

Interesting facts

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address.

Passing Function Pointer as an argument

```
#include<stdio.h>
void fun1(void(*test)())
{
int a=20;
test(a);
}
void test(int a)
{
printf("a=%d\n",a);
}
void main()
{
fun1(&test);
}
```

Output:

```
a=20
```

Array of Function Pointer

```
#include<stdio.h>
void add(int a,int b)
{
   printf("add=%d\n",a+b);
}
void sub(int a,int b)
{
   printf("sub=%d\n",a-b);
}
void mul(int a,int b)
{
   printf("mul=%d\n",a*b);
}
```

```
   void main()
   {
      void
(*fun[])(int,int)={add,sub,mul};
      int ch;
      int a,b;
      printf("Enter a and b\n");
      scanf("%d%d",&a,&b);
      printf("Enter the
operation\n");
      scanf("%d",&ch);
      if(ch>2)
         printf("Wrong Input\n");
      else
         (*fun[ch])(a,b);
   }
```

Output

```
Enter a and b
4   5
Enter the operation
2
mul=20
```

Structures created and accessed using pointers

- Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name
 {
member1;
 member2;
 . .
};
int main()
{
struct name *ptr;
 }
```

Here, the pointer variable of type struct name is created.

Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

Referencing pointer to another address to access the memory

Consider an example to access structure's member through pointer.

```
#include <stdio.h>
typedef struct person
{   int age;
 float weight;
};
int main()
{
struct person *personPtr, person1;
personPtr = &person1;          // Referencing pointer to memory address of person1
 printf("Enter integer: ");
scanf("%d",&(*personPtr).age);
 printf("Enter number: ");
scanf("%f",&(*personPtr).weight);
printf("Displaying:\n");
  printf("age=%d\nweight=%f",(*personPtr).age,(*personPtr).weight);
return 0;
}
```

Output

```
Enter integer: 3
Enter number: 6
Displaying:
age=3
weight=6.000000
```

->

- **Using -> operator to access structure pointer member.**
- Structure pointer member can also be accessed using -> operator.
- (*personPtr).age is same as personPtr->age
- (*personPtr).weight is same as personPtr->weight

Accessing structure member through pointer using dynamic memory allocation

- To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.
- **Syntax to use malloc()**

  ptr = (cast-type*) malloc(byte-size)

Example1

```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
   float weight;
   char name[30];
};
int main()
{
   struct person *ptr;
   int i, num;
   printf("Enter number of persons: ");
   scanf("%d", &num);
   ptr = (struct person*) malloc(num *
sizeof(struct person));
   // Above statement allocates the memory for n
structures with pointer personPtr pointing to
base address */
```

```
for(i = 0; i < num; ++i)
{
printf("Enter name, age and weight of the
person respectively:\n");
 scanf("%s%d%f", &(ptr+i)->name,
&(ptr+i)->age, &(ptr+i)->weight);
 }

printf("Displaying Infromation:\n");
 for(i = 0; i < num; ++i)
printf("%s\t%d\t%.2f\n", (ptr+i)->name,
(ptr+i)->age, (ptr+i)->weight);
 return 0;
}
```

Output

```
Enter number of persons: 2
Enter name, age and weight of the person respectively:
aaa
12
45
Enter name, age and weight of the person respectively:
bbb
87
65
Displaying Infromation:
aaa     12      45.00
bbb      87      65.00
```

Self-Referential Structure

- A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
struct struct_name
{
datatype datatypename;
struct_name * pointer_name;
};
```

Self-Referential Structure Cont…

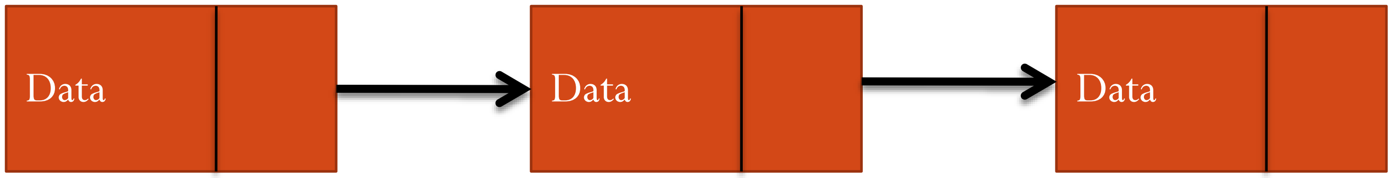
- A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type.
- For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

```
typedef struct listnode {
void *data;
struct listnode *next;
} linked_list;
```

In the above example, the listnode is a self-referential structure – because the *next is of the type struct listnode.

Self-Referential Structure Cont…

```
typedef struct listnode {
    void *data;
    struct listnode *next;
    } linked_list;
```



THANK YOU