Lab Session 10

Pointers in C++ Language.

Objectives:

1. Illustration of Pointers.
2. To learn the syntax and semantics of the Pointer as arguments, and addressing.
3. Demonstrate a thorough understanding of Pointers as arguments, and addressing through logic building and implementing programs logic.

Introduction to Pointers

Up until now, we've only been able to work with a fixed amount of memory, an amount decided upfront before the program has even started. Whenever you declare a variable, it causes some amount of memory to be allocated behind the scenes to hold the information stored in that variable. When you declare a variable, the amount of memory allocated is chosen at compile time—you can't change it or add to it while the program is running. We've been able to create arrays of data to get a lot of variables—a big chunk of memory—but the array can hold no more elements than the number that you specified when writing the program. In the next few chapters, we'll learn about how to get access to more memory than we started our program with. You'll learn how to create an unlimited number of enemy spaceships all flying around at once (minus the flying around...).

In order to get access to (nearly) unlimited amounts of memory, we need a kind of variable that can refer directly to the memory that stores variables. This type of variable is called a pointer.

Pointers are aptly named: they are variables that "point" to locations in memory. A pointer is very similar to a hyperlink. A webpage is located in one place—on some person's web server. If you want to send someone a copy of that web page, do you download the entire page and email it to them? No, you just email a link. Similarly, a pointer allows you to store or send a "link" to a variable, array or structure, rather than making a copy.

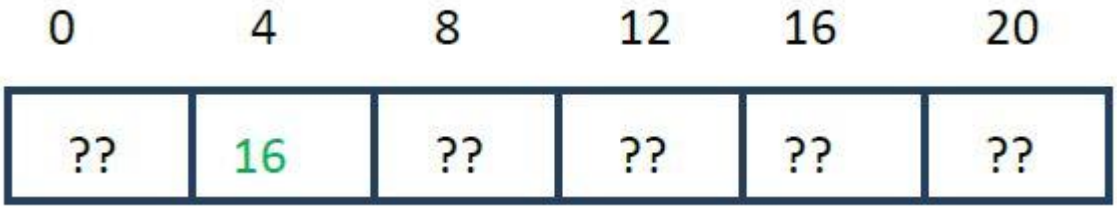
A pointer, like a hyperlink, stores the location of some other data. Because a pointer can store the location (the address) of other data, you can use it to hold on to memory that you get from the operating system. In other words, your program can ask for more memory and can access that memory using pointers.

What is memory?

An easy way to visualize computer memory is to think of an Excel spreadsheet. Spreadsheets are basically a large number of "cells" that can each store a piece of data. This, too, is what computer memory is: a large number of sequential pieces of data. Unlike Excel, in memory, each "cell" can store only a very small amount of data—1 byte, which can itself only store 256 possible values (0-255). Also unlike Excel, memory is organized "linearly" rather than in a grid. In fact, you can even think about memory as being a very long array of chars.

Just as each cell in an Excel spreadsheet has a way to locate it (using its row number and column letter), each cell in memory has an address. This address is the value that a pointer stores, when it holds the location of memory. (In Excel, a pointer would be a cell that holds the name of another cell—for example, if cell C1 held the string A1).

Here's a diagram that shows how you can think about a small chunk of memory. Notice that the diagram is a lot like an array; an array is just a bunch of sequential memory:



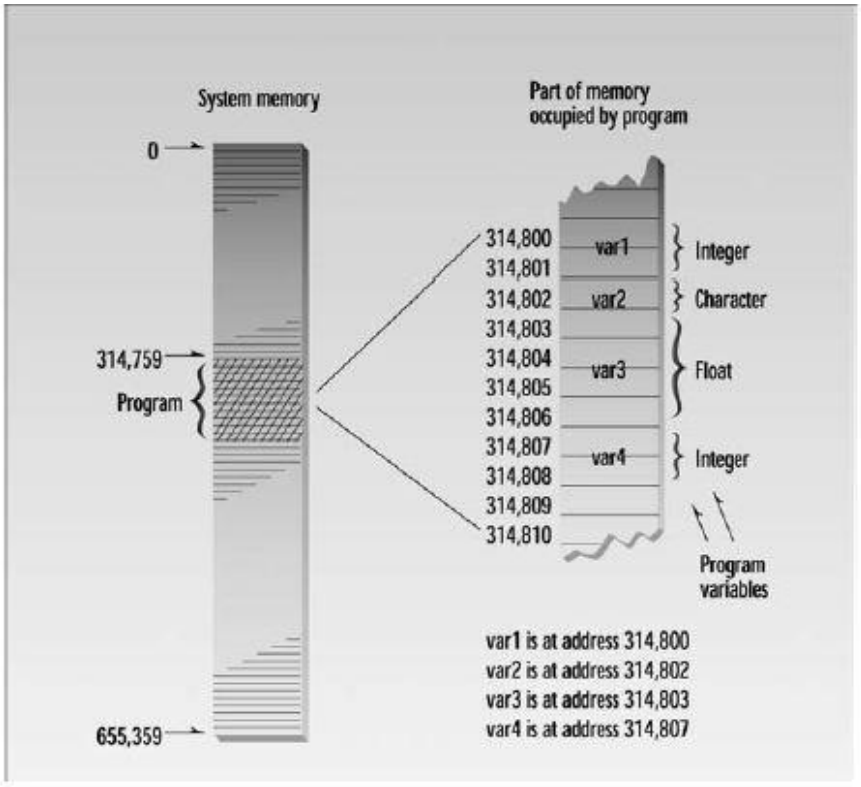
The boxes here represent locations in memory where data can be stored; the numbers are the **memory addresses**, which are the way to identify a location in memory. They're marked in steps of 4 because most variables in memory take up 4 bytes, so we're looking at the memory associated with 6 different variables of 4 bytes each.24 (By the way, you'll often see memory addresses written in hexadecimal form, which look quite a bit like gibberish if you've never seen them before; I'll use normal numbers.

Here, you can see the memory at address 4 stores a value that could be another memory address, 16. The memory at address 4 belongs to a pointer variable. The other values are marked as ?? to indicate that they don't have any particular known value, but of course there is something in each memory address at all times. Until that memory is initialized, the value is not useful—it could be anything.

Addresses and Pointers

The ideas behind pointers are not complicated. Here's the first key concept: Every byte in the computer's memory has an *address*. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575. (Of course you have much more.) Your program, when it is loaded

into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address. Figure 2 shows how this looks.



Variables vs. addresses

You might be confused by the distinction between a variable and an address. A variable is a representation of a value; that value is actually stored at a particular location in memory, at a particular memory address. In other words, the compiler uses memory addresses to implement the variables in your program. Pointers are a special kind of variable that lets you store the address that "backs" another variable.

The thing is that once you can talk about the address of a variable, you'll then be able to go to that address and retrieve the data stored in it. If you happen to have a huge piece of data that you want to pass into a function, it's a lot more efficient (when your program is running) to pass its location to the function than it is to copy every element of the data—just as we saw with arrays. We can also use this approach to avoid copying structures when passing them into functions. The idea is to take the address that stores the data associated with the structure variable and pass that address to the function instead of making a copy of the data stored in the structure.

The most important function of pointers is to enable you to get more memory at any time from the operating system. How do you get that memory from the operating system? 26 The operating system tells you the address of the memory. You need pointers to store the memory address. If you

need more memory later, you can just ask for more memory and change the value that you are pointing to. Consequently, pointers let us go beyond a fixed amount of data, letting us choose at run time how much memory we will need.

A note about terms

The word pointer can refer either to
1) A memory address itself
2) A variable that stores a memory address

Usually, the distinction isn't really that important: if you pass a pointer variable into a function, you're passing the value stored in the pointer—the memory address. When I want to talk about a memory address, I'll refer to it as a memory address or just an address; when I want a variable that stores a memory address, I'll call it a pointer. When a variable stores the address of another variable, I'll say that it is **pointing to** that variable.

Example 1

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Output:

```
firstvalue is 10
secondvalue is 20
```

Example 2

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

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Output

```
firstvalue is 10
secondvalue is 20
```

The Address-of Operator &

You can find the address occupied by a variable by using the address-of operator &. Here's a short program, VARADDR, that demonstrates how to do this:

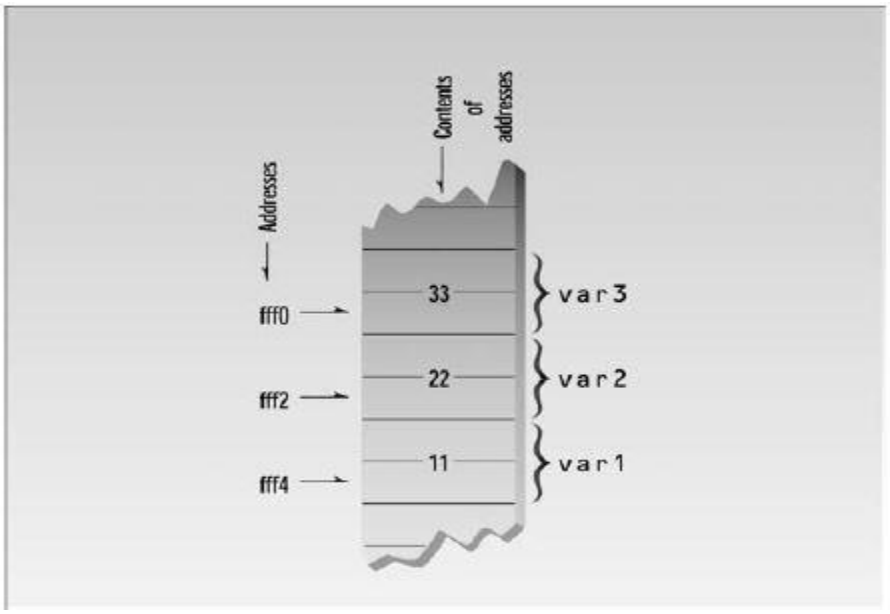
```
// varaddr.cpp
// addresses of variables
#include <iostream>
using namespace std;
int main()
{
        int var1 = 11; //define and initialize
        int var2 = 22; //three variables
        int var3 = 33;
        cout << &var1 << endl //print the addresses
        << &var2 << endl //of these variables
        << &var3 << endl;
        return 0;
}
```

This simple program defines three integer variables and initializes them to the values 11, 22, and 33. It then prints out the addresses of these variables.

The actual addresses occupied by the variables in a program depend on many factors, such as the computer the program is running on, the size of the operating system, and whether any other programs are currently in memory. For these reasons you probably won't get the same addresses we did when you run this program. (You may not even get the same results twice in a row.) Here's the output on our machine:

0x8f4ffff4 ← address of var1
0x8f4ffff2 ← address of var2
0x8f4ffff0 ← address of var3

Remember that the *address* of a variable is not at all the same as its *contents*. The contents of the three variables are 11, 22, and 33. Figure 2 shows addresses and contents of variables of the three variables in memory.



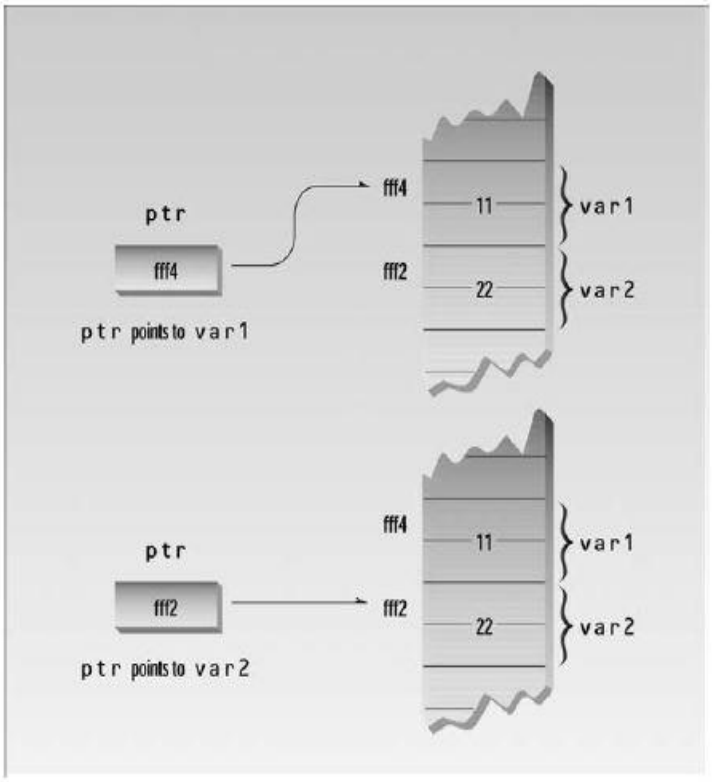
The << insertion operator interprets the addresses in hexadecimal arithmetic, as indicated by the prefix 0x before each number. This is the usual way to show memory addresses. If you aren't familiar with the hexadecimal number system, don't worry. All you really need to know is that each variable starts at a unique address. However, you might note in the output that each address differs from the next by exactly 2 bytes. That's because integers occupy 2 bytes of memory (on a

16-bit system). If we had used variables of type char, they would have adjacent addresses, since a char occupies 1 byte; and if we had used type double, the addresses would have differed by 8 bytes.

Pointers Must Have a Value

An address like 0x8f4ffff4 can be thought of as a *pointer constant*. A pointer like ptr can be thought of as a *pointer variable*. Just as the integer variable var1 can be assigned the constant value 11, so can the pointer variable ptr be assigned the constant value 0x8f4ffff4.

When we first define a variable, it holds no value (unless we initialize it at the same time). It may hold a garbage value, but this has no meaning. In the case of pointers, a garbage value is the address of something in memory, but probably not of something that we want. So before a pointer is used, a specific address must be placed in it. In the PTRVAR program, ptr is first assigned the address of var1 in the line.



ptr = &var1; ←□□□ put address of var1 in ptr

Following this, the program prints out the value contained in ptr, which should be the same address printed for &var1. The same pointer variable ptr is then assigned the address of var2, and this value is printed out. Figure 10.3 shows the operation of the PTRVAR program. Here's the output of PTRVAR:

0x8f51fff4 ← address of var1
0x8f51fff2 ← address of var2
0x8f51fff4 ← ptr set to address of var1
0x8f51fff2 ← ptr set to address of var2

To summarize: A pointer can hold the address of any variable of the correct type; it's a receptacle awaiting an address. However, it must be given some value, or it will point to an address we don't want it to point to, such as into our program code or the operating system. Rogue pointer values

can result in system crashes and are difficult to debug, since the compiler gives no warning. The moral: Make sure you give every pointer variable a valid address value before using it.

Memory layout

Where exactly does that memory come from? Why do you need to request it from the operating system, anyway?

In Excel, you have one very large group of cells that you can access. In computer memory, you also have a great deal of memory available. But that memory is more structured. Some parts of the memory available to your program are already in use. One part of memory is used to store the variables that you declare in the functions that are currently being executed—this part of memory is called the **stack**. Its name comes from the fact that if you make several function calls, the local variables for each function "stack up" on top of each other in this part of memory. All of the variables we've worked with so far have been stored on the stack.

A second part of memory is the free store (sometimes known as the heap), which is unallocated memory that you can request in chunks. This part of memory is managed by the operating system; once a piece of memory is given out, it should only be used by the original code that allocated the memory— or by code to which that address is provided by the memory allocator. Using pointers will allow us to gain access to this memory.

Being able to access this memory is powerful, but with great power comes great responsibility. Memory is a scarce resource. Not as scarce as it used to be before multiple gigabytes of RAM became standard, but it's still limited. Each piece of memory you have allocated from the free store should eventually be released back to the free store when your program no longer needs it. The part of the code responsible for releasing a particular piece of memory is called the **owner** of that memory. When the owner of memory no longer needs it—for example, in a space shooter game, if a ship is destroyed—the code that owns the memory should return it to the free store so that it can be given out to other code. If you don't return the memory, eventually your program will start to run out of memory, causing slowdowns or even crashes. You may have heard people complain (or seen yourself) that the Firefox web browser used too much memory, causing the browser to slow down to a crawl. That's because someone didn't free memory that they should have, causing what's known as a **memory leak.**

The concept of ownership is part of the interface between a function and its users—it is not explicitly part of the language. When you write a function that takes a pointer, you should document whether the function takes ownership of the memory or not. C++ will not track this for you, and it will never free memory that you have explicitly allocated while your program is still running unless you explicitly request it.

Pointer Variables

Addresses by themselves are rather limited. It's nice to know that we can find out where things are in memory, as we did in VARADDR, but printing out address values is not all that useful. The potential for increasing our programming power requires an additional idea: *variables that hold address values*. We've seen variable types that store characters, integers, floating-point numbers, and so on. Addresses are stored similarly. A variable that holds an address value is called a *pointer variable*, or simply a *pointer*.

What is the data type of pointer variables? It's not the same as the variable whose address is being stored; a pointer to int is not type int. You might think a pointer data type would be called something like pointer or ptr. However, things are slightly more complicated. The next program, PTRVAR, shows the syntax for pointer variables.

```
// ptrvar.cpp
// pointers (address variables)
#include <iostream>
using namespace std;
int main()
{
        int var1 = 11; //two integer variables
        int var2 = 22;
        cout << &var1 << endl //print addresses of variables << &var2 << endl << endl;
        int* ptr; //pointer to integers
        ptr = &var1; //pointer points to var1
        cout << ptr << endl; //print pointer value
        ptr = &var2; //pointer points to var2
        cout << ptr << endl; //print pointer value
        return 0;
}
```

This program defines two integer variables, var1 and var2, and initializes them to the values 11 and 22. It then prints out their addresses.

The program next defines a pointer variable in the line
int* ptr;

To the uninitiated this is a rather bizarre syntax. The asterisk means *pointer to*. Thus the statement defines the variable ptr as a *pointer to* int. This is another way of saying that this variable can hold the addresses of integer variables.

What's wrong with the idea of a general-purpose pointer type that holds pointers to any data type? If we called it type pointer we could write declarations like

pointer ptr;

The problem is that the compiler needs to know what kind of variable the pointer points to. (We'll see why when we talk about pointers and arrays.) The syntax used in C++ allows pointers to any type to be declared:

```
char* cptr; // pointer to char
int* iptr; // pointer to int
float* fptr; // pointer to float
Distance* distptr; // pointer to user-defined Distance class and so on.
```

Example:

The program 4 demonstrates the & and * pointer operators. Memory locations are output by << in this example as hexadecimal (i.e., base-16) integers. Number Systems. The hexadecimal memory addresses output by this program are compiler and operating-system dependent, so you may get different results when you run the program.

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

int main()
{
   int a; // a is an integer
   int *aPtr; // aPtr is an int * which is a pointer to an integer

   a = 7; // assigned 7 to a
   aPtr = &a; // assign the address of a to aPtr

   cout << "The address of a is " << &a
      << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is " << a
      << "\nThe value of *aPtr is " << *aPtr;
   cout << "\n\nShowing that * and & are inverses of "
      << "each other.\n&*aPtr = " << &*aPtr
      << "\n*&aPtr = " << *&aPtr << endl;
} // end main
```

Output

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Invalid pointers

One way that you can accidentally use invalid memory is to use a pointer without initializing it. When you declare a pointer, it will initially have effectively random data inside it—it will point to some place in memory that may or may not be valid, but it's certainly quite dangerous to use. In fact, it's almost as though you had generated a random number! Using this value will result in either a crash or data corruption. You must always initialize pointers before you use them!

Memory and arrays

Remember how I said that writing past the end of an array was a problem? Now that we know a bit more about memory, you can see why this would be. An array has a specific amount of memory associated with it, based on the size of the array. If you access an element past the end of the array, it's accessing memory that isn't associated with the array—that memory is, well, it's just not the array; exactly it is what will depend on the exact code and how the compiler is implemented. But it won't be part of the array, so using it will almost definitely cause problems.

Other advantages (and disadvantages) of pointers

Now that you've learned a bit about the details of pointers, let's go back to our previous analogy and look at some of the tradeoffs of using pointers. Hyperlinks and pointers have a lot of the same advantages and disadvantages.

1) You don't have to make a copy—if the web page is quite large or complicated, this could be hard (imagine trying to send someone a copy of all of Wikipedia!). Similarly, data in memory might be quite complicated, and it might be hard to copy correctly (more on this later) or just slow (copying a lot of memory may be time consuming).
2) You don't have to worry about whether you've got the latest version of the webpage. If the author updates the page, then you get the changes as soon as you revisit the link. If you have a pointer to memory, you are always able to access the latest value at that memory address.

Of course, there are also disadvantages to sending a link rather than a copy:

1) The page might be moved, or deleted. Similarly, memory can be returned to the operating system, even if someone still has pointer to it. To avoid these issues, the code that owns the memory must keep track of whether anyone else might be using it.
2) You have to be online to access it. This one generally doesn't affect pointers.

Applications of Pointers:

- To pass arguments by reference. ...
- For accessing array elements. ...
- To return multiple values. ...
- Dynamic memory allocation : We can use pointers to dynamically allocate memory. ...
- To implement data structures. ...
- To do system level programming where memory addresses are useful.

Task:

1:
For each of the following, write a single statement that performs the specified task. Assume that floating-point variables number1 and number2 have been declared and that number1 has been initialized to 7.3. Assume that variable ptr is of type char *. Assume that arrays s1 and s2 are each 100-element char arrays that are initialized with string literals.

a) Declare the variable fPtr to be a pointer to an object of type double.
b) Assign the address of variable number1 to pointer variable fPtr.
c) Print the value of the object pointed to by fPtr.
d) Assign the value of the object pointed to by fPtr to variable number2.
e) Print the value of number2.
f) Print the address of number1.
g) Print the address stored in fPtr. Is the value printed the same as the address of number1?

2:
Find the error in each of the following program segments. Assume the following declarations and statements:

```
int *zPtr; // zPtr will reference array z
void *sPtr = 0;
int number;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
```

a) ++zPtr;
b) // use pointer to get first value of array
number = zPtr;
c) // assign array element 2 (the value 3) to number
number = *zPtr[2];
d) // print entire array z
for (int i = 0; i <= 5; ++i)
cout << zPtr[i] << endl;
e) // assign the value pointed to by sPtr to number
number = *sPtr;
f) ++z;