MODULE 8
DERIVED DATA TYPE - POINTERS

Point to here, point to there, point to that, point to this, and point to nothing!
But Oh N0000! they are just memory addresses!!

My Training Period: hours

Note: This Module may be one of the toughest topics in C/C++ that complained by students :o), although it is one of the most important topic.

Abilities

- Able to understand and use pointers.
- Able to understand and use pointer operators.
- Able to understand, relate and use array and pointers.
- Able to understand, relate and use array, functions and pointers.
- Able to understand and use the `main()` command line arguments.
- Able to understand and use function pointers.
- Able to understand and use `NULL` and `void` pointers.
- Able to understand and use string and pointers.

8.1 What Is Pointers?

- Pointers provide a powerful and flexible method for manipulating data in programs.
- Some program routines can be more efficiently executed with pointers than without them, and some routines can be performed only with pointers.
- To understand how to use pointers, a programmer must have a basic knowledge of how a computer stores information in memory. Pointers closely relate to memory manipulation.
- Basically, a personal computer Random Access Memory (RAM) consists thousands of sequential storage locations, with each location being identified by a unique address. Computer's processor also has their own memory, normally called registers and cache. They differ in term of access speed, price and their usage.
- The memory addresses in a given computer, range from 0 to a maximum value that depends on the amount of physical memory installed.
- Nowadays 128, 256 MB up to GB of RAM installed on the PCs are normal.
- Computer memory is used for storage, when program runs on the computer and for processing data and instructions.
- For example, when you use Microsoft Word program, it will occupy some of the computer's memory.
- In a very simple way, each program requires two portions of the memory that is:

    1. Data portion – for data or operands.
    2. The instruction code portion – what to do to the data such as operators etc.

- Each portion is referred to as a memory segment, so there is:

    1. A data segment (normally called **data** segment).
    2. An instruction code segment (normally called **text** segment).

- Now we just concentrate the memory storage for program data, the data segment. Details of the memory allocation story can be found in ModuleW and ModuleZ.
- When the programmer declares a variable in a C/C++ program, the compiler sets aside a memory location with a unique address to store that variable.
- The compiler associates that address with the variable's name. When the program uses the variable name, it automatically accesses the proper memory location.
- The locations address remains hidden from the programmer, and the programmer need not be concerned with it. What we are going to do is to manipulate the memory addresses by using pointers.
- Let say we declare one variable named `rate` of type integer and assign an initial value as follows:

```
int   rate = 100;
```

- Then the variable is stored at a specific memory address and as an illustration, can be depicted as follows:

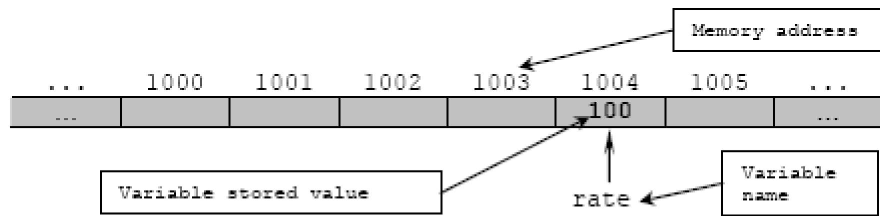


Figure 8.1

- You can see, the memory address of the variable `rate` (or any other variable) is a number, so can be treated like any other number in C/C++. Normally the number is in hexadecimal format.
- Then, if a variable's memory address is known, the programmer can create a second variable for storing a memory address of the first variable.
- From the above Figure, we can declare another variable to hold the memory address of variable `rate`; let say gives it a name, `s_rate` (Figure 8.2).
- At the beginning, `s_rate` is uninitialized. So, storage has been allocated for `s_rate`, but its value is undetermined, as shown below.

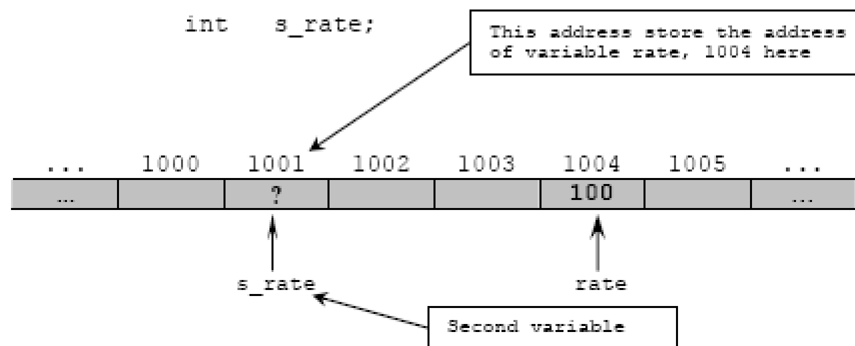


Figure 8.2

- Let store the memory address of variable `rate`, in variable `s_rate`, so, `s_rate` now contains the memory address of `rate`, which indicates its storage location in memory where the actual data (`100`) is stored.
- Finally, in C/C++ vocabulary, `s_rate` is pointing to `rate` or is said a pointer to `rate` and final illustration is shown below.

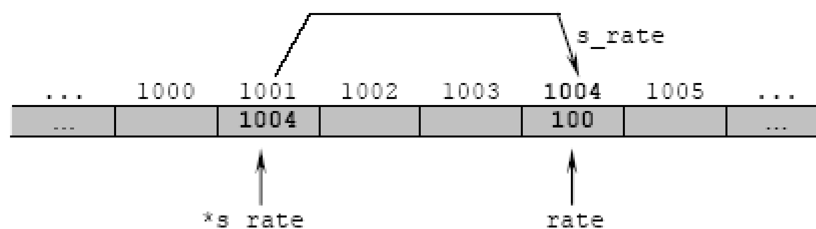


Figure 8.3

- In simplified form:

Where:

Figure 8.4

- So the declaration of the pointer variable becomes something like this:

```
int    *s_rate;
```

- In other word, the variable `s_rate` contains the memory address of the variable `rate` and is therefore a pointer to `rate`. The asterisk (*) is used to show that is it the pointer variable instead of normal variable.
- The **definition**: A pointer is a variable that contains the memory address of another variable, where, the actual data is stored.
- By using pointers, it is an efficient way of accessing and manipulating data. From user side implicitly, it contains the actual location of your data, where the compiler can find that data in memory.
- This is very useful for dynamic and efficient memory management programming as we know memory, registers and cache are scarce in computer system.
- Why it is very efficient way? One of the reasons is because for every computer system, it is provided with fixed and limited amount of memory for temporary data storage and processing.
- During the usage of computers, for example, the initialization of the Operating System and running programs (processes), there are loading and unloading data into and from the memory respectively, including the cache memory and the processors' registers for temporary data storage and processing. This loading and unloading need an efficient memory management.
- Furthermore some of the memory portion of computer system also used for shared data to further optimizes the utilization.
- Apart from primary memory, secondary memory also involved in this memory management through swapping and paging.
- Newer computer applications such as real time processing, graphic manipulations and complex mathematical calculations widely known for their memory intensive applications need storage efficiency to fully utilize the scarce computer memory.
- By using pointers, the usage of this fixed storage should be optimized.
- Yes, you can add extra memory module but without the usage 'optimization', it is not fully utilized somewhere, not sometime but most of the times :o).
- Familiar with the dead blue screen? Pointers also generate many bugs in programs if used improperly or maliciously :o).

8.2 Pointers And Simple Variables

- A pointer is a numeric variable and like other variables, as you have learned before, must be declared and initialized before it can be used.
- The following is a general form for declaring a pointer variable:

```
data_type    *pointer_variable_name;
```

- For example:

```
char*   x;
int   * type_of_car;
float   *value;
```

- `data_type` is any valid C/C++ type. It is the pointer base type such as `char`, `int` or `float`. It indicates the type of the variable's data to which the pointer points.
- `pointer_variable_name` follows the same rules as other variables naming convention and must be unique.
- From the above pointer declaration example, in the first line, the declaration tells us that `x` is a pointer to a variable of type `char`.
- The asterisk (*) is called **indirection operator**, and it indicates that `x` is a pointer to type `char` and not a normal variable of type `char`. Note the position of the *, it is valid for all the three positions.
- Pointers can be declared along with non pointer variables as shown below:

```
char   *ch1, *ch2;
// ch1 and ch2 both are pointers to type char.

float   *value, percent;
// value is a pointer to type float, and percent is
// an ordinary float variable.
```

8.3 Initializing Pointers

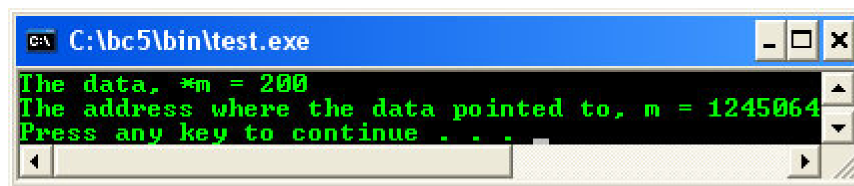
- Once a pointer is declared, the programmer must initialize the pointer, that is, make the pointer point to something. Don't make it point to nothing; it is dangerous.
- Like regular variables, uninitialized pointers will not cause a compiler error, but using an uninitialized pointer could result in unpredictable and potentially disastrous outcomes.
- Until pointer holds an address of a variable, it isn't useful.
- C/C++ uses two pointer operators:

    1. Indirection operator (`*`) – has been explained.
    2. Address-of-operator (`&`) – means **return the address of**.

- When `&` placed before the name of a variable, the address-of-operator returns the memory address of the variable/operand.
- For example:

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

int main(void)
{
   int *m;

   int location = 200;
   m = &location;

   printf("The data, *m = %d\n",*m);
   printf("The address where the data pointed to, m = %d\n", m);
   system("pause");
   return 0;
}
```

Output:



- Therefore a pointer must be initialized with a statement that uses `&` operator as shown below:

```
// declare a pointer variable, m of int type
int  *m;
// assign the address of variable location
// to variable m, so pointer m is pointing to
// variable location
m = &location;
// the actual data assigned to variable location
location = 200;
```

- This statement means places the memory address of variable `location` into variable `m`. The memory address refers to the computer's internal location where the actual data is stored. What and where the address, is determined by the system.
- In other word, pointer variable `m` receives the address of variable `location` or the memory address of the variable `location` is assigned to pointer variable `m`.
- Graphically:

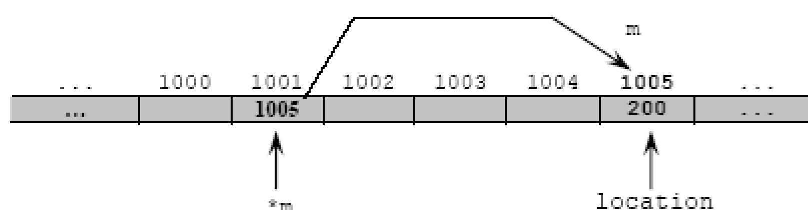


Figure 8.5

- Let re-examine the indirection operator (*). Actually * is a complement of &. It means returns the value of the variable located at the address that follows.
- From the previous example:

```
int  *m;
m = &location;
location = 200;
q = *m;
```

- The `q = *m;` statement will place the actual data value stored in variable `location` into variable `q`. That means `q` receives the actual data value stored at the memory address hold by variable `m` or the value stored in the variable `location` pointed to by variable `m`. Very confused huh?
- The * operator appears before a pointer variable in only two places:

 1. When declaring a pointer variable.
 2. When dereferencing a pointer variable (to find the data it points to).

8.4 Using Pointers

- Only the addition and subtraction operations are permitted in expression involving pointers.
- As with any variable, a pointer may be used on the right hand side of an assignment statement to assign its value to another pointer as shown in the following example.

```
// program to illustrate the basic use of pointers
//**********C++****************
#include <iostream>
// using C header in C++
#include <cstdlib>
using namespace std;

void main()
{
    int num = 10, *point_one, *point_two;
    //declares an integer variable and two pointers variables

    point_one = &num;
    //assigns the address of variable num to pointer point_one

    point_two = point_one;
    //assigns the (address) point_one to point_two

    cout<<"Pointers variables..."<<endl;
    cout<<"*point_one = "<<*point_one<<"\n";
    cout<<"*point_two = "<<*point_two<<"\n";

    cout<<"\nNormal variable..."<<endl;
    cout<<"num = "<<num<<"\n";

    cout<<"\n-Both pointer point_one and"<<"\n";
    cout<<"-point_two point to the same variable num."<<"\n";
    cout<<"-That is why, they have same value, 10."<<endl;
    // displays value 10 stored in num since point_one
    // and point_two now point to variable num
    system("pause");
}
```

Output:

```
"d:\testprogb\Debug\testprogb.exe"
Pointers variables...
*point_one = 10
*point_two = 10

Normal variable...
num = 10

-Both pointer point_one and
-point_two point to the same variable num.
-That is why, they have same value, 10.
Press any key to continue . . .
Press any key to continue
```

- The program example can be illustrated graphically as shown below. The memory address is arbitrarily chosen.

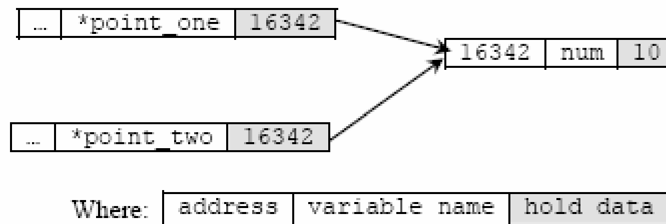


Figure 8.6

- Notice the difference between memory address and actual data value, which stored at the memory address. Do not worry about the addresses, they are determined and provided by the system.
- From the above example, we can:

1. Access the contents of a variable by using the variable name (`num`) and is called direct access.
2. Access the contents of a variable by using a pointer to the variable (`*point_one` or `*point_two`) and is called indirect access or indirection.

- As conclusion, if a pointer named `pter` of type `int` has been initialized to point to the variable named `var`, the following are true:

```
int  *pter;
pter = &var;
```

3. `*pter` and `var` both refer to the contents of `var` (that is, whatever data value stored there).
4. `pter` and `&var` refer to the address of `var` (the `pter` only hold the address of variable `var` not the actual data value).

- So, a pointer name without the indirection operator (*) accesses the pointer value itself, which is of course, the address of the variable pointed to. Very confused :o).
- Another example:

```
//Basic pointer use
#include <stdio.h>
#include <stdlib.h>

void main()
{
	//Declare and initialize an int variable
	int  var = 34;
	//Declare a pointer to int variable
	int  *ptr;

	//Initialize ptr to point to variable var
	ptr = &var;

	//Access var directly and indirectly
	printf("\nDirect access, variable var value = var = %d", var);
```

```
        //you can use %p for the pointer memory address directly or
        //%0x or %0X in hexadecimal representative instead of
        //%d, just to avoid confusion here…
        printf("\nIndirect access, variable var value = *ptr = %d", *ptr);

        //Display the address of var two ways
        printf("\n\nThe memory address of variable var = &var = %d", &var);
        printf("\nThe memory address of variable var = ptr = %d\n", ptr);
        system("pause");
}
```

Output:

```
C:\bc5\bin\proj0010.exe

Direct access, variable var value = var = 34
Indirect access, variable var value = *ptr = 34

The memory address of variable var = &var = 4094
The memory address of variable var = ptr = 4094
Press any key to continue . . .
```

- The address displayed for variable `var` may not be 4094 on your system because different computer will have different specification.
- For pointer arithmetic operation, only two arithmetic operations, that is addition and subtraction available. For example:

```
int age = 25;
```

- So, C/C++ reserved storage for the variable `age` and store the value 25 in it.
- Let say, C/C++ has stored this value at the memory address `1000`, and we declare a pointer variable named `ptr_age` that point to the variable `age`.
- Then after the following expressions have been executed:

```
int *ptr_age;
ptr_age = &age;
ptr_age++;
```

- The content of the pointer then becomes `1002`, not `1001` anymore (integer value takes two byte so C/C++ add 2 to the pointer).
- Different variable types occupy different amount of memory also depend on the platform used, whether 16, 32 or 64 bits. For example for 16 bits platform:

```
int = 2 byte.
float = 4 byte.
```

- Each time the pointer is incremented, it points to the next integer and similarly, when a pointer is decremented, it points to the previous integer.
- Each individual byte of memory has it own address; so multibyte variable actually occupies several addresses.
- When pointers used to handle the addresses of multibyte variables, the address of a variable is actually the address of the lowest byte it occupies.
- For example:

```
int     vint = 12252;
char    vchar = 90;
float   vfloat = 1200.156004;
```

- These variables are stored in memory as shown below.

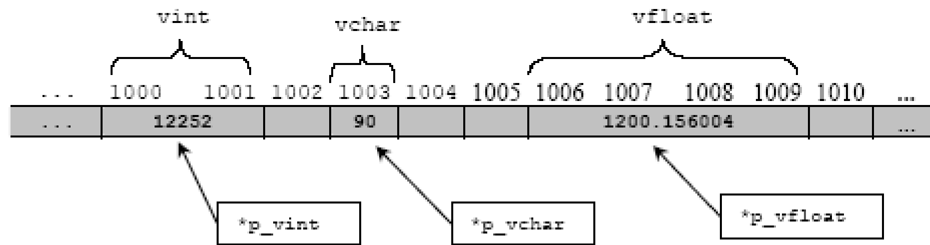


Figure 8.7

- So:

1. `int` variables occupy 2 byte starts at 1000.
2. `char` variables occupy 1 byte starts at 1003.
3. `float` variables occupy 4 byte starts at 1006.

- Then, how to declare and initialized pointers to these 3 variables? As explained to you before,
- Declaration - declare the pointer variables:

```
int   *p_vint;
char    *p_vchar;
float    *p_vfloat;
```

- Initialization - assign the addresses of the normal variables to the new pointer variables.

```
p_vint   =  &vint;
p_vchar   =  &vchar;
p_vfloat   =  &vfloat;
```

- Then pointer is equal to the address of the first byte of the pointed-to variable. So:

`p_vint` equals `1000`.
`p_vchar` equals `1003`.
`p_vfloat` equals `1006`.

- We have to understand this concept because we are dealing with addresses, not the actual data as before.
- Other pointer arithmetic operation is called differencing, which refers to subtracting 2 pointers.
- For example, two pointers that point to different elements of the same array can be subtracted to find out how far apart they are.
- Pointer arithmetic automatically scales the answer, so that it refers to the array elements.
- Thus, if `ptr1` and `ptr2` point to elements of an array (of any type), the following expression tells you how far apart the elements are:

```
ptr1 – ptr2;
```

8.5 Pointers Comparison

- The comparison is valid only between pointers that point to the same array.
- Under this circumstances, the following relational operators work for pointers operation.

`==`, `!=`, `>`, `<`, `>=`, and `<=`

- A lower array element that is those having a smaller subscript, always have a lower address than the higher array elements.
- Thus if `ptr1` and `ptr2` point to elements of the same array, the following comparison:

`ptr1 < ptr2`   is TRUE

- If `ptr1` points to an earlier member of the array than `ptr2` does.

- Many arithmetic operations that can be performed with regular variables, such as multiplication and division, do not work with pointers and will generate errors in C/C++.
- The following table is a summary of pointer operations.

Operation	Description
1. Assignment (=)	You can assign a value to a pointer. The value should be an address with the address-of-operator (&) or from a pointer constant (array name)
2. Indirection (*)	The indirection operator (*) gives the value stored in the pointed to location.
3. Address of (&)	You can use the address-of operator to find the address of a pointer, so you can use pointers to pointers.
4. Incrementing	You can add an integer to a pointer to point to a different memory location.
5. Differencing	You can subtract an integer from a pointer to point to a different memory location.
6. Comparison	Valid only with 2 pointers that point to the same array.

Table 8.1: Pointer operations

8.6 Uninitialized Pointers

- Let say we declare a pointer something like this:

```
int *ptr;
```

- This statement declares a pointer to type `int` but not yet initialized, so it doesn't point to anything known.
- Then, consider the following pointer assignment statement:

```
*ptr = 12;
// this is not an address lol!
```

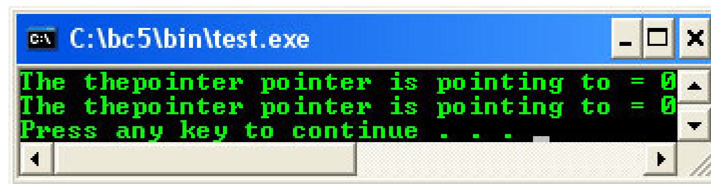
- This statement means the value 12 is assigned to whatever address `ptr` point to. That address can be almost anywhere in memory.
- The 12 stored in that location may overwrite some important information, and the result can be anything from storage program errors to a full system crash. So, you must initialize a pointer so that it point to something. Do not create a stray pointer.
- A better solution may be you can assign `NULL` (`\0`) value during the initialization before using the pointer, by pointing it to something useful or for pointer that point to a string you may just point to an empty string for dummy such as:

```
char * mystring = "";
```

- Program example:

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

int main()
{

int *thepointer;

thepointer = NULL;
//do some testing....
printf("The thepointer pointer is pointing to = %X\n", thepointer);
printf("The thepointer pointer is pointing to = %d\n", thepointer);
system("pause");
return 0;
}
```

Output:



8.7 Arrays And Pointers

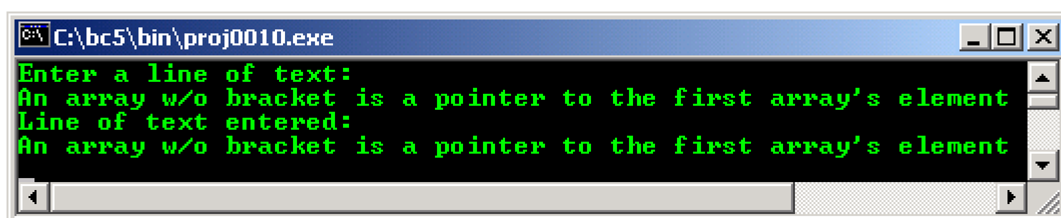
- A special relationship exists between pointers and arrays and this have been explained briefly in Module Array.
- An array name without brackets is a pointer to the array's first element. So, if a program declared an array `data[ ]`, `data` (array's name) is the address of the first array element and is equivalent to the expression `&data[0]` that means references the address of the array's first element.

`data` equivalent to `&data[0]` or a pointer to the array's first element.

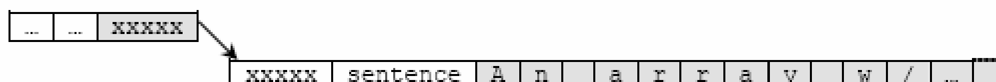
- The **array's name** is, therefore a pointer to the array's first element and therefore to the string if any.
- A pointer is a constant. It cannot be changed and remains fixed for the duration of program execution.
- Many string operations in C/C++ are typically performed by using pointers because strings tend to be accessed in a strictly sequential manner.
- Program example.

```
//Array, pointer and string
#include  <stdio.h>

void main()
{
        //an array variable of type char, sized 79
        char  sentence[80];

        //prompt for user input...
        printf("Enter a line of text:\n");
        //read the user input...
        gets(sentence);
        //display what has been read by gets()
        printf("Line of text entered: \n%s\n", sentence);
        getchar();
}
```

Output:



- Graphically can be depicted as follows:



Where: xxxxx is a memory address set by the system. Memory addresses used in this discussion just an arbitrary.

Figure 8.8

- Element of an array are stored in sequential memory locations with the first element in the lowest address.
- Subsequent array elements, those with an index greater than 0 are stored at higher addresses.

- As mentioned before, array of type `int` occupies 2 byte of memory and a type `float` occupies 4 byte.
- So, for `float` type, each element is located 4 bytes higher than the preceding element, and the address of each array element is 4 higher than the address of the preceding element.
- For example, relationship between array storage and addresses for a 6-elements int array and a 3-elements `float` array is illustrated below.

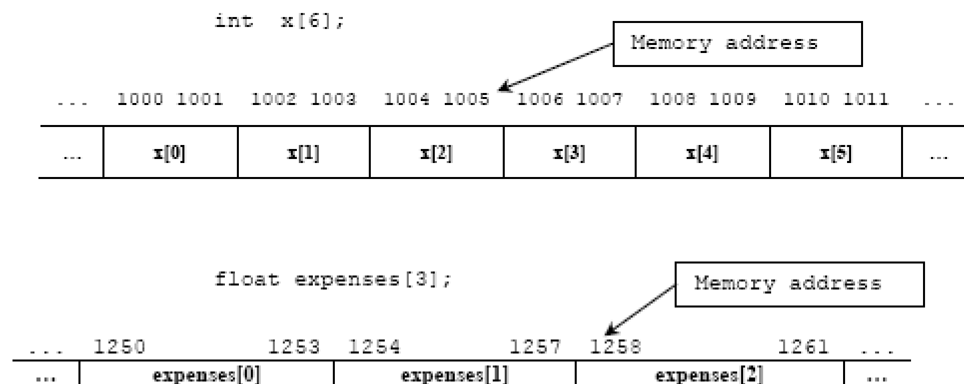
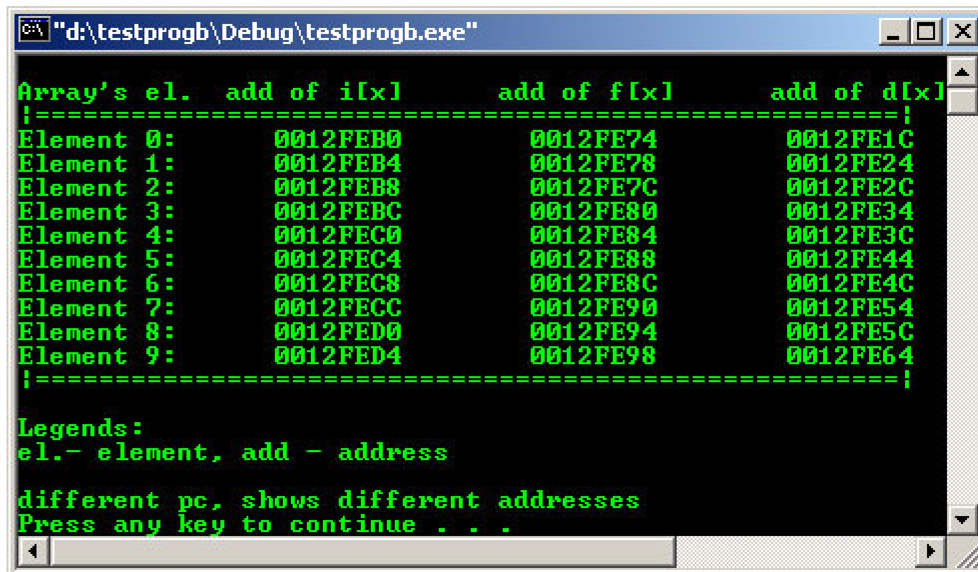


Figure 8.9

- The `x` variable without the array brackets is the address of the first element of the array, `x[0]`.
- The element is at address of `1000`; the second element is `1002` and so on.
- As conclusion, to access successive elements of an array of a particular data type, a pointer must be increased by the `sizeof(data_type)`. `sizeof()` function returns the size in bytes of a C/C++ data type. Let take a look at the following example:

```
//demonstrates the relationship between addresses
//and elements of arrays of different data type
#include <stdio.h>
#include <stdlib.h>

void main()
{
       //declare three arrays and a counter variable
       int  i[10], x;
       float  f[10];
       double  d[10];

       //print the table heading
       printf("\nArray's el.  add of i[x]      add of f[x]      add of d[x]");
       printf("\n|================================");
       printf("======================|");

       //print the addresses of each array element
       for(x=0; x<10; x++)
       printf("\nElement %d:\t%p\t%p\t%p",x,&i[x],&f[x],&d[x]);
       printf("\n|================================");
       printf("======================|\n");
       printf("\nLegends:");
       printf("\nel.- element, add - address\n");
       printf("\ndifferent pc, shows different addresses\n");
       system("pause");
 }
```

Output:



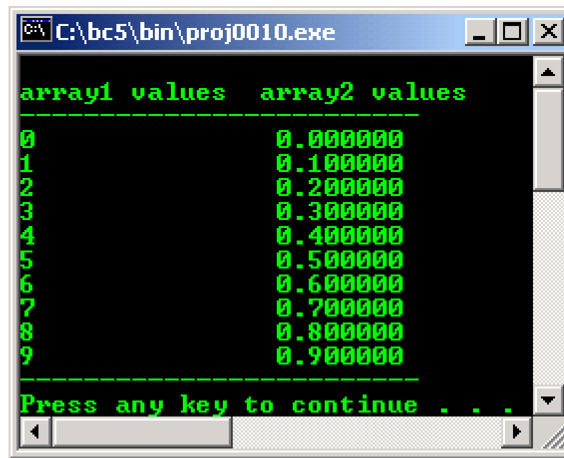
- Notice the difference between the element addresses.

  `12FEB4` – `12FEB0` = 4 bytes for `int`
  `12FE78` – `12FE74` = 4 bytes `float`
  `12FE24` – `12FE1C` = 8 bytes `double`

- The size of the data type depends on the specification of your compiler, whether your target is 16, 32 or 64 bits systems, the output of the program may be different for different PC. The addresses also may different.
- Try another program example.

```
//demonstrates the use of pointer arithmetic to access
//array elements with pointer notation
#include <stdio.h>
#include <stdlib.h>
#define   MAX  10

void main()
{
       //declare and initialize an integer array
       int array1[MAX] = {0,1,2,3,4,5,6,7,8,9};
       //declare a pointer to int and an int variable
       int *ptr1, count;
       //declare and initialize a float array
       float array2[MAX] =  {0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};
       //declare a pointer to float
       float *ptr2;

       //initialize the pointers
       //just an array name is the pointer to the
       //1st array element, both left value and right value
       //of the expression are pointers types...
       ptr1 = array1;
       ptr2 = array2;

       //print the array elements
       printf("\narray1 values  array2 values");
       printf("\n-------------------------");
       //iterate or loop the arrays and display the content...
       for(count = 0; count < MAX; count++)
              printf("\n%d\t\t%f", *ptr1++, *ptr2++);
       printf("\n-------------------------\n");
       system("pause");
    }
```

Output:



- Let make it clear, if an array named `list[]` is a declared array, the expression `*list` is the array's first element, `*(list + 1)` is the array's second element, and so on.
- Generally, the relationship is as follows:

```
*(list)  ==  list[0]        //first element
*(list + 1) == list[1]      //second element
*(list + 2) == list[2]      //third element
   …
   …
*(array + n) == list[n]     //the nth element
```

- So, you can see the equivalence of array subscript notation and array pointer notation.

8.8 Arrays Of Pointers

- Pointers may be arrayed like any other data type. The declaration for an `int` pointer array of size 20 is:

```
int *arrayPtr[20];
```

- To assign the address of an integer variables called `var` to the first element of the array, we could write something like this:

```
arrayPtr[0] = &var;
```

- Graphically can be depicted as follows:

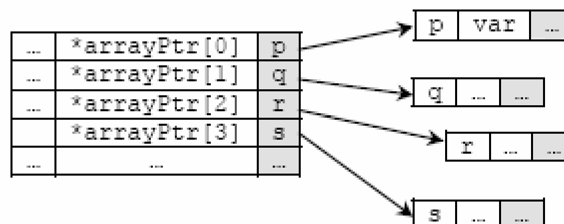


Figure 8.10

- To find the value stored in `var`, we could write something like this:

```
*arrayPtr[0]
```

- To pass an array of pointers to a function, we simply call the function with the **array's name** without any index/subscript, because this is automatically a pointer to the first element of the array, as explained before.
- For example, to pass the array named `arrayPtr` to `viewArray` function, we write the following statement:

```
viewArray(arrayPtr);
```

- The following program example demonstrates the passing of a pointer array to a function. It first declares and initializes the array variable `var` (not a pointer array).
- Then it assigns the address of each element (`var[i]`) to the corresponding pointer element (`arrayPtr[i]`).
- Next, the array `arrayPtr` is passed to the array's parameter `q` in the function `viewArray()`. The function displays the elements pointed to by `q` (that is the values of the elements in array `var`) and then passes control back to `main()`.

```
//program that passes a pointer array to a function
#include  <iostream.h>
#include  <stdlib.h>

//function prototype for viewArray
void viewArray(int  *[]);

void main()
{
        //declare and initialize the array variables...
        int i,*arrayPtr[7], var[7]={3,4,4,2,1,3,1};

        //loop through the array...
        for(i=0; i<7; i++)
        //arrayPtr[i] is assigned with
        //the address of var[i]
        arrayPtr[i] = &var[i];

        //A call to function viewArray,
        //pass along the pointer to the
        //1st array element
        viewArray(arrayPtr);
        cout<<endl;
        system("pause");
}

//function prototype...
//arrayPtr is now passed to parameter q,
//q[i] now points to var[i]
void viewArray(int *q[])
{
        int  j;
        for(j = 0; j < 7; j++)
                cout<<*q[j]<<" ";
        //displays the element var[i] pointed to by q[j]
        //followed by a space.  No value is returned
        //and control reverts to main()
}
```

Output:

8.9 Pointers To Pointers

- Graphically, the construct of a pointer to pointer can be depicted as shown below. `pointer_one` is the first pointer, pointing to the second pointer, `pointer_two` and finally `pointer_two` is pointing to a normal variable num that hold integer 10.

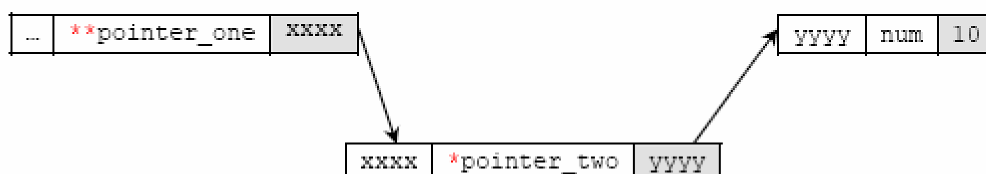


Figure 8.11

- Another explanation, from the following figure, a pointer to a variable (first figure) is a single indirection but if a pointer points to another pointer (the second figure), then we have a double or multiple indirections.

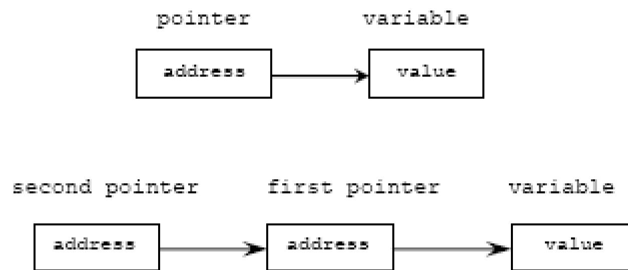


Figure 8.12

- For the second figure, the second pointer is not a pointer to an ordinary variable, but rather, a pointer to another pointer that points to an ordinary pointer.
- In other words, the second pointer points to the first pointer, which in turn points to the variable that contains the data value.
- In order to indirectly access the target value pointed to by a pointer to a pointer, the asterisk operator must be applied twice. For example, the following declaration:

```
int **SecondPtr;
```

- Tell the compiler that `SecondPtr` is a pointer to a pointer of type integer. Pointer to pointer is rarely used but you will find it regularly in programs that accept argument(s) from command line.
- Consider the following declarations:

```
char chs;           /* a normal character variable */
char *ptchs;        /* a pointer to a character */
char **ptptchs;     /* a pointer to a pointer to a character */
```

- If the variables are related as shown below:

Figure 8.13

- We can do some assignment like this:

```
chs = 'A';
ptpch = &chs;
ptptpch = ptchs;
```

- Recall that `char *` refers to a `NULL` terminated string. So one common way is to declare a pointer to a pointer to a string something like this:

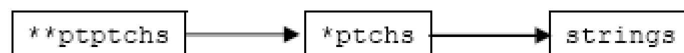


Figure 8.14

- Taking this one stage further we can have several strings being pointed to by the integer pointers (instead of char) as shown below.

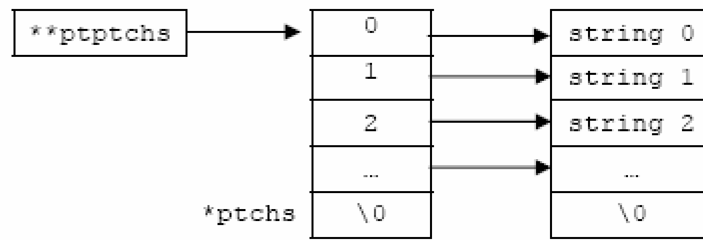


Figure 8.15

- Then, we can refer to the individual string by using `ptptchs[0]`, `ptptchs[1]`,…. and generally, this is identical to declaring:

```
char *ptptchs[] /* an array of pointer */
```

- Or from Figure 8.15:

```
char **ptptchs
```

- Thus, programs that accept argument(s) through command line, the `main()` parameter list is declared as follows:

```
int main(int argc, char **argv)
```

- Or something like this:

```
int main(int argc, char *argv[])
```

- Where the `argc` (argument counter) and `argv` (argument vector) are equivalent to `ptchs` and `ptptchs` respectively.
- For example, program that accept command line argument(s) such as `echo`:

```
C:\>echo This is command line argument
This is command line argument
```

- Here we have:

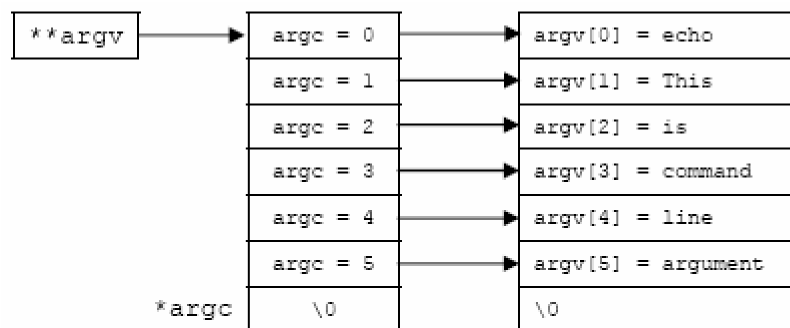


Figure 8.16

- So our `main()` function now has its own arguments. As a convention, these are the only arguments `main()` accepts. Notice that the `argv[0]` is program name.

1. `argc` (argument counter) is the number of arguments, which we typed including the program name that is equal to `0`.
2. `argv` (argument vector) is an array of strings holding each command line argument including the program name that is the first array element, `argv[0]`.

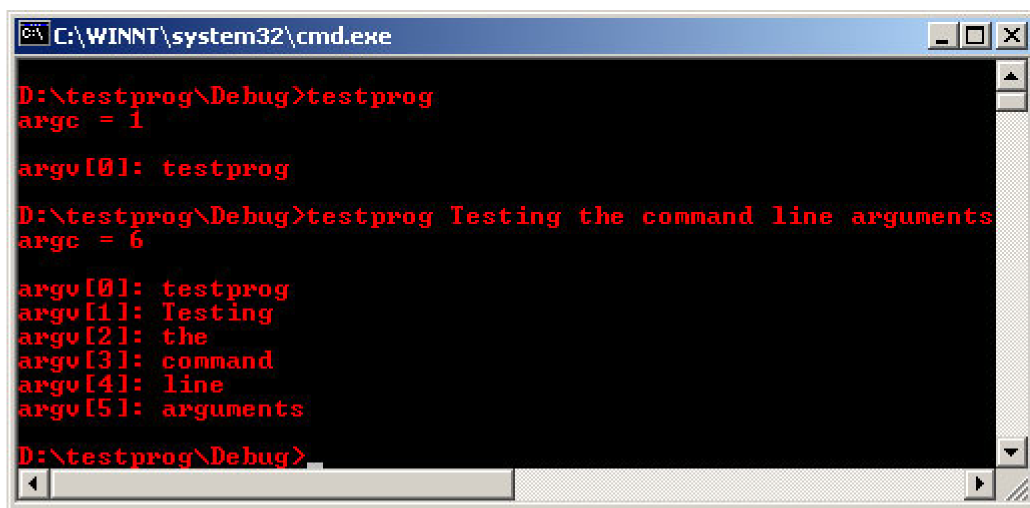
- But some implementation will have another third parameter, envp/env, a pointer to an environment variable as shown below. Environment variable in Operating System is an array of strings.

```
int main(int argc, char *argv[], *envp[])
```

- Let try a program example:

```
/*program to print arguments from command line*/
/*run this program at the command prompt*/
#include <stdio.h>

/*or int main(int argc, *argv[])*/
int main(int argc, char **argv)
{
        int i;
        printf("argc = %d\n\n", argc);
        for (i=0; i<argc; ++i)
        printf("argv[%d]: %s\n", i, argv[i]);
        return 0;
}
```

Output:



- Another silly program example :o):

```
// pointer to pointer...
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
int **theptr;
int *anotherptr;

int data = 200;
anotherptr = &data;

// assign the second pointer address to
// the first pointer...
theptr = &anotherptr;

printf("The actual data, **theptr = %d\n", **theptr);
printf("\nThe actual data, *anotherptr = %d\n", *anotherptr);
printf("\nThe first pointer pointing to an address, theptr = %p\n", theptr);
printf("\nThis should be the second pointer address, &anotherptr = %p\n", &anotherptr);
printf("\nThe second pointer pointing to address(= hold data),\nanotherptr = %p\n",
anotherptr);
printf("\nThen, its own address, &anotherptr = %p\n", &anotherptr);
printf("\nThe address of the actual data, &data = %p\n", &data);
printf("\nNormal variable, the data = %d\n", data);

system("pause");
return 0;
}
```

Output:

```
"d:\testprog\Debug\testprog.exe"
The actual data, **theptr = 200

The actual data, *anotherptr = 200

The first pointer pointing to an address, theptr = 0012FEC8

This should be the second pointer address, &anotherptr = 0012FEC8

The second pointer pointing to address(= hold data),
anotherptr = 0012FEBC

Then, its own address, &anotherptr = 0012FEC8

The address of the actual data, &data = 0012FEBC

Normal variable, the data = 200
Press any key to continue . . .
```

8.10 Pointers and Function – Function Pointers

- Because of C functions have addresses we can use pointers to point to C functions. If we know the function's address then we can point to it, which provides another way to invoke it.
- Function pointers are pointer variables which point to functions. Function pointers can be declared, assigned values and then used to access the functions they point to. The declaration is as the following:

```
int (*funptr)();
```

- Here, `funptr` is declared as a pointer to a function that returns `int` data type.
- The interpretation is the dereferenced value of funptr, that is `(*funptr)` followed by `()` which indicates a function, which returns integer data type.
- The parentheses are essential in the declarations because of the operators' precedence. The declaration without the parentheses as the following:

```
int *funptr();
```

- Will declare a function `funptr` that returns an integer pointer that is not our intention in this case. In C, the name of a function, used in an expression by itself, is a pointer to that function. For example, if a function, `testfun()` is declared as follows:

```
int testfun(int x);
```

- The name of this function, `testfun` is a pointer to that function. Then, we can assign them to pointer variable `funptr`, something like this:

```
funptr = testfun;
```

- The function can now be accessed or called, by dereferencing the function pointer:

```
/* calls testfun() with x as an argument then assign to the
variable y */
y = (*funptr)(x);
```

- Function pointers can be passed as parameters in function calls and can be returned as function values.
- Use of function pointers as parameters makes for flexible functions and programs. It's common to use typedefs with complicated types such as function pointers. You can use this typedef name to hide the cumbersome syntax of function pointers. For example, after defining

```
typedef int (*funptr)();
```

- The identifier `funptr` is now a synonym for the type of 'a pointer to function takes no arguments, returning `int` type'. This typedef would make declaring pointers such as `testvar` as shown below, considerably easier:

```
funptr testvar;
```

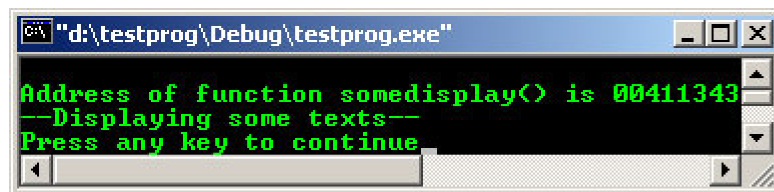
- Another example, you can use this type in a `sizeof()` expression or as a function parameter as shown below:

```
/* get the size of a function pointer */
unsigned ptrsize = sizeof (int (*funptr)());

/* used as a function parameter */
void signal(int (*funptr)());
```

- Let try a simple program example using function pointer.

```
/* Invoking function using function pointer */
#include <stdio.h>

int somedisplay();

int main()
{
    int (*func_ptr)();

    /* assigning a function to function pointer
    as normal variable assignment */
    func_ptr = somedisplay;
    /* checking the address of function */
    printf("\nAddress of function somedisplay() is %p", func_ptr);
    /* invokes the function somedisplay() */
    (*func_ptr)() ;
    return 0;
}

int somedisplay()
{
    printf("\n--Displaying some texts--\n");
    return 0;
}
```

Output:

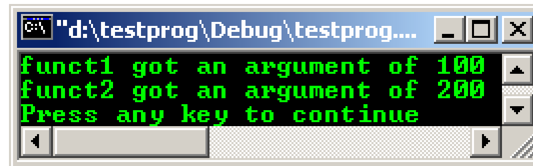


- Another example with an argument.

```
#include <stdio.h>

/* function prototypes */
void funct1(int);
void funct2(int);

/* making FuncType an alias for the type
'function with one int argument and no return value'.
This means the type of func_ptr is 'pointer to function
with one int argument and no return value'. */
typedef void FuncType(int);

int main(void)
{
    FuncType *func_ptr;

    /* put the address of funct1 into func_ptr */
    func_ptr = funct1;
    /* call the function pointed to by func_ptr with an argument of 100 */
```

```
    (*func_ptr)(100);

    /* put the address of funct2 into func_ptr */
    func_ptr = funct2;
    /* call the function pointed to by func_ptr with an argument of 200 */
    (*func_ptr)(200);
    return 0;
}

/* function definitions */
void funct1 (testarg)
{printf("funct1 got an argument of %d\n", testarg);}

void funct2 (testarg)
{printf("funct2 got an argument of %d\n", testarg);}
```

Output:



- The following codes in the program example:

```
    func_ptr = funct1;
    (*func_ptr)(100);
```

- Can also be written as:

```
    func_ptr = &funct1;
    (*func_ptr)(100);
```

- Or

```
    func_ptr = &funct1;
    func_ptr(100);
```

- Or

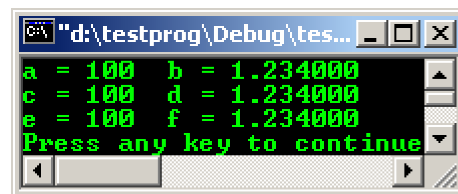
```
    func_ptr = funct1;
    func_ptr(100);
```

- As we have discussed before, we can have an array of pointers to an `int`, `float` and `string`. Similarly we can have an array of pointers to a function. It is illustrated in the following program example.

```
/* An array of pointers to function */
#include <stdio.h>

/* functions' prototypes */
int fun1(int, double);
int fun2(int, double);
int fun3(int, double);
/* an array of a function pointers */
int (*p[3]) (int, double);

int main()
{
    int i;

    /* assigning address of functions to array pointers */
    p[0] = fun1;
    p[1] = fun2;
    p[2] = fun3;

    /* calling an array of function pointers with arguments */
    for(i = 0; i <= 2; i++)
    (*p[i]) (100, 1.234);
    return 0;
```

```
}

/* functions' definition */
int fun1(int a, double b)
{
    printf("a = %d  b = %f", a, b);
    return 0;
}

int fun2(int c, double d)
{
    printf("\nc = %d  d = %f", c, d);
    return 0;
}

int fun3(int e, double f)
{
    printf("\ne = %d  f = %f\n", e, f);
    return 0;
}
```

Output:



- In the above program we take an array of pointers to function `int (*p[3]) (int, double)`. Then, we store the addresses of three function `fun1()`, `fun2()`, `fun3()` in array `(int *p[])`. In for loop we consecutively call each function using their addresses stored in array.
- For function and array, the only way an array can be passed to a function is by means of a pointer.
- Before this, an argument is a value that the calling program passes to a function. It can be `int`, a `float` or any other simple data type, but it has to be a single numerical value.
- The argument can, therefore, be a single array element, but it cannot be an entire array.
- If an entire array needs to be passed to a function, then you must use a pointer.
- As said before, a pointer to an array is a single numeric value (the address of the array's first element).
- Once the value of the pointer (memory address) is passed to the function, the function knows the address of the array and can access the array elements using pointer notation.
- Then how does the function know the size of the array whose address it was passed?
- Remember! The value passed to a function is a pointer to the first array element. It could be the first of 10 elements or the first of 10000 or what ever the array size.
- The method used for letting a function knows an array's size, is by passing the function the array size as a simple type int argument.
- Thus the function receives two arguments:

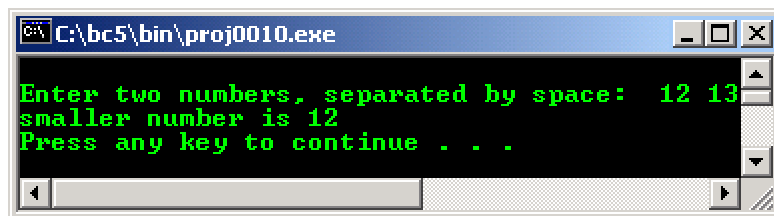
  1. A pointer to the first array element and
  2. An integer specifying the number of elements in the array, the array size.

- The following program example illustrates the use of a pointer to a function. It uses the function prototype `float (*ptr) (float, float)` to specify the number and types of arguments. The statement `ptr = &minimum` assigns the address of `minimum()` to `ptr`.
- The statement `small = (*ptr)(x1, x2);` calls the function pointed to by `(*ptr)`, that is the function `minimum()` which then returns the smaller of the two values.

```
// pointer to a function
#include <iostream.h>
#include <stdlib.h>

// function prototypes...
float minimum(float, float);
// (*ptr) is a pointer to function of type float
float (*ptr)(float, float);

void main()
{
   float x1, x2, small;
```

```
    // Assigning address of minimum() function to ptr
    ptr = minimum;

    cout<<"\nEnter two numbers, separated by space:  ";
    cin>>x1>>x2;

    // call the function pointed by ptr small
    // has the return value
    small = (*ptr)(x1, x2);
    cout<<"\smaller number is "<<small<<endl;
    system("pause");
}

float minimum(float y1, float y2)
{
    if (y1 < y2)
     return  y1;
    else
     return y2;
}
```

Output:



- Study the program and the output.

8.11 `NULL` Pointer Constant

- The null pointer constant is guaranteed not to point to any real object. You can assign it to any pointer variable since it has type `void *`. The preferred way to write a null pointer constant is with `NULL`.
- This `NULL` pointer when set, actually pointing to an address of `0x00000000` (32 bits) of memory. The computer systems have reserved this zero address for `NULL` pointer and that is why it cannot be used for other purposes.
- Since C functions can only return one variable, it is common practice for those which return a pointer to set it to `NULL` if there's a problem or no object pointed to yet so that we can avoid the stray pointer. To play safe, you can also set a pointer to `NULL` to indicate that it's no longer in use.
- Program example segment is illustrated below:

```
/* declare a pointer to type char */
char *bufptr;
/* allocate memory on the heap */
bufptr = malloc(1000);

/* verify the memory allocation
if fail… */
if (bufptr == NULL)
{
  printf("Memory allocation is failed lol!\n");
}
/*If OK….*/
else
{
  /* call the LoadBuffer() function */
  LoadBuffer(bufptr);
}
...
/* free up memory on the heap */
free(bufptr);
bufptr = NULL;
```

- You can also use `0` or `(void *)0` as a null pointer constant, but using `NULL` is cleaner because it makes the purpose of the constant more evident.

8.12 `void` Pointers

- There are times when you write a function but do not know the data type of the returned value. When this is the case, you can use a `void` pointer, a pointer of type `void`.

```
#include <stdio.h>

/* void pointer */
int func(void *thePtr);

int main()
{
    /* assigning a string to the pointer */
    char *theStr = "abcd1234";
    func(theStr);
    return 0;
}

int func(void *thePtr)
{
    printf("%s\n", thePtr);
    return 0;
}
```

Output:

- A pointer can only be copied from a pointer of the same type, or from a pointer to `void`. A pointer to `void` is a generic pointer that can be used to hold an address, but cannot be 'deferenced', that means you can't use it with an asterisk before it to update a variable. It acts as a placeholder.
- `void` pointers are commonly used in the parameter list of built-in functions, such as `memcpy()` and `memset()`. This is why you can use the same function to copy strings (pointer to character) and structures (pointer to structure).
- On the other hand, when neither pointer is of type pointer to `void`, the only way to copy an address from a pointer of a different type is to cast the second pointer to the type of the first pointer as shown below:

```
char *bufptr;
struct record *recptr;

recptr = (struct record *)bufptr;
```

---Have a break!---

Reference/Dereference Operators Revisited

Function

Passes arguments in a function definition header by reference.

Reference Syntax

```
function_name (&parameter, ..., ...)
{statements}
```

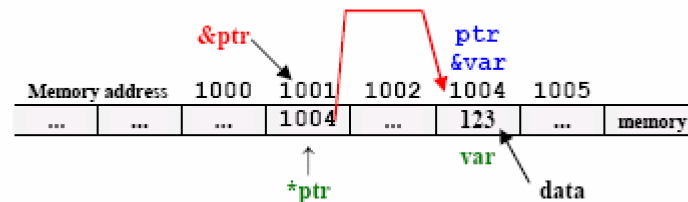
Some description

The default function calling convention is to pass by value. The reference operator can be applied to parameters in a function definition header to **pass the argument by reference** instead.
The reference types created with the `&` operator, create aliases for objects and let you pass arguments to functions by reference.
When a variable x is passed by reference to a function, the matching formal argument in the function receives an alias for x. Any changes to this alias in the function body are reflected in the value of x.

When a variable x is passed by value to a function, the matching formal argument in the function receives a **copy** of x. Any changes to this copy within the function body are not reflected in the value of x itself. The function also can return a value that could be used later to change x, but the function cannot directly alter a parameter passed by value. The reference operator is only valid when used in function definitions as applied to one or more of its parameters. It can be used to obtain the address of a variable.

Function And Pointer

Study the following illustration and memorize it, you will remember what the pointer and reference are, forever :o).



`*ptr` - pointer variable
`var` - normal variable

```
//declare a pointer variable ptr of type int
int *ptr;

//declare and initialize normal variable var of type int
int var = 123;

//assign the address of normal variable var to pointer ptr.
ptr = &var;
```

So, the pointer `ptr` now is pointing to the data value `123`, hold by variable `var`.
The `&` and `*` operators work together to reference and dereference pointers that are passed to functions.

Syntax

`&expression` e.g. `&var`
`*expression` e.g. `*ptr`

Referencing operator (&)

Use the reference operator to pass the address of a pointer to a function. The expression operand must be one of the following:

- A function designator - an lvalue designating an object that is not a bit field and is not declared with a register storage
- Class specifier

If the operand is of type `type`, the result is of type pointer to `type`. Consider the following generic example:

```
type varA = 1, varB = 2;

type *ptr = &varA;        //Initialized pointer

*ptr = varB;              //Same as varA = varB
```

`type *ptr = &varA` is treated as

```
type *ptr;

ptr = &varA;
```

So it is `ptr`, or `*ptr`, that gets assigned. Once `ptr` has been initialized with the address `&varA`, it can be safely dereferenced to give the lvalue `*ptr`.

Indirection operator (*)

Use the asterisk (*) in a variable expression to create pointers. And use the indirect operator in external functions to get a pointer's value that was passed by reference. If the operand is of type pointer to function, the result is a function designator.
If the operand is a pointer to an object, the result is an lvalue designating that object.
The result of indirection is undefined if either of the following occurs:

1. The expression is a null pointer.
2. The expression is the address of an automatic variable and execution of its block has terminated.

Other pointer declarations that you may find and can make you confused **:o)** are listed below.

Pointer declaration	Description
`int *x`	`x` is a pointer to `int` data type.
`int *x[10]`	`x` is an `array[10]` of pointer to `int` data type.
`int *(x[10])`	`x` is an `array[10]` of pointer to `int` data type.
`int **x`	`x` is a pointer to a pointer to an `int` data type – double pointers.
`int (*x)[10]`	`x` is a pointer to an `array[10]` of `int` data type.
`int *funct()`	`funct` is a function returning an integer pointer.
`int (*funct)()`	`funct` is a pointer to a function returning `int` data type – quite familiar constructs.
`int (*(*funct())[10])()`	`funct` is a function returning pointer to an `array[10]` of pointers to functions returning int.
`int (*(*x[4])())[5]`	`x` is an `array[4]` of pointers to functions returning pointers to `array[5]` of `int`.

And something to remember:

`*`	- is a pointer to
`[ ]`	- is an array of
`( )`	- is a function returning
`&`	- is an address of

---Have a break!---

Program Examples

Example#1

```
//Program that changes the value of a pointer variable
#include <iostream>
using namespace std;

void main()
{
        //declare and initialize two
        //float variables
        float var1 = 58.98;
        float var2 = 70.44;
        //declare a float pointer variable
        float *ptr_var;

        //make ptr_var point to variable var1...
        ptr_var = &var1;

        //prints 58.98
        cout<<"\nThe first value is(var1) "<<*ptr_var;
        cout<<"\nThe address of the first data is "<<ptr_var<<"\n";
        cout<<"\nThen let the same pointer (*ptr_var)";
        cout<<"\npoint to other address...\n";

        //make ptr_var point to variable var2...
        ptr_var = &var2;

        //prints 70.44
```

```
        cout<<"\nThe second value is(var2) "<<*ptr_var;
        cout<<"\nThe address of the second data is "<<ptr_var<<endl;
}
```

Output:

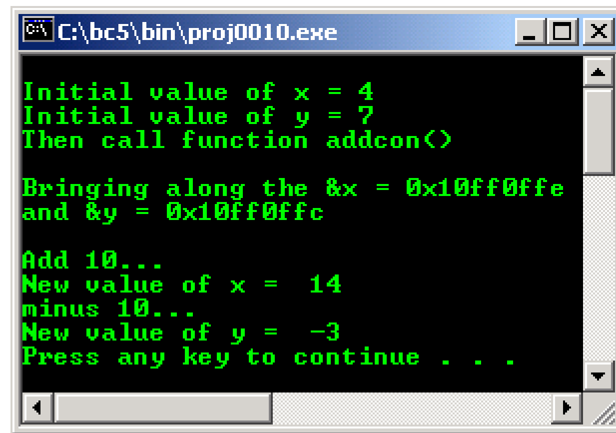
```
C:\bc5\bin\proj0010.exe

The first value is(var1) 58.98
The address of the first data is 0x12b20ffc

Then let the same pointer (*ptr_var)
point to other address...

The second value is(var2) 70.44
The address of the second data is 0x12b20ff8
Press any key to continue . . .
```

- This program demonstrates how you can make a pointer point to different values in memory. The program defines two floating-point values.
- A floating-point pointer points to the first variable `var1` and is used in the `cout` statement. The same pointer is then changed to point to the second variable `var2`.

Example#2

```
//Illustrates that function receives addresses
//of variables and then alters their contents
#include <iostream>
using namespace std;

void main()
{
        int  x = 4, y = 7;
        //function prototype...
        void addcon(int*, int*);

        cout<<"\nInitial value of x = "<<x;
        cout<<"\nInitial value of y = "<<y;
        cout<<"\nThen calls function addcon()\n";
        cout<<"\nBringing along the &x = "<<&x<<endl;
        cout<<"and &y = "<<&y<<"\n";
        cout;

        //function call, address of x any y are passed to addcon()
        addcon(&x, &y);

        cout<<"\nAdd 10...";
        cout<<"\nNew value of x =  "<<x;
        cout<<"\nminus 10...";
        cout<<"\nNew value of y =  "<<y<<endl;
}

//function definition
//parameters are pointers...
void addcon(int *px, int *py)
{
        //Adds 10 to the data stored in memory pointed to by px
        *px = *px + 10;
        //minus 10 to the data stored in memory pointed to by py
        *py = *py - 10;
}
```

Output:



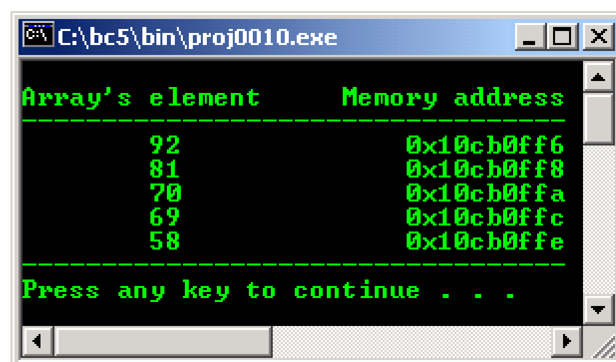
- The program illustrates the use of pointers. The function `main()` invokes the function `addcon()` with two arguments of type pointer, the first one gives the address of variable `x` and the second the address of variable `y`.
- The `addcon()` function then uses the addresses received from the function `main()` to reference the value of `x` and `y`. It then increments the values of x by 10 and decrement the value of y by 10 and restore the results. After execution, x will have the value 14 and y the value -3.

Example#3

- This program outputs the values in the array `nums`, using a pointer rather than index/subscript. In the first loop iteration, `*(nums + dex)` refers to the first value (given by index `0`) in the array `nums` `(that is 92);` in the second iteration, it refers to the second value (given by index `1`) in the array (that is `81`), and so on.

```
//A program that uses pointers to print
//the values of an array
#include <iostream>
using namespace std;

void main()
{
	//declare and initialize an array nums
	int nums[] = {92,81,70,69,58};

	cout<<"\nArray's element     Memory address";
	cout<<"\n----------------------------------";

	//using for loop, displays the elements
	//of nums and their respective memory address
	for(int dex=0; dex<5; dex++)
	cout<<"\n\t"<<*(nums + dex)<<"\t\t"<<(nums + dex);
	cout<<"\n----------------------------------\n";
}
```

Output:



Example#4

- This program first reads an arbitrary number of temperature readings into the array `temper` using pointers (rather than subscripts).
- Then, again using pointers, it computes the average temperature. The program terminates when the value entered for temperature is 0.

```
//To compute the average of an arbitrary number
//of temperature readings
#include <iostream>
using namespace std;

void main()
{
	float temper[40], sum = 0.0, *ptr;
	int   num, day = 0;

	//set a pointer to an array...
	ptr = temper;
	do
	{
		cout<<"Enter temperature for day "<<++day;
		//prompt for input user input...
		cout<<"\n(0-Terminate, Enter-Proceed):  ";
		//store in an array, pointed by ptr...
		cin>>*ptr;
	} while ((*ptr++) > 0);
	//Test if data entered is 0,
	//then point to the next array position

	//reset the pointer ptr to an array temper
	ptr = temper;

	num = (day - 1);
	//looping through the array temper...
	for(day = 0; day < num; day++)
		//do the summing up...
		sum += *(ptr++);
	//display the result...
	cout<<"\nAverage temperature =  "<<(sum/num)<<" Degree Celsius"<<endl;
	cout<<"for "<<num<<" readings"<<endl;
}
```

Output:

```
C:\bc5\bin\proj0010.exe
Enter temperature for day 1
(0-Terminate, Enter-Proceed):  85
Enter temperature for day 2
(0-Terminate, Enter-Proceed):  88
Enter temperature for day 3
(0-Terminate, Enter-Proceed):  90
Enter temperature for day 4
(0-Terminate, Enter-Proceed):  95
Enter temperature for day 5
(0-Terminate, Enter-Proceed):  86
Enter temperature for day 6
(0-Terminate, Enter-Proceed):  0

Average temperature =  88.8 Degree Celcius
for 5 readings
Press any key to continue . . .
```

Example#5

```
//using subscript and pointer notations with arrays
#include <stdio.h>
#include <stdlib.h>

void main()
{
	int   i, offset, b[] = {10, 20, 30, 40};
	//set bPtr to point to array b
	int   *bPtr = b;
```

```
        printf("So many notations?????....\n");
        //....separating code in multiple lines
        printf("Array b printed with: \n"
               "Array subscript notation\n");

        for(i=0; i<=3; i++)
               printf("b[%d] = %d\n", i, b[i]);
        printf("\nPointer/offset notation where \n"
                      "the pointer is the array name\n");

        for(offset = 0; offset <=3; offset++)
               printf("*(b + %d) = %d\n", offset, *(b + offset));
        printf("\nPointer subscript notation\n");

        for(i=0; i<=3; i++)
               printf("bPtr[%d] = %d\n",i,bPtr[i]);
        printf("\nPointer/offset notation\n");

        for(offset = 0; offset <=3; offset++)
               printf("*(bptr + %d) = %d\n", offset, *(bPtr + offset));
        system("pause");
}
```

Output:

```
C:\bc5\bin\proj0010.exe
So many notations?????....
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bptr + 0) = 10
*(bptr + 1) = 20
*(bptr + 2) = 30
*(bptr + 3) = 40
Press any key to continue . . .
```

- Program example compiled using VC++/VC++ .Net.

```
//Program that changes the value of a pointer variable
//compiled using VC++/VC++ .Net, C++ codes…
#include <iostream>
using namespace std;

void main()
{
   //declare and initialize two
   //float variables
   double var1 = 58.98;
   double var2 = 70.44;
   //declare a float pointer variable
   double *ptr_var;

   //make ptr_var point to variable var1...
   ptr_var = &var1;

   //prints 58.98
   cout<<"The first value is(var1) "<<*ptr_var;
```

```
        cout<<"\nThe address of the first data is "<<ptr_var<<"\n";
        cout<<"\nThen let the same pointer (*ptr_var)";
        cout<<"\npoint to other address...\n";

        //make ptr_var point to variable var2...
        ptr_var = &var2;

        //prints 70.44
        cout<<"\nThe second value is(var2) "<<*ptr_var;
        cout<<"\nThe address of the second data is "<<ptr_var<<endl;
}
```

Output:

```
"g:\vcnetprojek\searchpattern\Debug\sea...
The first value is(var1) 58.98
The address of the first data is 0012FED0

Then let the same pointer (*ptr_var)
point to other address...

The second value is(var2) 70.44
The address of the second data is 0012FEC0
Press any key to continue
```

- Previous C program example of function pointer, compiled using **gcc**.

```
/********* myptr.c *************/
#include <stdio.h>

int main()
{
        int   i, offset, b[] = {10, 20, 30, 40};
        //set bPtr to point to array b
        int   *bPtr = b;

        printf("So many notations?????....\n");
        //....separating code in multiple lines
        printf("Array b printed with: \n"
                "Array subscript notation\n");

        for(i=0; i<=3; i++)
                printf("b[%d] = %d\n", i, b[i]);
        printf("\nPointer/offset notation where \n"
                        "the pointer is the array name\n");

        for(offset = 0; offset <=3; offset++)
                printf("*(b + %d) = %d\n", offset, *(b + offset));
        printf("\nPointer subscript notation\n");

        for(i=0; i<=3; i++)
                printf("bPtr[%d] = %d\n", i, bPtr[i]);
        printf("\nPointer/offset notation\n");

        for(offset = 0; offset <=3; offset++)
                printf("*(bptr + %d) = %d\n", offset, *(bPtr + offset));
        return 0;
```

```
 [bodo@bakawali ~]$ gcc myptr.c -o myptr
[bodo@bakawali ~]$ ./myptr

So many notations?????....
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
```

```
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bptr + 0) = 10
*(bptr + 1) = 20
*(bptr + 2) = 30
*(bptr + 3) = 40
```

- Previous C++ program example compiled using **g++**.

```
////////////////funcref.cpp////////////////
//Illustrates that function receives addresses
//of variables and then alters their contents
#include <iostream>
using namespace std;

int main()
{
        int x = 4, y = 7;
        //function prototype...
        void addcon(int*, int*);

        cout<<"\nInitial value of x = "<<x;
        cout<<"\nInitial value of y = "<<y;
        cout<<"\nThen calls function addcon()\n";
        cout<<"\nBringing along the &x = "<<&x<<endl;
        cout<<"and &y = "<<&y<<"\n";
        cout;

        //function call, address of x any y are passed to addcon()
        addcon(&x, &y);

        cout<<"\nAdd 10...";
        cout<<"\nNew value of x =  "<<x;
        cout<<"\nminus 10...";
        cout<<"\nNew value of y =  "<<y<<endl;
        return 0;
}

//function definition
//parameters are pointers...
void addcon(int *px, int *py)
{
        //Adds 10 to the data stored in memory pointed to by px
        *px = *px + 10;
        //minus 10 to the data stored in memory pointed to by py
        *py = *py - 10;
}
```

```
[bodo@bakawali ~]$ g++ funcref.cpp -o funcref
[bodo@bakawali ~]$ ./funcref

Initial value of x = 4
Initial value of y = 7
Then calls function addcon()

Bringing along the &x = 0xbfed7944
and &y = 0xbfed7940

Add 10...
New value of x =  14
minus 10...
New value of y =  -3
```

---o0o ---

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.