

CS 2 Lecture Topic 4 : Pointers
Author: Satish Singhal Ph. D
Version 1.0

Table of Contents

Introduction.....	1
Address (&) Operator:.....	2
Syntax of Declaring Pointer Variables:.....	3
Initialization of Pointer Variables:	5
Assigning Pointer to Existing Program Variables or Making pointers point to “pointees”:	5
Using Pointers to Access the Program variables:	6
Dereferencing a pointer, which has no pointee, will crash your code!!.....	8
C++11 nullptr	11
A Point of Confusion and Clarification	12
Post increment or decrement of a variable using its pointer	15
Assigning More than one pointer to a Variable or Constant.....	17
Versatile void pointer.....	17
Pointers to other pointers.....	23
Summary points about pointer variables:.....	25
Relationship of arrays and pointers	25
Constant Pointers or Restricted Pointers.....	27
1.Constant Pointer to Program Constants/Variables:(Read-Only –Access and Non-reassignable pointers)	27
Conclusions about Constant Pointers To Program Constants:	29
2. Pointers to Program constants/variables (Still Read-Only Access, but Re- Assignable)	32
3. Constant Pointers (Have Read and write access, but not re-assignable)......	33
Pointer Arithmetic	35
Review Questions	42

Introduction

Every C++ program has certain number of program variables, which, during the program execution, are assigned places in the memory. Each memory location has an address. Thus, each program variable and constant in a C++ program has an address in the memory. Pointer data types are capable of storing the address or the location in the memory, for program variables and named constants. We are already aware of one kind of pointer datum in C++. It is the name of an array. As we have emphasized in past, the name of an array is a constant pointer to its first element. While passing arrays to functions, the efficient memory use in C++ rests in the fact that only the pointer to array (its name) is passed to during a function call. This saves program from making copies large data stored in an array, thereby minimizing the memory usage. Use of pointers in C++ creates highly memory efficient programs, bordering on the memory utilization as low as an assembly language program. Pointers can store addresses of pre-existing program variable.

However, an excellent use of pointers is made, when they are used to create program variables dynamically, at the run time. Such dynamically created variables and data structures are accessible, only through the pointers assigned to them. We discuss these dynamic pointers after discussing pointer basics.

Address (&) Operator:

As discussed above, every program variable has a memory location, with certain address. Using the address operator (&) the programmer can print as to what address has been assigned to the variable in the memory. Listing 4.1 below shows the procedure of printing the address of a pre-defined program variable or a constant.

```
//Listing 4.1
#include <iostream>
using namespace std;

int main()
{
    char my_char = char();
    short my_short = short();
    int my_int = int();
    float my_float = float();
    double my_double = double();
    const int your_int = int();

    cout<< "The address of my_char = "<<(long)&my_char<<endl;
    cout<< "The address of my_short = "<<(long) &my_short<<endl;
    cout<< "The address of my_int = "<<(long)&my_int<<endl;
    cout<< "The address of my_float = "<<(long)&my_float<<endl;
    cout<< "The address of my_double = "<<(long)&my_double<<endl;
    cout<< "The address of constant your_int = "
        <<(long)&your_int<<endl;
    cout<< "The address of \"Hello World\" is = "
        <<(long)&"Hello World"<<endl;

    return 0;
}
```

```

The address of my_char = 6684148
The address of my_short = 6684144
The address of my_int = 6684140
The address of my_float = 6684136
The address of my_double = 6684128
The address of constant your_int = 6684124
The address of "Hello World" is = 4354088

```

Listing 4.1

From above listing, you would notice, that even a literal string created on the fly, as “Hello World” has an address in the memory. Figure 4.1 shows, schematically, as to how, at the birth of a variable, a memory address is created for them.

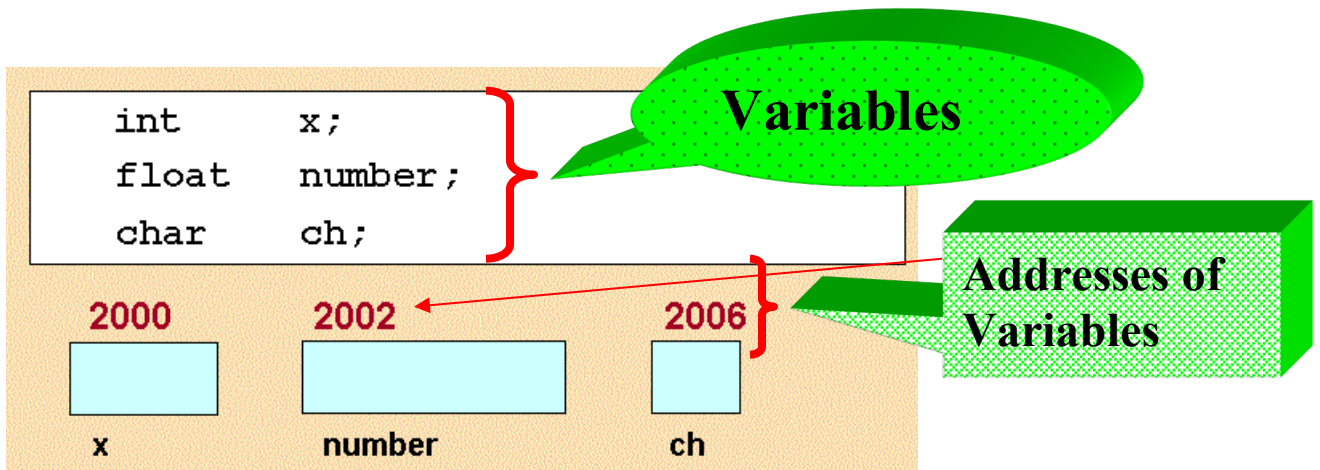


FIG. 4.1

Syntax of Declaring Pointer Variables:

Using certain data type reserved word and placing an asterisk next to it declare the pointers to that data type. Figure 4.2 shows the concept and program statements, needed to declare pointer variables.

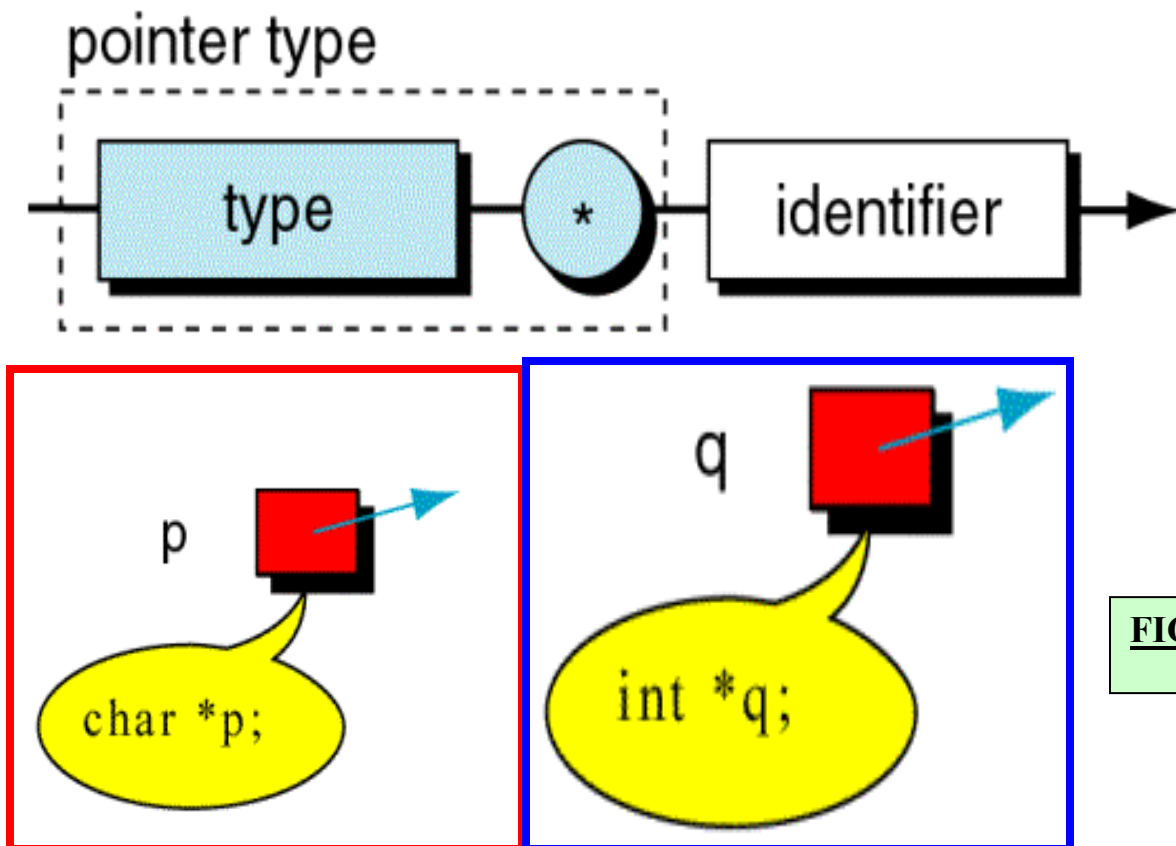


FIG. 4.2

First one declares the data type for which a pointer is to be declared, and then a asterisk is placed between the data type and the name of the pointer variable. The following pointer declarations are equivalent:

```
int* iptr;
int * iptr1;
int *iptr2;
```

The asterisk (*) may be attached to the data type or the variable name, or it may be placed exactly in the middle. A better practice is to put the asterisk (*) exactly in the middle so that data type and pointer variable names are visible unambiguously. Another practice common in C programming is to declare pointer and normal variables of the same type in the same line. For example consider this declaration:

```
int* iptr, val, val2, val3;
```

In above declaration it may seem like that all the variables iptr, val, val2 and val3 are pointer type. Unfortunately that is not true. Only the first variable iptr is pointer type as asterisk (*) after int applies only to iptr. The val, val2, and val3 are all normal int variables. To avoid such confusion in a multiple variable declaration

line it is best to attach the asterisk (*) to the variable name. Thus the above declaration is less confusing if it is modified as follows:

```
int *iptr, val, val2, val3;
```

Initialization of Pointer Variables:

The manner in which we have declared pointers in Figure 4.2 creates pointer variables, which contains unknown or garbage values stored in them. In other words we can say that pointers we have declared so far do not have “pointees” or they do not point to anything. **Pointers not having “pointees” are called dangling pointers. Dangling pointers, at times become major source of software bugs. Therefore it is essential that pointer variables be also initialized as soon as they are created.** Assigning a null value to it, which can be accomplished by any of the following three statements, can initialize pointer variables.

```
int * int_ptr = NULL;
```

```
float * float_ptr = 0;
```

```
char * char_ptr = '\0';
```

Assigning Pointer to Existing Program Variables or Making pointers point to “pointees”:

The address of(&) operator is used to assign pointers to store the address of program variables. Figure 4.3 shows the process of assigning pointer ptr to hold the address of int variable x.

Using a Pointer Variable

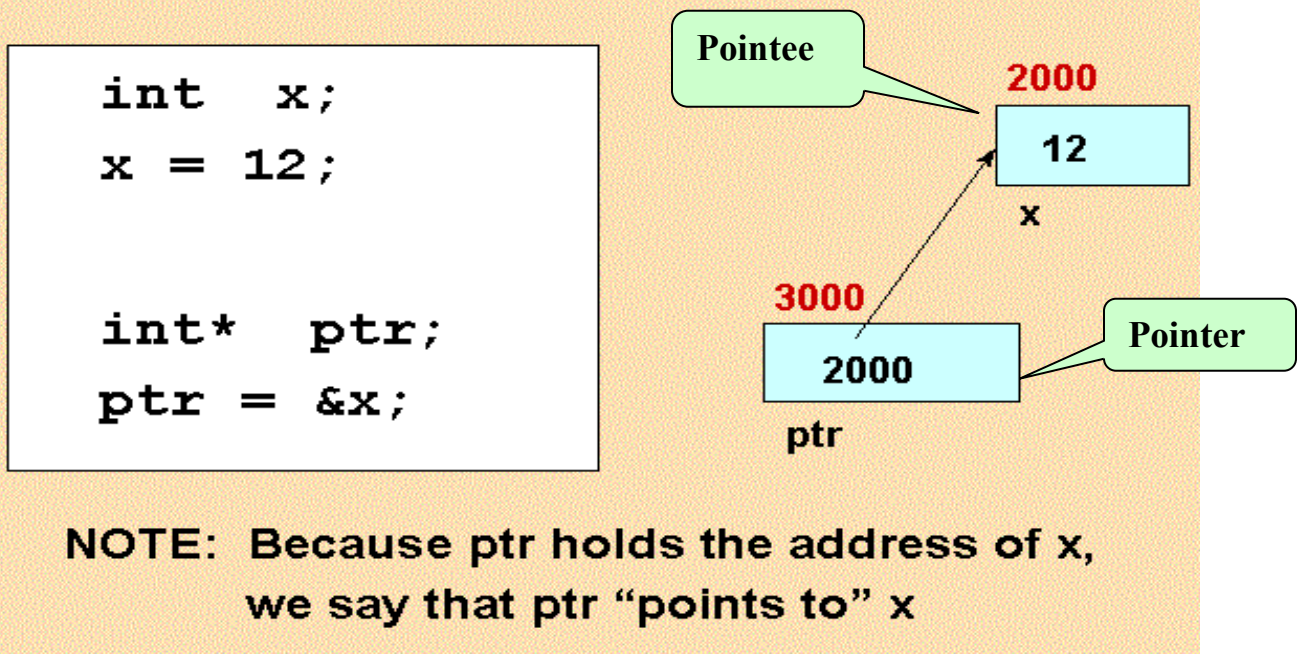


FIG. 4.3 [Taken from Prof. Nell Dale]

The last statement in the Figure 4.3 acts to store the address of int variable `x`, into the int pointer `ptr`. If the memory address of `x` was 2000, then a value of 2000 is stored in the memory assigned to pointer `ptr`. Note that pointer `ptr` will also have its own memory address, which is shown as 3000 in Figure 4.3. The declaration of a pointer and assignment to it can also be done in a single line. For example the last two lines of code in the Figure 4.3 can be condensed into one line as follows:

```
int * ptr = &x; // Now the pointer ptr stores the  
//address of pointee x
```

Using Pointers to Access the Program variables:

Once assigned to a program variable, its pointer can almost be used like another name for the variable. The pointers can be used to:

- Print the value of the variable it points to,
- Change the value of variable it points to,
- Participate in expressions involving program variable.

Using the pointer variable in above manners requires use of unary operator (*), which is also called as indirection or de-reference operator. The Figure 4.4 shows the use of indirection or de-reference operator to print the value of a program variable to which a pointer was assigned.

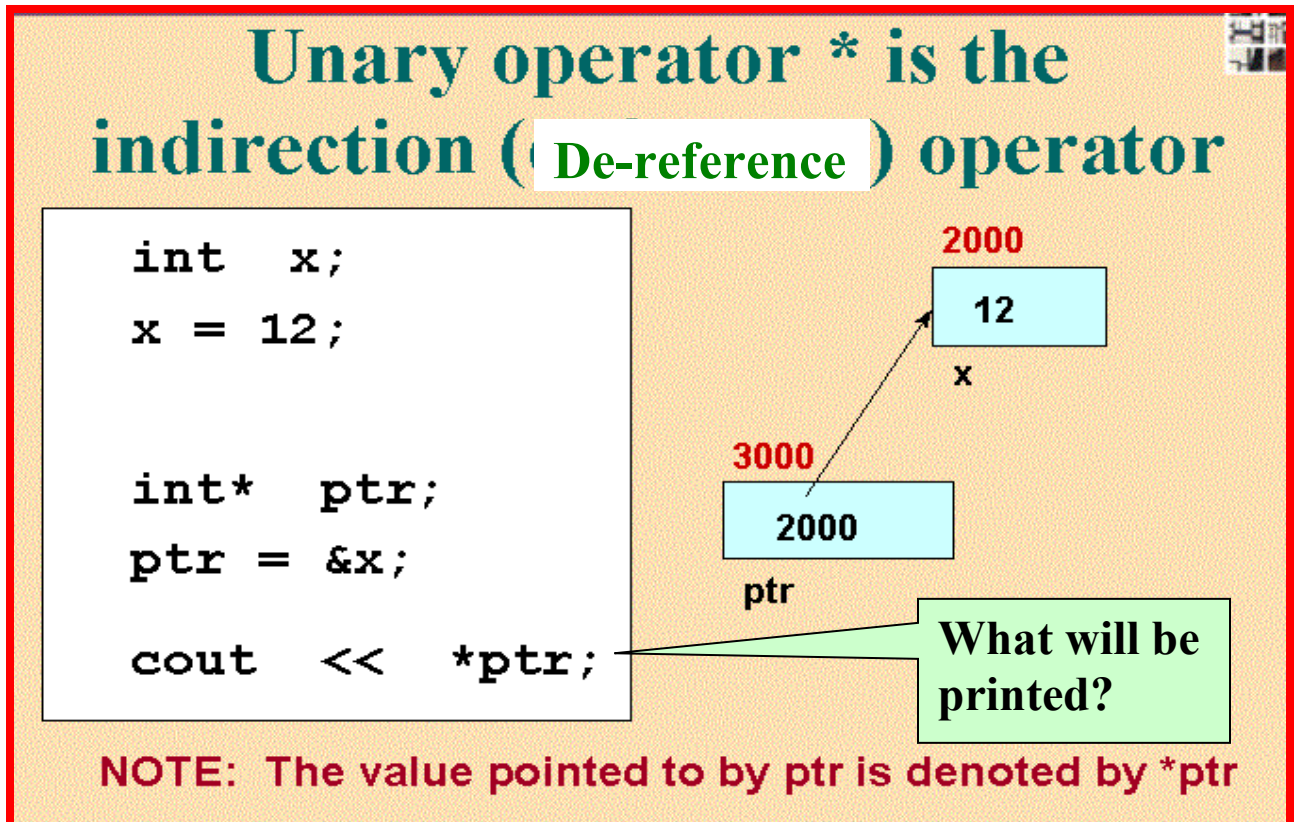


FIG 4.4 [Taken from Prof. Nell Dale]

In last line of the Figure 4.4 the pointer is used to print the value of variable x, whose address is stored in the pointer ptr. The syntax of using the pointer to a variable to get read and write access to it is as follows:

***ptr => acts as an alias for the variable whose address is stored in the ptr! Or in other words by putting the asterisk or dereference operator before the pointer variable makes it an alias for its “pointee”.**

Think of the asterisk in front of the pointer variable as saying, “Follow the pointer!”

If we follow the pointer ptr, where do we end up? We end up at the variable x, whose address is stored in the pointer ptr. **Thus de-**

referencing allows one to get read and write access to the memory location reached by following the pointer.

The pointer to a variable can also be used to change the value of a variable. This is shown in Figure 4.5.

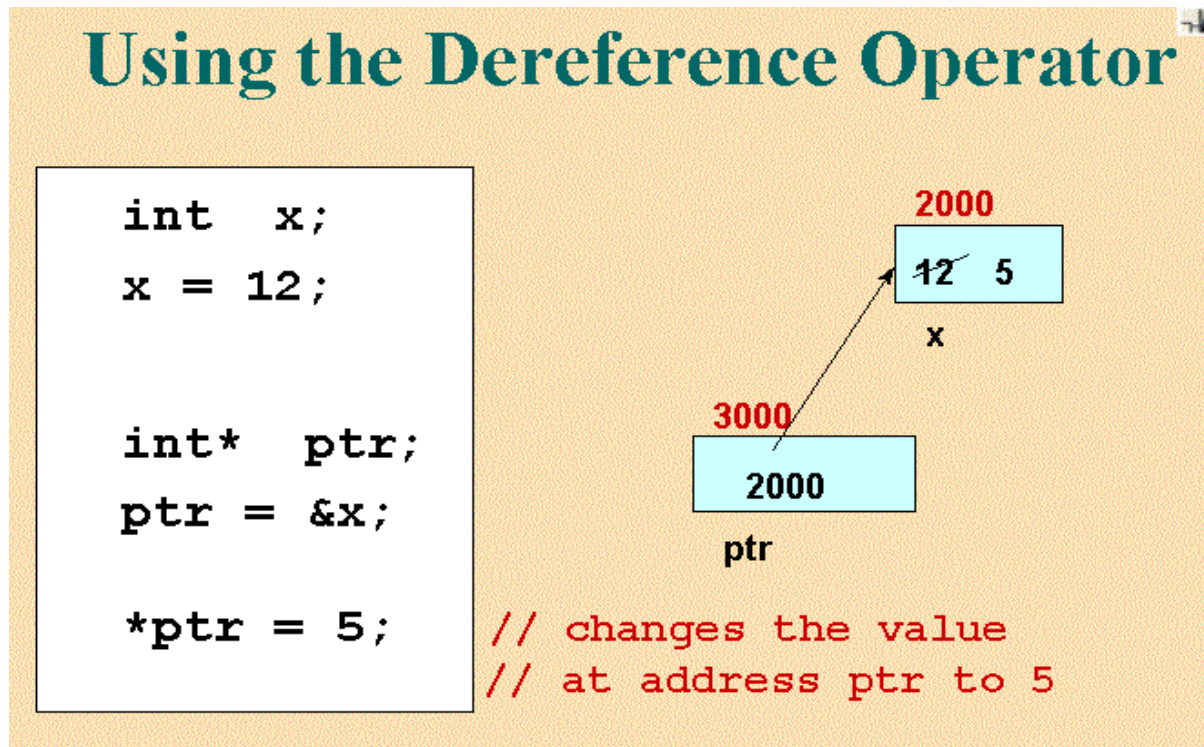






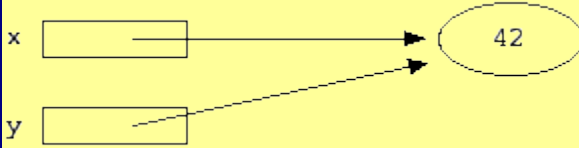
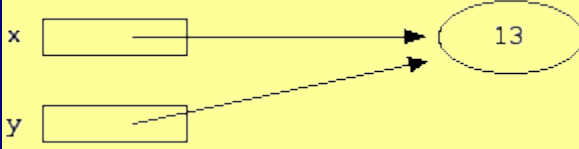
FIG 4.5 [Taken from Prof. Nell Dale]

In above code fragment, setting the de-referenced value of the pointer ptr, effectively changes the value stored in the memory location for variable x.

Dereferencing a pointer, which has no pointee, will crash your code!!

A common programming error is to try and de-reference a pointer that does not have a pointee. This is aptly proved by a video produced by Stanford University called “Pointer Fun with Binky”. Let us follow the following example from Binky Video code.

Description	Code	Picture of Program memory
1. Allocate two pointers x and y . Allocating the pointers does not allocate any pointees.	<pre>int * x; int * y;</pre>	
2. Allocate a pointee and set x to point to it. Each language has its own syntax for this. What matters is that memory is dynamically allocated for one pointee, and x is set to point to that pointee.	<pre>int pointee; x = &pointee;</pre>	
3. Dereference x to store 42 in its pointee. This is a basic example of the dereference operation. Start at x , follow the arrow over to access its pointee.	<pre>*x = 42;</pre>	
4. Try to dereference y to store 13 in its pointee. This crashes because y does not have a pointee -- it was never assigned one.	<pre>*y = 13;</pre>	

5. Assign <code>y = x;</code> so that <code>y</code> points to <code>x</code> 's pointee. Now <code>x</code> and <code>y</code> point to the same pointee -- they are "sharing".	<code>y = x;</code>	
6. Try to dereference <code>y</code> to store 13 in its pointee. This time it works, because the previous assignment gave <code>y</code> a pointee.	<code>*y = 13;</code>	
Table 4.1		

In table 4.1, in step one we declare two integer pointers `x` and `y`, which do not have “pointees”. Then in step two we create a pointee of integer type also with name pointee and make pointer `x` point to pointee. In step three we follow the pointer `x` or de-reference it to store 42 in its pointee. Up to that point the process works fine. But then in step four we try to de-reference the pointer `y`, which has no pointee and try to store 13. That is where your code will crash because `y` is not pointing to anything, or in other words it has no pointee. Therefore always remember:

Attempting to dereference a pointer not attached to a pointee (dangling pointer)

OR

Attempting to dereference a null pointer will crash your code!!!

The reason for such crash is very simple. The dangling pointer may have some value stored in it, which means that; the memory location there does not belong to your program. The attempt to de-reference such pointer means that your program is trying to take over the memory location that is not assigned to it. Therefore, operating system has no choice but to crash your program by a process called general protection fault. Same process repeats when one tries to de-reference a null pointer. A null pointer has a value, but it still has no “pointee”, and it also has no read or write access to any memory location. Therefore another way to rephrase the above conclusion is:

Pointers have no read or write access unless they are attached to pointees!

In step five we give the pointer y a pointee by making an assignment $x=y$. That simply means that store same memory address in y, which is stored in x. Now both x and y have same pointee. In step six we dereference pointer y and store 13 in its pointee. That process works fine because the value 13 is stored at the location where y and x both point. Another conclusion from this is:

A pointee can have more than one pointers pointing to it. All the pointers to a pointee have read and write access to it!

C++11 nullptr

C++ 11 introduced a null pointer constant that can be initialized to use pointers that are yet to get a pointee. C++98 used NULL, 0, '\0' etc. initializers for pointers holding null value. None of those values are quite appropriate and at times can create confusion, especially during function calls. Consider the code below where user intends to call the function that takes a pointer as an argument, but instead ends up calling a function that takes an int as an argument.

	<pre> #include <iostream> using namespace std; void foo(char * ptr){ cout<<"From foo(char * ptr).\n"; } void foo(int val){ cout<<"From foo(int val).\n"; } int main(int argc, const char * argv[]){ // Smart compilers such as xcode will not compile line below because it is //ambiguous // foo(NULL); foo(nullptr); // This will correctly call foo(char * ptr) return 0; } </pre>
	Output
	From foo(char * ptr).

Thus using compilers that support C++ 11 nullptr should be used as an initializer for pointers.

A Point of Confusion and Clarification

The use of indirection or de-reference operator some time confuses students. Let us look at the following code fragment.

```
int my_int = -10;

int *ptr = nullptr; //Statement #2

ptr = &my_int;

*ptr = 100;          //Statement #4

cout<< ptr;
```

Confusion in reading this code comes, when students look at the statement numbers 2 and 4 together and get the idea as if the pointer ptr has been de-referenced twice. Worst yet, as if the variable name is *ptr, rather than ptr. We can suggest way out of this confusion by using the keyword typedef and declaring a different name for the pointer data types. This is shown in Listing 4.2 below.

```

// *Listing 4.2
#include <iostream>
using namespace std;
typedef int* int_ptr;
typedef char* char_ptr;

int main()
{
    int my_int = -10;
    int_ptr iptr = NULL;
    iptr = &my_int;
    cout<< "The value of my_int = "<<my_int<<endl;

    *iptr = 100;
    cout<< "The new value of my_int = "<<*iptr<<endl;

    char_ptr string1 = "Hello World!";
    cout<<string1<<endl;

    char my_char = 'N';
    char_ptr cptr = NULL;
    cptr = &my_char;
    cout<< "The value of my_char = "<<*cptr<<endl;

    return 0;
}

```

No asterisk needed in declaration because we did that in defining int_ptr as pointer type.

No de-referencing needed, when printing a string.

```

The value of my_int    10
The new value of my_int - 100
Hello World!
The value of my_char   N

```

Listing 4.2

The listing 4.2 illuminates the advantage of changing the name of pointer variable such as int* to int_ptr by using typedef keyword. Now the role of indirection or de-reference becomes clearer. One can see it easily now that asterisk placed before the name of pointer variable can effectively be used as a second name for a variable.

Also, notice that no de-referencing is needed when a character pointer is used to print a string, but one would need it when printing a character through its pointer.

Question: In Listing 4.2, what will be printed if the following code fragment is added before return statement?

```
*iptr = 999 ;  
cout<<(++*iptr);  
cout<< (" degree temperature in ");  
cout<< *string1;  
cout<<*++string1;  
cout<<*++string1;  
cout<< *++string1;  
cout<< "!!!!\n";  
cout<< (string1)<<"\n";
```

Question: What will be the output of the following code fragment:

```
#include <iostream>
using namespace std;
typedef int* int_ptr;
int main( )
{
    int my_int = 1000;
    --my_int+=++my_int;
    cout<< my_int<<endl;
    int_ptr iptr = NULL;
    iptr = &my_int;
    *iptr/=2;
    cout<< *iptr<<endl;
    --*iptr+=++*iptr*2;
    cout<< --*iptr;
    return 0;
}
//Listing 4.3
```

Post increment or decrement of a variable using its pointer

The value of initialized variables may be changed using the post-decrement or post-increment operators by putting the operators directly behind the variable names. However, post incrementing or decrementing a variable via its pointer is a bit tricky. Let us examine the code fragment below.

```

typedef int* int_ptr;
int my_int = -22;
int_ptr iptr = nullptr;
my_int++; //Value of my_int = -21 now.
iptr = &my_int; //iptr points to my_int
--*iptr; //The value of my_int decremented by 1

```

```
//*iptr++ ;
```

Will this post-increment operation restore the value of my_int to -21?

```
(*iptr)++;
```

What is the value of my_int

In the code fragment above the results match expected pattern up to the statement:

```
--*iptr;
```

This because in a statement like above, the indirection operator is applied first, and then the pre-decrement operator applies. It is not true of the statement:

```
*iptr++;
```

In above statement, scanning from left to right, the indirection operator is seen first, but post-fix increment has a higher precedence than the indirection operator (*). Therefore, post-fix operator is applied first. But that is applied to iptr, and the result is that the pointer iptr is moved by one pointer unit. That has the affect of moving the address stored in the pointer iptr away from the location my_int. Results are much unexpected. If your system does not crash, then the *iptr will print the value of whatever the pointer ends up pointing to after the movement. This situation and correct procedures are shown in presentation below.



Microsoft PowerPoint
Presentation

Assigning More than one pointer to a Variable or Constant

C++ allows programmer to assign more than one pointer to a variable or a constant. However, in case of variable, then any of its pointers may be used to change its value. So one has to keep track of how the operation of each pointer is affecting the value stored inside the variable. Figure 4.6 illustrates this situation.

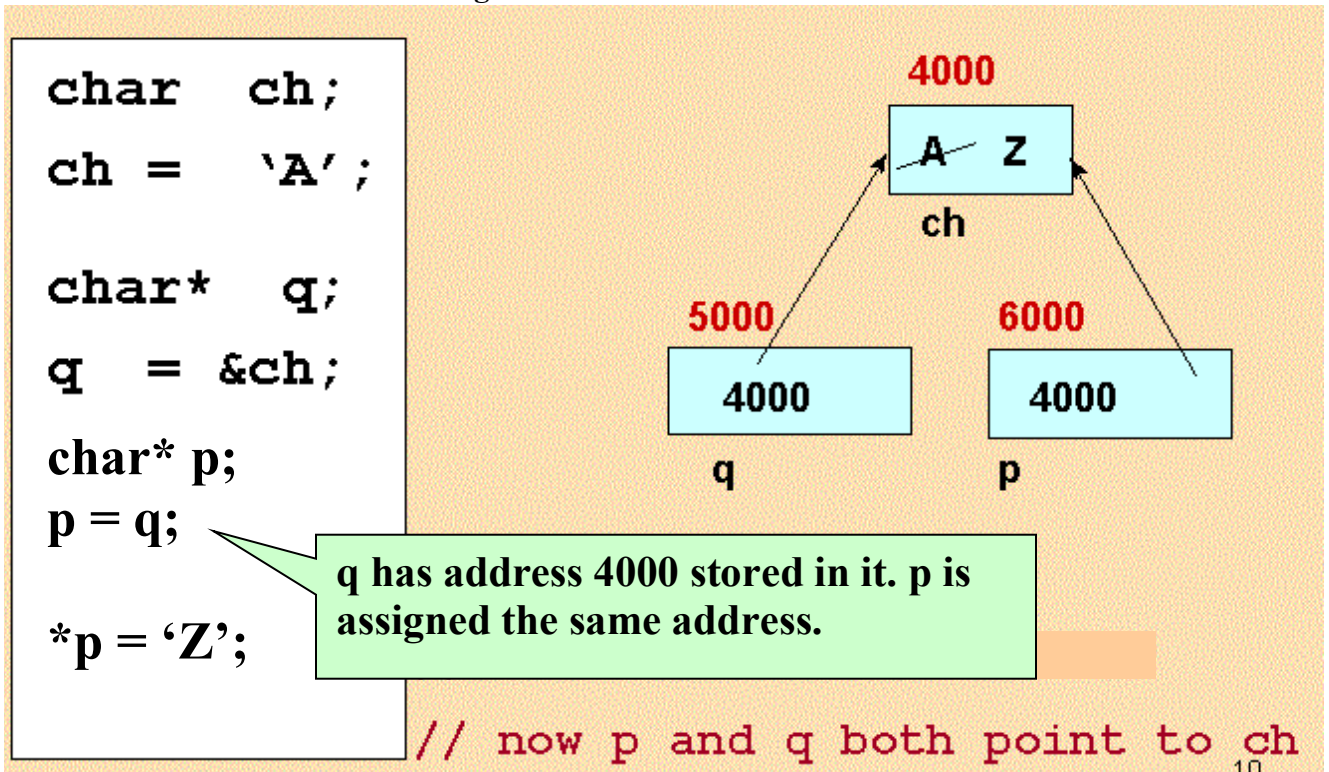


FIG. 4.6

In above Figure, first the character pointer **q** is made to point to **ch**. Then the second character pointer **p** is also made to point to **ch**. De-reference operator applied to **p** is used to change the value stored in **ch** from **'A'** to **'Z'**. Both pointer **p** and **q** have full access to the variable **ch** now.

Versatile void pointer

C has a void pointer data type, which may be used to hold the address of any of the data types. The syntax of declaring the void pointer is as follows:

```
void*   vptr1 = nullptr;  
//OR  
typedef void* void_ptr;  
void_ptr vptr2 = nullptr;
```

However, the void pointer cannot de-reference the variable, whose address is stored in it, directly. Rather it must be first cast into that pointer type, which can directly point to the data in the variable. After casting the void pointer, the de-referencing may be carried out as usual. The Listing 4.5 shows the use of void pointer.

```
#include <iostream>
using namespace std;
```

```
typedef int* int_ptr;
typedef void* void_ptr;
typedef float* float_ptr;
typedef char* char_ptr;
```

```
int main( )
```

```
{
```

```
    void_ptr vptr = nullptr;
```

```
    int my_int = -10;
```

```
    vptr = &my_int;
```

```
    cout<<"The address of my_int printed using "<<
        "the address operator = " <<(long)&my_int<<endl;
```

```
    cout<<"The address of my_int printed using "<<
        " the void pointer vptr operator = " <<(long)vptr<<endl;
```

```
    cout<<"Value of my_int = " <<*int_ptr(vptr)<<endl;
```

```
    float my_float = -9.999f;
```

```
    vptr = &my_float;
```

```
    cout<<"Value of my_float = " <<*float_ptr(vptr)<<endl;
```

```
    char_ptr cptr= "Hello El Camino!!!";
```

```
    vptr = cptr;
```

```
    cout<<char_ptr(vptr)<<endl;
```

```
    char ch = 'M';
```

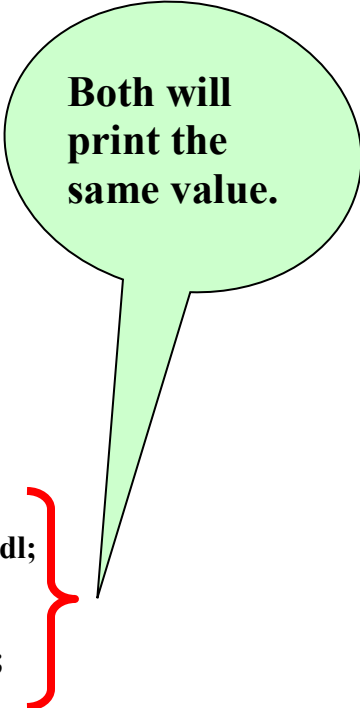
```
    cptr = &ch;
```

```
    vptr = cptr;
```

```
    cout<<"The value of ch = " <<*char_ptr(vptr)<<endl;
```

```
    return 0;
```

```
}
```



**Both will
print the
same value.**


```

The address of my_int printed using
the address operator = 6684144
The address of my int printed using
the void pointer vptr operator 6684144
Value of my int 10
Value of my float 9.999000
Hello El Camino!!!
The value of ch = H

```

Listing 4.5

The mechanism of using the void pointer vptr is similar in all cases. We illustrate the mechanism, by the system of following power point slides.



Microsoft PowerPoint
Presentation

The description of the first part of the program is given below.

Referring to the Figure 4.7 the statement,

`int my_int = -10;`

creates an int type memory, in which -10 is stored. The statement,

`void_ptr vptr = NULL;`

creates a void pointer and NULL is stored in it.

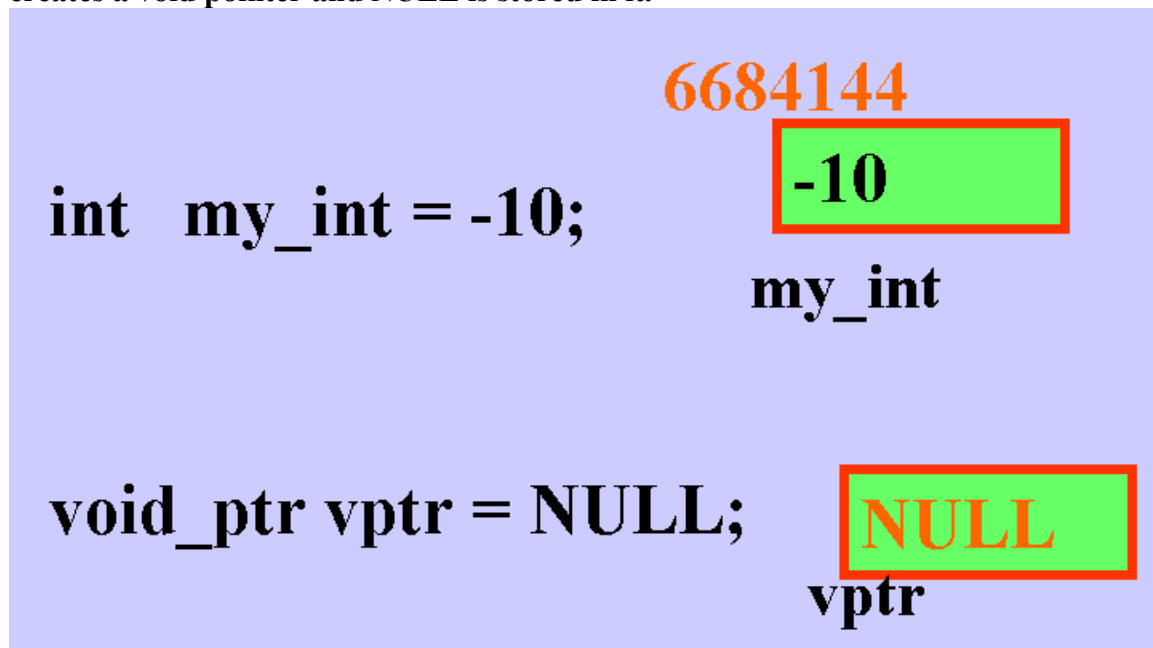
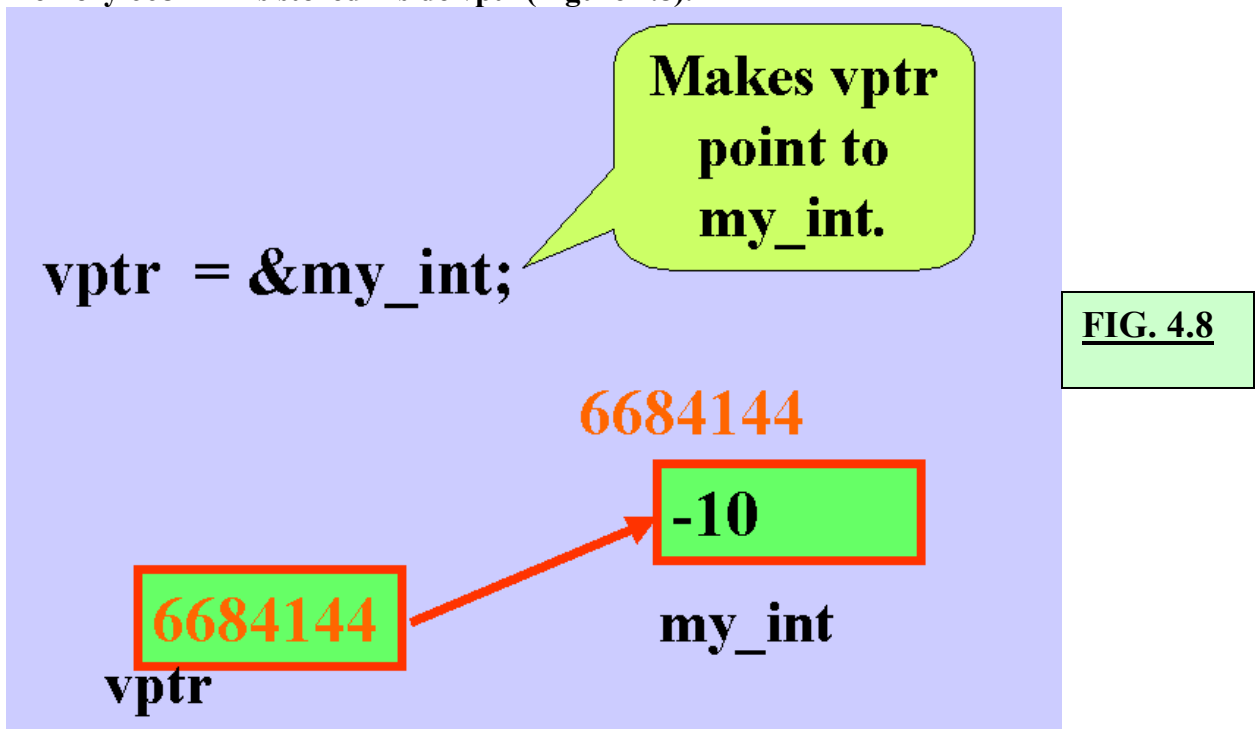


FIG. 4.7

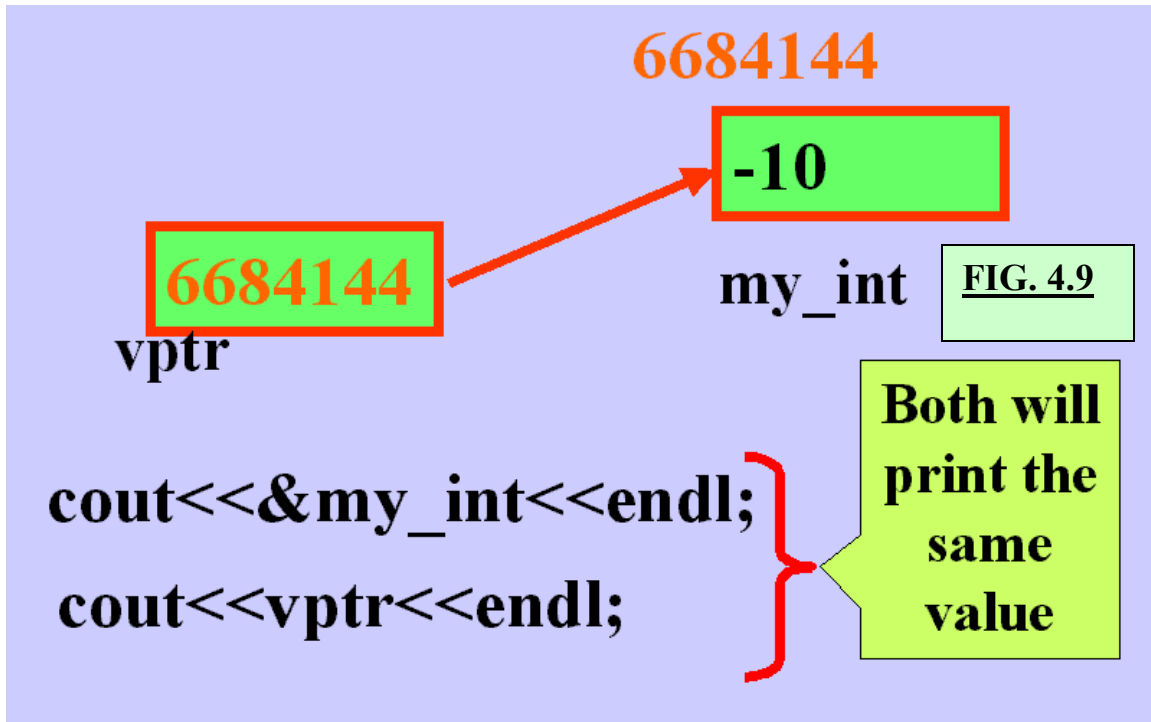
NULL should be replaced by `nullptr` in above Figure.

The statement,
`vp_ptr = &my_int;`
makes the pointer `vp_ptr` point to the memory location `my_int`, and the address of the memory 6684144 is stored inside `vp_ptr` (Figure 4.8).



Now both the following statements will print the address of `my_int` a value of 6684144 assumed in the illustration (Figure 4.9).

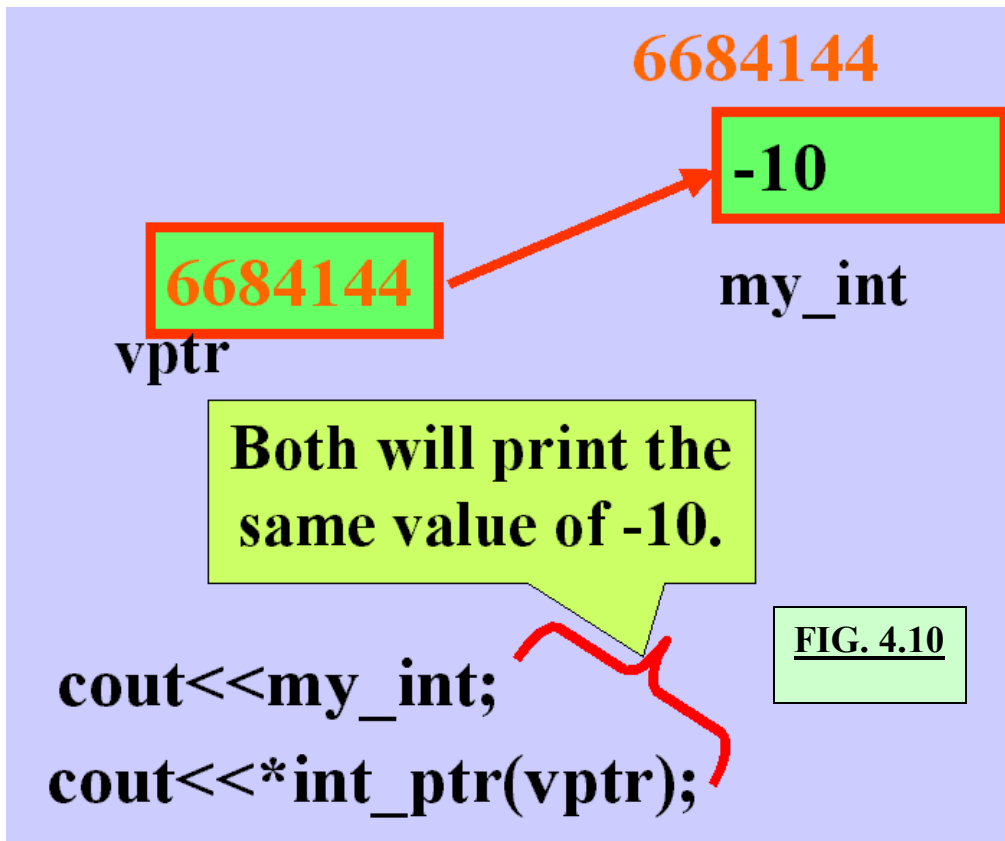
```
cout<< &my_int;  
cout<< vp_ptr;
```



Now both the following statements will print the value stored in `my_int` a value of -10.

```
cout<< my_int;
cout<< *int_ptr(vp_ptr);
```

In the second statement, the void pointer `vp_ptr` is being cast into an `int` type pointer first, and then being de-referenced (Figure 4.10).



Also note that when we use character pointer to store the string “Hello El Camino!!!”, we do not need to use address operator to store the address of the string in the void pointer. Therefore the statements,

```
char_ptr cptr = “Hello El Camino!!!”;
```

```
vp_ptr = cptr;
```

are adequate to store the address of string in the void pointer. But when we attempt to print the string, then we must re-cast the void pointer to a character pointer explicitly. In addition, as usual the de-referencing operator is not needed to print a string.

Pointers to other pointers.

The pointers can store the addresses of other pointers as well, provided the data type match is present. For example, consider the following declarations:

```
int* iptr = nullptr;
```

iptr can store the address of an int data.

```
int** ptr_iptr = nullptr;
```

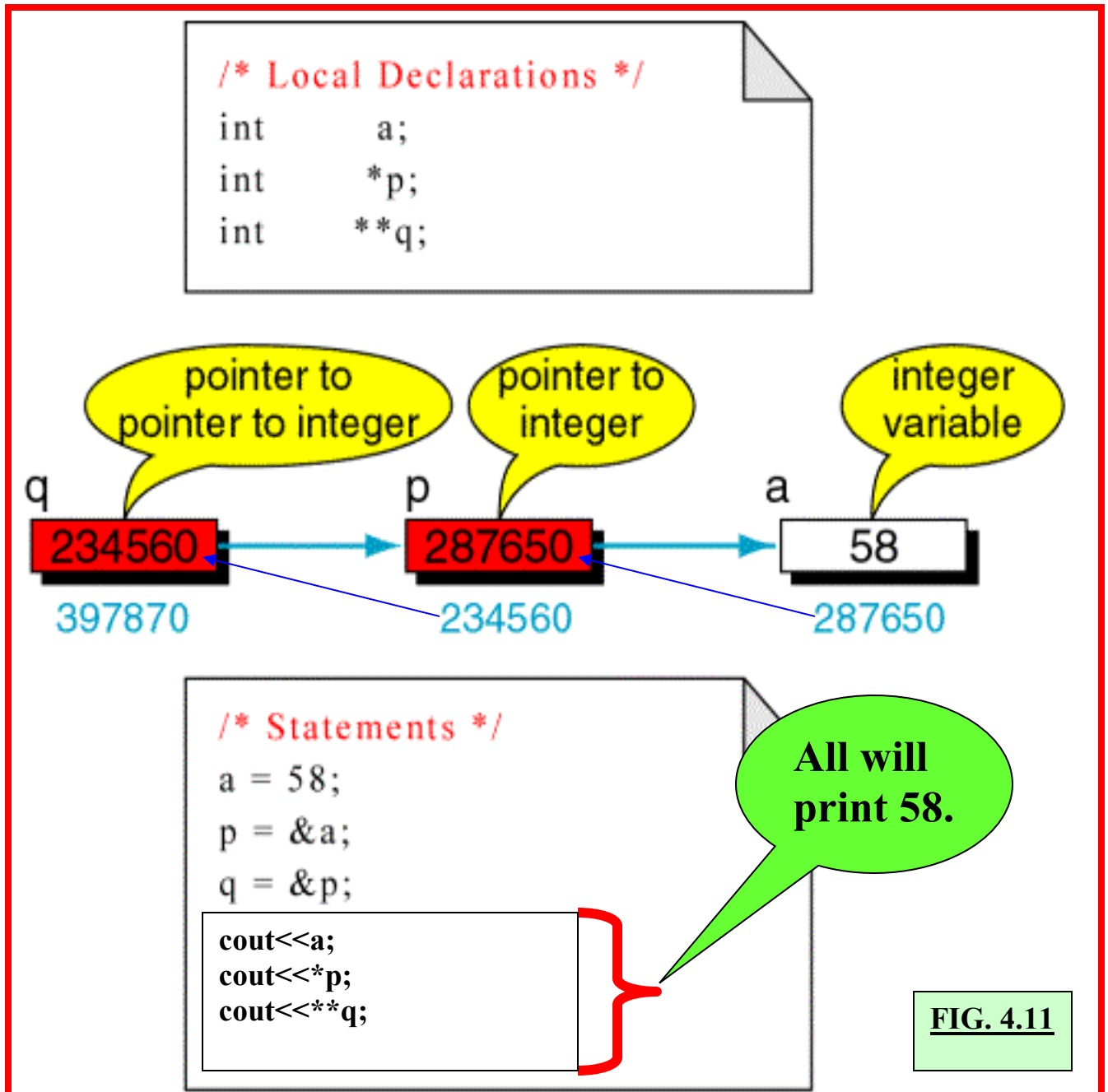
Can store the address of an int pointer data.

```
int*** ptr_ptr_iptr = nullptr;
```

Can store the address of a double int pointer data.

Note that more than two levels of indirection are rarely useful!

Therefore, at the most one may expect to use a double pointer of type `int**` or `char**` in the real programs. We show a toy example of using the pointer to pointers in Figure 4.11.



In above program, we declare a variable `a`, and two pointers `p` and `q`. `p` is a pointer to `a`, and `q` is a pointer to `p`. All relate to `int` data types. We set the value of `a` to 58. Then we make pointer `p`, point to `a` and pointer `q` point to `p`. This stores the address of `a` in `p`, and the address of `p` in `q`. The value of `a`, can now be printed in three different ways. Directly by printing the contents of `a`, or indirectly, either by de-referencing `p`, or double de-referencing `q`. Last three lines of the program print the value of `a` directly and through the pointer and double pointer.

Summary points about pointer variables:

- The pointer variables have the read and write access to memory locations of their pointees.
- The pointer variables can be re-assigned to “other” pointees.
- However, the pointer variables cannot be assigned to named constants. For example, the below will not compile:

```
const int val = 10;  
int * ptr;  
ptr = &val;//Compile Error
```

A pointer variable cannot hold the address of a named constant!

Relationship of arrays and pointers

Arrays have random read and write access because given the array name and its script; program can calculate the exact location of the array element, and access the element. This is made possible by one simple fact:

The name of the array holds the memory address of its first element. Thus array name is a pointer to its first element.

The name of an array holds the address of its very first element, but it is not a pointer variable. Rather it is like a named constant. We summarize three facts about the array name:

1. Name of an array holds the address of the location of its first element in the memory.
2. But the name of an array acts not like a variable, but more like a named constant.
3. Since pointers can also hold the memory address of variables, one can say that the name of an array is a constant pointer. (This fact will become clear soon.)

We will prove the first of three points made above first. In listing 4.6 we print the address of first element of the array in two ways; one by using just the array name and secondly using the address of operator. You will see that they both will give the same value.

```

#include <iostream>
using namespace std;

const int MAX_SIZE = 5;
const float FLOAT_INITIALIZER = 0.0f;
const char MESSAGE1[ ] =
"The memory address of the 1st array element is = \n";
const char MESSAGE2 [ ] =
"The above 2 values must be same.\n";

int main()
{
    float Array[MAX_SIZE] = {FLOAT_INITIALIZER};

    cout<<MESSAGE1;
    cout<<(long)Array<<endl;
    cout<<MESSAGE1;
    cout<<(long)&Array[0]<<endl;
    cout<<MESSAGE2;

    printf(MESSAGE2);
    return 0;
}

```

cout<<MESSAGE1;
 cout<<(long)Array<<endl;
 cout<<MESSAGE1;
 cout<<(long)&Array[0]<<endl;
 cout<<MESSAGE2;

Only the array name is used to print the address of first element.

Array name with subscript and ampersand is used to print the address of first element.

```

The memory address of the 1st array element is =
6684132
The memory address of the 1st array element is
6684132
The above 2 values must be same.

```

Listing 4.6

The Listing 4.6 proves that array name holds the address of its first element and is a pointer to its first element. This proves the first of the three points we made on previous page. In order to prove the other two points we will have to learn little bit more about pointers.

Constant Pointers or Restricted Pointers

Just as the program data may be variables or constants the pointer data can also be variables or constants. Normal program pointers, which we have discussed so far, have read and write access to memory address they point to, and they also can be re-assigned to other memory addresses of their types. Now we discuss the special or restricted pointers, where for the sake of data security and other purposes some of the freedoms allowed to normal pointers are removed. The generic name for such pointers is constant pointers. So far, we know only of one constant pointer, which is the name of an array. We can, however explicitly declare constant pointers and assign them to hold the address of other named constants. These pointers can involve varied degree of constancy.

1.Constant Pointer to Program Constants/Variables:(Read-Only – Access and Non-reassignable pointers)

The syntax of declaration is as follows:

```
char string1 [ ] = "Hello El Camino.";
```

Must be initialized in the same line like a named constant! Both forms of declaration are valid.

```
const char * const constant_cptr = string1;  
char const * const constant_cptr = string1;
```

Use of two const ascertains that pointer only has read access and is not re-assignable. Note that 2nd const goes after the asterisk.

Compared to pointer variables, the first **const** takes away the write access and second **const** takes away the re-assignability.

Another example is below:

```
const int arr [5] = {1,2,3,4,5};  
const int * const val = arr;
```

const arrays must be fully initialized! Partial initialization is not allowed!

```
cout<<constant_cptr;
```

Read access through this constant pointer permitted!

```
string1 [0] = 'Y';
```

Write access through the array name is permitted!

```
//constant_cptr[0] = 'Y';
```

But write access through this constant pointer is not permitted ever!

```
// const char * const constant_cptr2;
```

This is compiling Error!
This constant pointer must be initialized in the declaration line.

```
char my_char = 'X';
```

```
const char * const constant_cptr2 = &my_char;
```

OK, even though my_char is not a constant! One can assign a constant pointer to a variable.

```
cout<< *constant_cptr2;
```

Read access through a constant pointer permitted!

```
//*constant_cptr2 = 'Z';
```

Compiling error! Write access through this constant pointer not permitted, even if variable pointed to is not a constant!


```
char your_char = 'Y';  
// constant_cptr2 = &your_char;
```

Also a compiling error!
Once assigned, a constant
pointer cannot be
reassigned!

Conclusions about Constant Pointers To Program Constants:

- Like a named constant, a constant pointers discussed above must be initialized in the line declaring it. This simply means that its pointee must have been created before the constant pointer is declared.
- Has read-only access. No, write access, even in the array. In that sense, even more strict than the array name.
- Can store the address of a variable. But treats the variable like a named constant.
- Also once assigned, this constant pointer cannot be reassigned to other pointees.
- Ideal for secure processing of those strings, whose value must never change, like username or password. Listing 4.7 shows an example.

```
0001 #include <iostream>  
0002 #include <string>  
0003 using namespace std;  
0008 const int MAX = 7;  
0012 const int NUM_TRY = 3;  
0021 bool findName(const char * const username[],  
string string1);  
0022  
0023 int main()  
0024 {  
0025     const char * const username [MAX] =  
0026     {"Bertha", "Tina", "Rob", "Freddy", "Mike",  
0027     "Sinichi", "Lincoln"};  
  
//Uncommenting the line below will not compile  
//because there is no write access through  
//constant pointer username.
```

Username stored in a
read-only array. The
contents of this array
cannot be altered.

```

00032 //      username [1] = "Donna";
//The line below will compile because
//names is not a constant pointer.
00037      char * names [] = {"abc", "jhg"};
00038      names [1] = "Donna";
//The constant pointer can also be assigned to a
//named constant
    /*const int VAL = 5;
    const int * const iptr = &VAL; // this also OK
    cout<<*iptr<<endl;*/
00039
00040      string string1;
00041      bool found = false;
00042      bool done = false;
00043
00044      cout<<("Please enter your username : ");
00045      getline(cin,string1);
00046
    for(int try_counter = 0;!done&&!found && try_counter<NUM_TRY;
    try_counter++)
00048    {
    if(!(found= findName(username,string1))&& try_counter<NUM_TRY-1)
00050    {
00051    cout<<("Try again? Enter 0 to continue and 1 to exit: ");
00052    cin>>done;
00053    cin.ignore (128, '\n');
00054    cout<<("Please enter your username: ");
00055    getline (cin, string1);
00056    }//end of if
00057    }//end of for loop
00058
00059    if(!found)
00060    {
00061    cout<<("Sorry, your username was not found.\n");
00062    cout<<("Please contact System Administrator.\n");
00063    }
00064    return 0;
00065 }
00066
00067 bool findName (const char * const username [], string Name)
00068 {
00069    bool found = false;
00070    for (int index=0;! found&&index<MAX; index++)
00071    {

```

```

00072         if (Name==string (username [index]))
00073         {
00074             cout<<("Authorized to use computer.\n");
00075             found = true;
00076         }
00077     }
00078
00079     return found;
00080 }

```

```

Please enter your username : Satish
Try again? Enter 0 to continue and 1 to exit : 0
Please enter your username : Mary
Try again? Enter 0 to continue and 1 to exit : 0
Please enter your username : Rob
Authorized to use computer.

```

```

Please enter your username : Satish
Try again? Enter 0 to continue and 1 to exit : 0
Please enter your username : Mary
Try again? Enter 0 to continue and 1 to exit : 0
Please enter your username : Nguyen
Sorry, your username was not found.
Please contact System Administrator.
Press any key to continue

```

Listing 4.7

In above Listing, we hard code the username in an array, whose contents can never be modified, because the pointer to it is a read only pointer. Then user enters their username. A function findName is used to check, whether the username entered matches with any of the names stored in the read-only array. If the name matches, then the user is given a message that they are authorized. However, if username does not match, then user is given two more chances to enter the correct username. After three attempts, the user is asked to contact the system administrator.

The important thing about this code is that the contents of the read-only array username can never be altered. However, the limitation is that username must be hard coded in the program, an inconvenient feature. I do not believe that any string in C++, which gets data from an external source, like, file, database or user entry, can ever be made safe from possible alteration. Therefore, the language like Java, which has a string as an unalterable object promote security better than C++ was able to.

2. Pointers to Program constants/variables (Still Read-Only Access, but Re-Assignable)

The pointers to constants must be such that pointer cannot change the value of the constant it points to. A constant read-only pointer discussed earlier can point to a constant but is not re-assignable. If we remove the second `const` from the definition of the read-only-non-reassignable pointer defined earlier, we get a pointer, which still has read-only access but it can be re-assigned to other constants and variables. The syntax of defining this read-only pointer is as follows:

```
const int * iptr_const1 = nullptr;
int const * iptr_const = nullptr;
```

Both declarations are valid. The pointers have read only access. Since they are re-assignable, the initialization to a data address in same line is no longer necessary.

Notice that `const` goes before the asterisk!

Listing 4.8 shows syntax, and allowed and prohibited operations.

```
#include <iostream>
using namespace std;

int main()
{
    const int my_int = -9;
    const int * iptr_const = NULL;
    //alternate declaration below
    //int const * iptr_const = NULL;

    iptr_const = &my_int;
    cout<<*iptr_const<<endl;

    //*iptr_const = 99;

    int your_int = 99;
    iptr_const = &your_int;

    // *iptr_const = -99;
    cout<<*iptr_const<<endl;

    return 0;
}
```

Has read-Access.

No write access!

Reassignable to other data addresses.

No write access even for a non-constant!

Listing 4.8 (C++ 11 note: Replace `NULL` by `nullptr`)

The read-only reassignable pointer works best for those situations where one may have to deal with lot of constant data, but they are not used for the entire life of the program.

3. Constant Pointers (Have Read and write access, but not re-assignable).

The syntax of declaration is illustrated by the following examples:

```
int value;  
int arr [5] = {0};  
int * const iptr = arr;
```

Read and write access to array arr allowed through iptr. Acts just like an array name.

```
iptr = &value; // Compile error
```

Constant pointer cannot be re-assigned!

```
int * const iptr1 = {1,2,4, 6}; // compile error
```

Cannot be used to declare an array!

Listing 4.9 gives further examples and illustration of a constant pointer. Most characteristic of a constant pointer is the address stored in it is fixed at the time of declaration and can never be changed after that. However, it has write access to the variable it is pointing to. This pointer cannot be assigned to named constants. For example, the below will be a compile error.

```
const int data = 5;
```

```
int * const val = &data; // Compile error! Cannot be assigned to a constant!
```

This type of constant pointer can however be assigned to already created arrays or other program variables, where pointer must not be re-assigned during the program life.

Must be initialized with a pointee in the declaration line.

Has write and read access!

```

#include <iostream>
using namespace std;

int main()
{
    int my_int = -9;
    int * const const_iptr = &my_int;
    int * iptr = NULL;

    cout<<*const_iptr<<endl;

    *const_iptr = 99;

    cout<<*const_iptr<<endl;

    //const_iptr = iptr;

    int your_int = 100;

    //const_iptr = &your_int;
    return 0;
}

```

Listing 4.9 (C++11 Note: Replace NULL by nullptr)

The name of an array acts like a constant pointer discussed in this section. The array name has a read and write access, but is not re-assignable. For example the following code will cause a compile error:

```

int arr1 [2];
int arr2 [2];
arr1 = arr2; //compile error

```

Array Names in C++ are not reassignable. That is why arrays cannot be returned as a return value from a function with a proto- type such as `int []`

When you run above code the compiler will give you a rather cryptic message: “left operand must be l-value”. What that means is that array name is a constant pointer, and constants cannot be on left hand side of an expression, except the first time

when they are declared. An l-value operand can always be on the left hand side of expressions. All variables are l-value operands. Constants are not l-value operands.

The table 4.2 below summarizes the properties of three kinds of pointers relating to different kinds of constancy.

Declaration	Read-Access?	Write-Access?	Re-assignable?	Alternate Declaration
const <datatype> * const <name> = <initialization>;	Yes	No	No	<datatype> const * const <name> = <initialization>;
const <datatype> * <name> ;	Yes	No	Yes	<datatype> const * <name>;
<datatype> * const <name> = <initialization>;	Yes	Yes	No	No

Pointer Arithmetic

When a pointer is assigned to point to an element of array, the pointer can be used to traverse the array towards increasing or decreasing indices. This forms the basis of pointer arithmetic. We know that the name of an array is a constant pointer to its first element. In fact we can use either the array subscript or the de-reference operator applied to the array pointer to have random read or write access to any array element. The Figure 4.12 shows part of this mechanism in action.

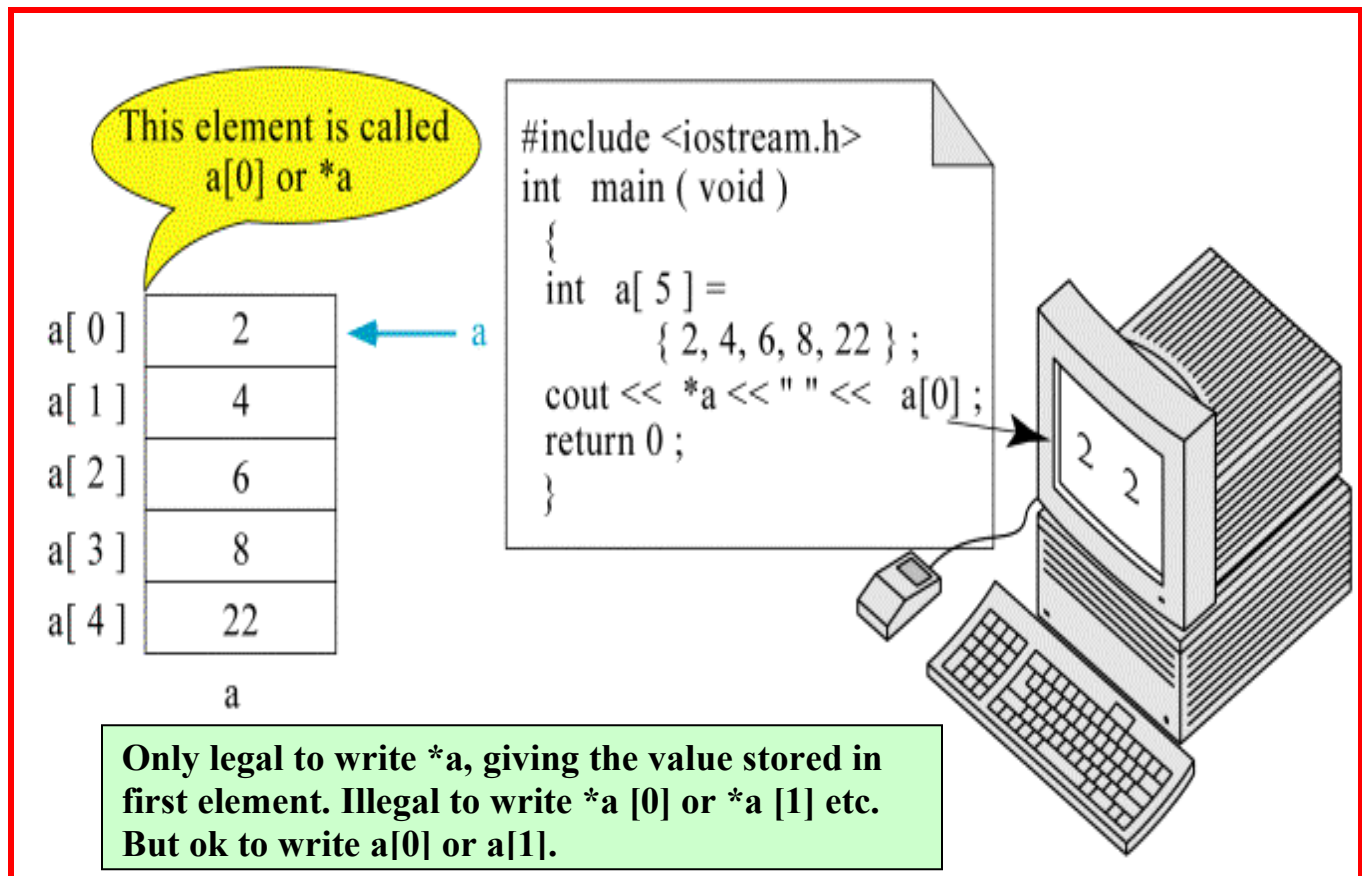


Figure 4.12

In Figure 4.12 the first element can be printed either by de-referencing the array name `*a`, or by using the array subscript `a[0]`.

One can assign the pointer to any element of an array and then depending upon the array bounds one can move pointer backwards or forward. Example of this is shown in Figure 4.13.

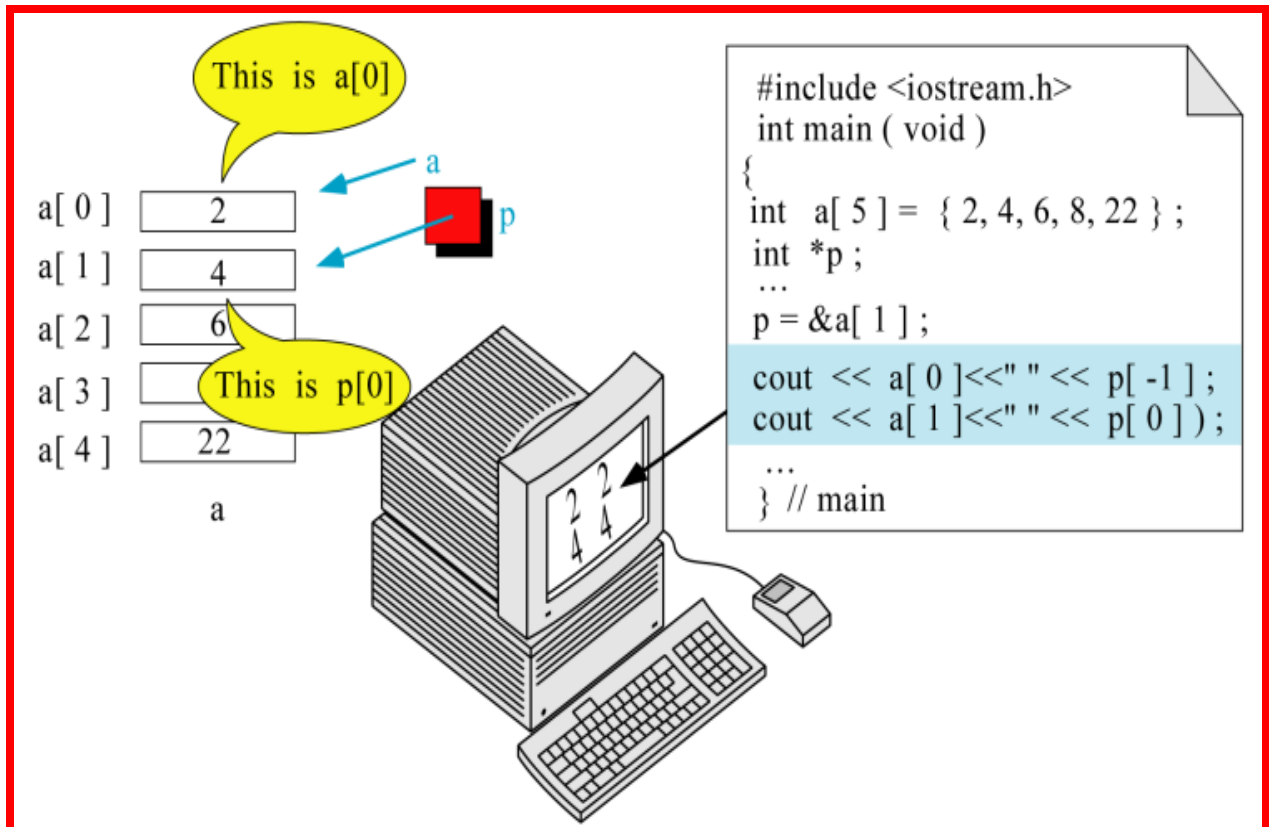


FIG. 4.13

Note that in Figure 4.13, we have assigned the pointer `p` to the second element of the array. Therefore `p[-1]` is a valid, inbound array element. Now we can print the array elements, either using the array name and the subscript operator, or by using the pointer and subscript operator. The above Figure points to this conclusion about pointer assigned to an array element:

Given a pointer `p`, the `p + n` or `p - n` is also a pointer to an element, which is either `n` elements forward or `n` elements backwards in location, with respect to `p`. Of course `p + n` or `p - n` both must be within the legal array bounds.

We reinforce this concept by giving two versions of access to array elements using the array name, and three versions of access using the pointer assigned to the array. This is shown in Listing 4.10.

```

#include <iostream>
using namespace std;

int main()
{
    int int_array[4] = {2,4,6,8};
    int * iptr = nullptr;
    int index = 0;
    iptr = int_array;

    cout<<("Printing using the array subscript.\n");
    for(index=0; index<4; index++){
        cout<<"int_array[ "<<index<<" ] = "<<int_array[index]<<endl;
    }

    cout<<("Printing using the array name ");
    cout<<("and de-reference operator.\n");
    for(index=0; index<4; index++){
        cout<<"*(int_array+ "<<index<<" ) = "<<*(int_array+index)<<endl;
    }

    cout<<("Printing using the pointer and subscript.\n");
    for(index=0; index<4; index++){
        cout<<"iptr[ "<<index<<" ] = "<<iptr[index]<<endl;
    }

    cout<<("Printing using the pointer name ");
    cout<<("and de-reference operator.\n");
    for(index=0; index<4; index++){
        cout<<"*(iptr+ "<<index<<" ) = "<<*(iptr+index)<<endl;
        cin.get();
    }

    cout<<("Printing by advancing the pointer 1 unit ");
    cout<<("at a time and de-reference operator.\n");
    for(index=0; index<4; index++){
        cout<<"*iptr at array index "<<index<<" = "<<*iptr++<<endl;
    }

    return 0;
}

```

2. De-referencing the array name plus the index.

3. Using the pointer and subscript

4. De-referencing the pointer name plus the array index.

5. Advancing the pointer 1 unit at a time and printing the contents at the current location.

```

Printing using the array subscript.
int_array[ 0 ] = 2
int_array[ 1 ] = 4
int_array[ 2 ] = 6
int_array[ 3 ] = 8
Printing using the array name and de reference oprator.
*(int_array+ 0 ) = 2
*(int_array+ 1 ) = 4
*(int_array+ 2 ) = 6
*(int_array+ 3 ) = 8
Printing using the pointer and subscript.
iptr[ 0 ] = 2
iptr[ 1 ] = 4
iptr[ 2 ] = 6
iptr[ 3 ] = 8
Printing using the pointer name and de reference oprator.
*(iptr+ 0 ) = 2
*(iptr+ 1 ) = 4
*(iptr+ 2 ) = 6
*(iptr+ 3 ) = 8
Printing by advancing the pointer 1 unit at a time and de reference oprator.
*iptr at array index 0 = 2
*iptr at array index 1 = 4
*iptr at array index 2 = 6
*iptr at array index 3 = 8

```

Listing 4.10

The below also works:

```

int arr[5] = {5,2,1,4,6};
int * ptr = nullptr;
ptr = arr;
for (int i= 0; i<5; i++) {
    cout<< i[ptr]<<endl;
}

```

Above program will print 5 2 1 4 6

That is because i[ptr] does the same pointer arithmetic that ptr[i] does! In either case pointer ptr is advanced by offset of one.

The Listing 4.10 draws the conclusion that random access to array elements, by using any of the following mechanisms produces the same result.

1. Using the array name and subscript, like `array [index]`,
2. De-reference the array name plus the index, like `*(array+index)`,
3. Using the subscript to an external pointer assigned to array element, like `ptr [index]`,
4. De-referencing the pointer name plus the index, like `*(ptr+index)`,
5. Keeping track of where the pointer `ptr` is in the array and advancing it one element at a time, using the post increment operator, like `ptr++`. The post decrement operator may also be used if one needs to traverse array backwards.

All five cases are clearly shown in the Listing 4.10 and its results.

The above examples were shown to establish the equivalence between the pointer movements through an array and use of array name and subscript operator. This equivalence forms the basis of pointer arithmetic, as well as the formula for calculating the address of any array element, provided the address of first element is given. We further show some more interesting and perhaps more useful applications of pointer arithmetic. One important example is to traverse a C string to find its length. Listing 4.11 gives several methods to do so.

```

#include <iostream>
using namespace std;

int main()
{
    char string1[] = "I am a string of length 26";
    char * cptr = string1;

    while(*++cptr);

    cout<<"The length of \"<string1<<\"\" = "<(cptr-string1)<<endl;

    cptr = string1;

    while(*cptr)
        cptr++;

    cout<<"The length of \"<string1<<\"\" = "<(cptr-string1)<<endl;

    cptr = string1;

    while(*cptr++)

    cout<<"The length of \"<string1<<\"\" = "<(cptr-string1-1)<<endl;
    return 0;
}

```

1. Loop will exit, as soon as the *cptr is nullptr.

2. Results are similar to case 1.

String length = address of NULL pointer - address of first cell.

3. Post-incrementing in the loop expression forces cptr to move 1 unit past the null pointer. Thus string length formula changes

```

The length of "I am a string of length 26" = 26
The length of " I am a string of length 26" 26
The length of " I am a string of length 26" - 26

```

Listing 4.11

The logic behind Listing 4.10 is simple. We set the character pointer cptr to the first element of the string, by making the assignment,
cptr = string1;

We know that the string ends, as soon as the de-referenced value of `cptr` will become null. Therefore we pre or post increment the `cptr` through the while loop, so that loop exits, as soon as `*cptr` is null. This is what happens in cases 1 and 2. Then subtracting the address of first cell (`string1`) from the `cptr` gives us the string length. In third case, where we post increment the pointer `cptr` inside the loop expression, the pointer is moved 1 unit past the null pointer. That is because, even though loop expression evaluates to false, as soon as the `*cptr` becomes NULL, the loop expression must complete its task and increment the pointer `cptr` by one unit. Therefore, in case three the string length is given correctly, by `cptr` minus `string1` minus one.

Review Questions

1. What will an address of operator (&) attached before the variable name will print?
2. What will the code line below print? (Assume that function body is provided after the main.

```
void function( );
void main( )
{
    cout<<(long)&function<<endl;
//Line #2
}
```
3. If in above code we add the following line at place line#2 will that compile? If yes why?

```
void * vptr = &function;
```
4. Will the code line below compile and print anything?

```
cout<<(long)&"Hello World"<<endl;
```
5. Describe the syntax of declaring a pointer variable to a data type in C++.
6. What will happen if the code below is run?

```
int * ptr;
ptr =5;
```
7. In the declaration below, which variables are pointers and which one's are integers?

```
int*   val1, val2, val3, *val4=0; val5;
```
8. What are the three different ways to initialize a pointer variable?
9. What is a dangling pointer?
10. Describe the procedure of making a pointer point to a "pointee".
11. Why would the code given below not compile?

```
int* ptr;
float val = 10;
ptr = &val;
```
12. Why would the code given below compile?

```
void* ptr;
float val = 10;
ptr = &val;
```

13. Describe the syntax of printing the value of val in above question through the void pointer assigned to it.
14. What are the two names for the operator * used in pointer declaration? What is the meaning of de-referencing a pointer?
15. A pointee can have more than one pointer attached to it. (True/False?).
16. What will the code below print?

```
char * cptr ="Hello World";
cout<<cptr<<endl;
```
17. What will the code below print?

```
char ch = 'Y' ;
char * cptr = &ch;
cout<<cptr<<endl;//line #3
```
18. In above code why some unrecognizable characters are printed? How should code line #3 must be modified so that it prints only Y and not the extra characters?
19. Describe the procedure of applying postfix and prefix operators to an integer pointer so that that the value of pointee is incremented or decremented properly.
20. In code below, describe the procedure of printing the value of variable x through the double pointer Q.

```
int x = 58;
int * p = &x;
int** Q = &p;
```
21. How many levels of indirection are allowed by C++ in pointer declaration?
22. Can a pointer variable store the address of a named constant?
23. Name of array holds the memory address of which of its element?
24. Name of an array is a pointer variable?(True/False?)
25. Describe the syntax of declaration of a read-only and non-reassignable pointer. In what software applications such pointers are useful?
26. Describe the syntax of declaration of a read-only but reassignable pointer.
27. Describe the syntax of declaration of a read and write access but non-reassignable pointer.
28. The array name is what kind of pointer? Choose the correct name from pointers described in questions 25 to 27.
29. Can read-only pointers assigned to a program variable? Describe your answer.
30. Can read and write access pointers be assigned to named constants? Describe your answer.
31. Can static array be declared using a constant pointer? Describe.
32. Will the following code compile? If not then why not?

```
int arr1[2] = {0};
int arr2[2] = {0};
arr2 = arr1;
```

33. Will the following code compile? If yes then why?
`int arr1[2] = {0};`
`int * const arr2 = arr1;`
34. Will the following code compile? If yes then why?
`int arr1[2] = {0};`
`int * arr2 = arr1;`
35. Will the following code compile? If yes then why?
`int arr1[2] = {0};`
`const int * arr2 = arr1;`
36. In code above why the pointer arr2 cannot change the values of array elements stored in arr1?
37. Will the following code compile? If yes then why?
`int arr1[2] = {0};`
`const int * const arr2 = arr1;`
38. Programmers Zack and Zelda argue about as to what is the correct syntax of printing the array elements by de-referencing the array name. Zack says that since the first element can be printed by simply de-referencing the array name, the other elements can be printed as follows:
`cout<<*arr [n];` should print the n+1th element of the array. Zelda says that above syntax is incorrect. Who is right? What is the syntax Zelda may be referring to?
39. In an array of length N, a pointer is assigned to the element number N/2. Assuming the address stored in pointer to be X, derive the formula for the addresses of first and last element of the array assuming:
A. N is an even number.
B. N is an odd number.
Assume that the size of data type of the array is Y bytes.
40. You may wish to find out whether the `length()` function of string class works right or not. Write a function which takes a string as an argument and returns its length with out using the `length()` function. Check your answer by using the `length()` function.
41. Describe five methods of printing an array of integers. Two methods by using the array name and other three by using a pointer to its first element.
42. What is the last character stored in a C string?
43. What is the error in this piece of code?
`char ch = 'A';`
`char * ptr = ch;`
44. Even if the above code compile, what did programmer intended to do?
How can you fix the above code?
45. Given the definitions:
`double var1, *ptr1, *ptr2;`
`float * ptr3;`
`int var2, var4;`
What are the data types of
var1, ptr1, ptr1, ptr2, *ptr2, ptr3, *ptr3, var2, var4, and *var4?

Assume the following definitions and initializations:

```
char c = 'T';  
char d = 'S';  
char e;  
char * p1 = &c;  
char *p2 = &d;  
char * p3;
```

Assume that the address stored in c is 6940, address stored in d is 9772, and the one stored in e is 2224. What are the values of expressions in questions 46 to 54 after the following code lines are executed?

```
p1=&c;  
p2 = &d;  
p3=p1;  
p1 = p2;  
p2=p3;  
p3 = &e;  
*p3 = *p1;  
*p1 = *p2;  
*p2 = *p3;
```

- 46. &c**
- 47. d**
- 48. e**
- 49. c**
- 50. p2**
- 51. p3**
- 52. *p1**
- 53. *p2**
- 54. *p3.**