CS 11 Data Structures and Algorithms

Lesson 12: Pointers

Skip to Main Content

Section 12.1: Introduction to Pointers

A pointer is a variable that instead of storing data stores the memory address of a second variable. It is called a pointer because we think of this memory address as a way for the pointer variable to "point" to the second variable. Despite the fact that you are only just now learning about pointers, they are really a very fundamental part of the C++ programming language. Because they can be quite complex, we have avoided discussing them until now.

Pointers have several important applications in programming. Probably the most important application is in linked data structures. Without pointers, we have only one option for how to store a large list of data: arrays. Arrays are very inflexible. You have to specify when you write the program how big the array will be, and this can be a big waste of memory. With linked data structures, each time we want to add a piece of data to our list we can create the memory for it and "link" it to the rest of the list using a pointer.

A second programming application of pointers is dynamic arrays. Dynamic arrays are similar to the arrays we discussed in lesson 9, but you don't have to specify the size of the array when you write your program. These dynamic arrays are more flexible than the non-dynamic arrays from lesson 9, since you don't have to specify when you write the program how big the array will be, but they are less flexible than linked data structures since, unlike linked data structures, you still have to specify how big the array will be before you use the array for anything.

A third programming application of pointers is in C-strings. Using C-strings is an alternate technique for storing strings. It is also a much older technique. Lots of existing C++ code uses C-strings rather than the string class because the string class has only been part of the C++ standard since the late 1990s. C-strings are usually implemented as **dynamic** arrays of characters, and so you will need to understand pointers to use them.

One more word of warning. Pointers are incredibly useful; however, there are a lot of technical details that need to be understood before getting to the good stuff, and it's hard to see how they could be of any use while you are wading your way through the muck. So pay close attention to these details and take my word for it that you'll get to put them to good use before we are done!

Section 12.2: Pointer Basics

12.2.1 Declare a Pointer

The following statement declares a variable of type "pointer to int":

```
int* intptr;
```

When you declare a pointer, you must specify the type of variable that will be pointed to. You will get a syntax error if you then try to make it point to a variable of another type.

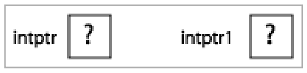
The asterisk in the declaration statement can go either next to the type or next to the variable name. Be careful, however, if you are going to declare two pointers on the same line. You might try this:

```
int* intptr, intptr1;
```

This will not, however, give you two pointers to int. It will give you one pointer to int (intptr) and one regular int (intptr1). The reason is that the compiler associates the asterisk with the variable name, not with the type. For this reason, I think that it is better to always declare each pointer variable on a separate line:

```
int* intptr,
int* intptr1;
```

After these two declarations, we picture the situation like this:

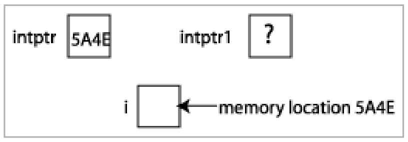


12.2.2 The "Address Of" Operator

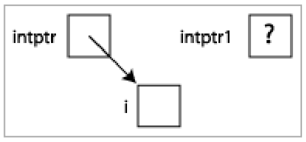
Let's create an int variable and make intptr point to it. The statements to do this are:

```
int i;
intptr = &i;
```

The & operator is called the "address of" operator and the second statement above is read "intptr gets the address of i". In order to see what actually happens in the computer's memory at this point, we need to know what i's address is. Let's assume for a moment that i's address is 5A4E (addresses are usually given in hexadecimal and we're adopting this convention). After we execute the two statements above, our picture will look like this:



Fortunately, programmers don't usually think about specific memory addresses like 5A4E when we are dealing with pointers. Essentially, storing the address of i is simply storing a way for one variable (the pointer variable) to find another variable (the int variable). When we think about pointer variables or draw pictures of them (which we do a lot a lot a lot), we don't use memory addresses. Instead, we draw a picture of an arrow originating in the pointer variable and pointing at the "pointed to" variable, like this:



12.2.3 The "Dereference" or "Indirection" Operator

Our next task is to assign to i the value 47. Of course, we could easily do that with this statement:

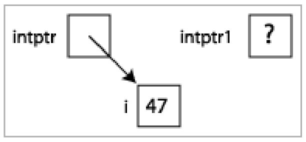
i = 47;

This would be an example of "direct addressing". But for the sake of illustration we would like to do this assignment indirectly (i.e. using "indirect addressing"), using only intptr (not using i). Here is the statement that performs this task:

*intptr = 47;

Here the asterisk is used in a completely different way than it was used in the declaration of pointers. Don't try to think of this as two different uses of the same operator; it will only confuse you. Think of it as two different operators. The fact that the asterisk is used for both purposes is irrelevant. The name of the operator in this case is the "dereference" or "indirection" operator. The name "indirection operator" should make sense, since the operator provides a way to access variables without using the name of the variable itself (i.e. a way to access variables indirectly).

To access a variable that a pointer points to, we place the indirection operator in front of a pointer variable. So in our little example we are saying "assign the value 47 to the variable that intptr is pointing to". After executing this statement, our picture of the situation will look like this:



Let's add a cout statement to our code so that we have some output to look at, and run our program. Here is the code we have created so far, along with the output I get when I run the program on my computer:

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

int main()
{

    // 12.2.1 Declare a Pointer

    int* intptr;
    int* intptr1;

    // 12.2.2 The "Address Of" Operator

    int i;
    intptr = &i;

    // 12.2.3 The "Dereference" or "Indirection" Operator

    *intptr = 47;                // indirect addressing
    // equivalent to: i = 47     // direct addressing
    cout << intptr << " " << i << " " << *intptr << endl;
}
```

```
// OUTPUT

0x0bf59918 47 47
```

The first number above may look a bit mysterious. Remember that memory addresses are typically given in hexadecimal instead of decimal notation. Numeric constants that begin with 0x in C++ are hexadecimal numbers.

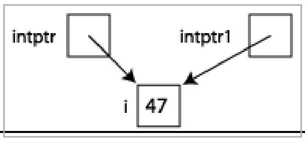
If you attempt to dereference a pointer which is uninitialized you will get an unhandled exception, so it is important to keep careful track of the state of all of your pointers. In our diagrams we will always indicate an unitialized pointer by putting a question mark (?) in the box.

12.2.4 Assignment of Pointers

Here is a statement that assigns one pointer to another:

intptr1 = intptr;

This copies the memory address that was stored in intptr into the pointer variable intptr1. The result of this in terms of our picture of the situation is that intptr1 now points to whatever intptr was pointing to. When you see one pointer assigned to another, you should think of it as a redirection of the arrows that we draw in our pictures. Data is never changed, only the arrows change. The result of adding this assignment statement to the end of the code that we have developed so far is pictured here:



Here's our updated code, with a cout statement added so that we can see the results of our assignment statement:

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

    int main()
    {

        // 12.2.1 Declare a Pointer

        int* intptr;
        int *intptr1;

        // 12.2.2 The "Address Of" Operator

        int i;
        intptr = &i;

        // 12.2.3 The "Dereference" or "Indirection" Operator

        *intptr = 47;                // indirect addressing
        // equivalent to: i = 47     // direct addressing
        cout << intptr << " " << i << " " << *intptr << endl;


        // 12.2.4 Assignment of Pointers

        // The following statement should be pronounced "make
        // intptr1 point to whatever intptr is pointing to"

        intptr1 = intptr;
        cout << intptr1 << " " << i << " " << *intptr1 << endl;
    }
```

```
    // OUTPUT

    0x0bf59918 47 47
    0x0bf59918 47 47
```

12.2.5 Pointers to Objects

Pointers to class objects can be declared and used in exactly the same way that pointers to the primitive types like int or char are. There is one potential pitfall to watch out for. Consider the following code:

```
string* strptr1;
string str1("hello");
strptr1 = &str1;
cout << *strptr1.substr(2, 3) << endl;
```

If we add this code to the end of our program (and also add "#include <string> to the top of the file) and attempt to compile it, we an error message something like this:

```
Error   : expression syntax error

pointers.cpp line 35   cout << *strptr1.substr(2, 3) << endl;
```

The reason is that in C++ the dot operator (.) has a higher precedence than the dereference operator (*), and the dot operator cannot be applied directly to a pointer. To make this expression work, we would have to put parentheses around the *strptr1:

```
cout << (*strptr1).substr(2, 3) << endl;
```

This would solve the problem, but there is a better solution. Because this sequence of dereferencing (with *) and then selecting (with .) happens so frequently in programming, C++ provides a shortcut:

```
cout << strptr1 -> substr(2, 3) << endl;
```

This statement is exactly equivalent to the previous statement. The -> operator (that's a dash followed by a greater-than symbol) does two things: first it dereferences the pointer, and then selects.

Let's update our program and see the output:

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

    int main()
    {
        // 12.2.1 Declare a Pointer

        int* intptr;
        int *intptr1;

        // 12.2.2 The "Address Of" Operator

        int i;
        intptr = &i;

        // 12.2.3 The "Dereference" or "Indirection" Operator

        *intptr = 47;                // indirect addressing
        // equivalent to: i = 47     // direct addressing
        cout << intptr << " " << i << " " << *intptr << endl;

        // 12.2.4 Assignment of Pointers

        // The following statement should be pronounced "make
        // intptr1 point to whatever intptr is pointing to"

        intptr1 = intptr;
        cout << intptr1 << " " << i << " " << *intptr1 << endl;

        // 12.2.5 Pointers to Objects

        string* strptr1;
        string str1("hello");
        strptr1 = &str1;

        // cout << *strptr1.substr(2, 3) << endl;  won't work here
        // because "." has higher precedence than "*"

        cout << (*strptr1).substr(2, 3) << endl;
        cout << strptr1 -> substr(2, 3) << endl;
        cout << endl;
    }
```

```
    // OUTPUT

    0x7fff5c21aa9c 47 47
    0x7fff5c21aa9c 47 47
    llo
    llo
```

12.2.6 Arrays of Pointers

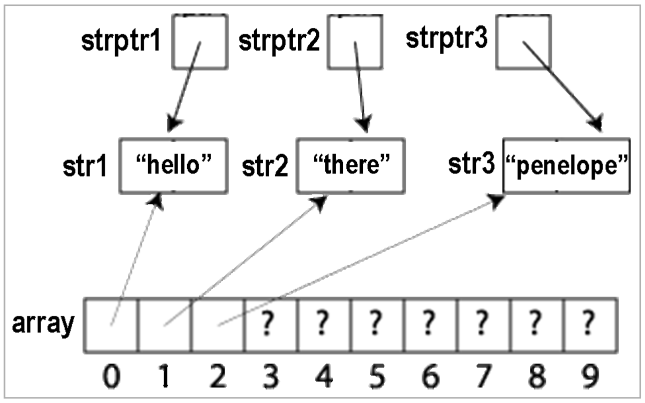
Let's illustrate using an array of pointers. Just to make things even more interesting, we'll make it an array of pointers to objects. First we'll declare two string objects and two pointers to string objects and then make the pointers point to the objects (we'll also assume that we still have str1 and strptr1 from the program above):

```
string* strptr2;
string* strptr3;
string str2("there");
string str3("penelope");
strptr2 = &str2;
strptr3 = &str3;
```

Next let's declare an array of pointers to string objects and initialize the first three elements of the array to point to the string objects we just created.

```
string* array[10] = {strptr1, strptr2, strptr3};
```

Here is a picture of what we have done so far.



Now let's do an assignment and print the results:

```
*array[0] = *array[1] + *array[2];
cout << *array[0] << " " << *array[1] << " " << *array[2] << endl;
```

Here is the updated picture.

12.2.7 nullptr

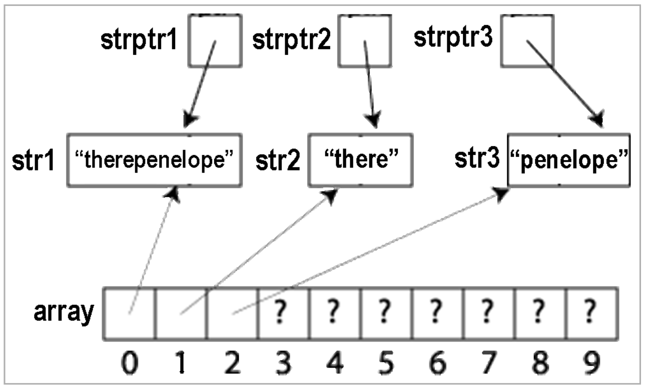
It is not the case that when you first declare a pointer it points to nothing. When you first declare a pointer, it is unitialized, just like any other variable is when you first declare it. Sometimes it is desirable to represent a pointer that points to "nothing". To do this, we set the pointer to "nullptr", like this:

```
array[0] = nullptr;
```

When a pointer has a value of nullptr we draw a line through the variable's box, like this:

The value nullptr is a relatively new addition to the C++ programming language. In practice you will very commonly see the value NULL used in the place of nullptr. At this writing I have not updated most of the lessons, so you will see NULL there. nullptr is better, and I would recommend using it instead of NULL.

12.2.8 Array Name Without [ ] Is a Pointer Constant



In C++ arrays are actually pointer constants. To illustrate this, let's declare an array of ints and a pointer to an int:

int array2[10];
int* ptr;

After this declaration, array2 is actually a pointer to the beginning location of an array. Because of this, we can assign ptr to array2:

ptr = array2;

To put this another way, this statement is exactly equivalent to saying

ptr = &array2[0];

Now that ptr and array2 are the same thing, we can use them interchangeably. For example, if there is a function named init that takes an array of ints as its argument, we could call that function with either ptr or array2. The statements

init(ptr);

and

init(array2);

would be exactly equivalent. To print out the contents of the array we could say either

```
for (int i = 0; i < 10; i++){
    cout << array2[i];
}
```

or

```
for (int i = 0; i < 10; i++){
    cout << ptr[i];
}
```

In addition, the declaration of the parameter in the function init could be for either a pointer or an array, and regardless of which way it is declared it can be used like an array in the body of the function (see the definition of init in the code below). The one thing that we cannot do is reassign array2, because an array name without the square brackets is not just a pointer, it is a pointer ***constant***. So the statement

array2 = ptr

will result in a compiler error.

Here is a program that incorporates all of the code we have seen so far.

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

void init(int* x);

int main()
{

    // 12.2.1 Declare a Pointer

    int* intptr;
    int *intptr1;




    // 12.2.2 The "Address of" Operator

    int i;
    intptr = &i;




    // 12.2.3 The "Dereference" or "Indirection" Operator

    *intptr = 47;                // indirect addressing
    // equivalent to: i = 47     // direct addressing
    cout << intptr << " " << i << " " << *intptr << endl;
```

```
        // 12.2.4 Assignment of Pointers

        // The following statement should be pronounced "make
        // intptr1 point to whatever intptr is pointing to"

        intptr1 = intptr;
        cout << intptr1 << " " << i << " " << *intptr1 << endl;




        // 12.2.5 Pointers to Objects

        string* strptr1;
        string str1("hello");
        strptr1 = &str1;

        // cout << *strptr1.substr(2, 3) << endl;  won't work here
        // because "." has higher precedence than "*"

        cout << (*strptr1).substr(2, 3) << endl;
        cout << strptr1 -> substr(2, 3) << endl;
        cout << endl;



        // 12.2.6 Arrays of Pointers

        string* strptr2;
        string* strptr3;
        string str2("there");
        string str3("penelope");
        strptr2 = &str2;
        strptr3 = &str3;

        string* array[10] = {strptr1, strptr2, strptr3};

        *array[0] = *array[1] + *array[2];
        cout << *array[0] << " " << *array[1] << " " << *array[2] << endl;



        // 12.2.7 nullptr

        array[0] = nullptr;


        // 12.2.8 Array Name Without [ ] Is a Pointer Constant

        int array2[10];
        int* ptr;

        ptr = array2;    // equivalent to ptr = &array2[0];

        init(ptr);       // equivalent to init(array2);
        for (int i = 0; i < 10; i++){
            cout << array2[i] << " ";        // equivalent to ptr[i]
        }
        cout << endl;

        // can't say "array2 = ptr;" because array2 is a constant
    }


    void init(int* x)        // equivalent to init(int x[])
    {
        for (int i = 0; i < 10; i++){
            x[i] = i;
        }
    }
```

```
    // OUTPUT

    0x7fff53212934 47 47
    0x7fff53212934 47 47
    llo
    llo

    therepenelope there penelope
    0 1 2 3 4 5 6 7 8 9
```

Section 12.3: Dynamic Memory

12.3.1 Categories of Data:

So far we have studied 2 types of variables in C++: automatic variables and static variables. To review, automatic variables are local variables, like the ones we have been using in all of our programs. They are called automatic because C++ automatically allocates memory for them when a block of code (usually a function) is entered and automatically deallocates (or frees) their memory when the end of a block is reached. The second type is the static variable. Static variables are variables that are allocated at the beginning of program execution and are not deallocated until execution stops. Global variables (which you have been encouraged to avoid) are static variables, and you can also declare a local variable to be static by placing the reserved word "static" in front of its declaration.

We now need to discuss the third way of using memory in C++: dynamic variables. With dynamic variables the programmer has complete control over when the variables are allocated and deallocated. This results in a great deal of flexibility, but it is also a lot more work for the programmer. Automatic variables are allocated in a highly structured area of memory called the stack. Each time a function is called new memory is allocated on the stack for all of the new variables, and when control returns to the calling function that memory is released so that it can be used by subsequent function calls. Dynamic variables, on the other hand, are allocated in a less structured area of memory called, appropriately, the heap (or the free store). When the programmer allocates memory for a dynamic variable using the new operator (explained below) the computer is free to choose any piece of available memory in the heap. That memory is then unavailable for other uses until the programmer explicitly releases the memory using the delete operator (also explained below). The use of dynamic memory, and particularly dynamic arrays, is an important programming technique, and cannot be done without the use of pointers.

12.3.2 The new operator

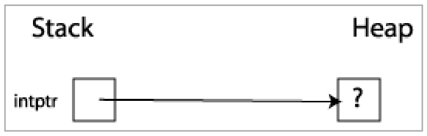
The reason that pointers are necessary when working with dynamic data is that variables allocated on the heap do not have names; that is, they can only be accessed indirectly, through a pointer that points to them. If we want to allocate an int on the heap, we first have to declare a pointer to int variable on the stack, and then allocate an integer variable on the heap and let the stack variable point to it. The code to accomplish this looks like this:

```
int* intptr;
intptr = new int;
```

The new operator does 2 things. First it allocates a new variable, of the type specified, on the heap. Then it returns a pointer to that variable. To put this another way, the statement intptr = new int; creates a new variable of type int and makes intptr point to it. You can also combine these two statements into one, like this:

```
int* intptr = new int;
```

Here is a picture of the situation created by these two statements:

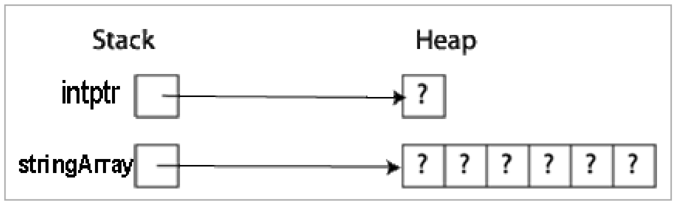


If not enough memory is available to carry out the new operation, your program will crash with an unhandled exception. (Later we will study how to "handle" the exception so that the program doesn't crash, but for now it's safe to assume your program will simply crash.)

You can also allocate an array of variables on the heap with a single use of the new operator. This is how you create a dynamic array. The advantage of using a dynamic array instead of the non-dynamic arrays we have studied previously is that the size of the array can be determined as the program is running. For example, we could ask the user to enter the size of the array. With non-dynamic arrays, we were required to decide on the size of the array as we were writing the program. To create a dynamic array with a base type of string with the size determined by the user, you would use the following statements:

```
cout << "Enter the size of the array: ";
cin >> arraySize;
string* stringArray = new string[arraySize];
```

which would result in the situation pictured below.

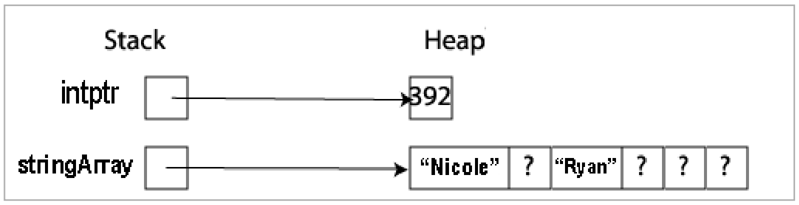


In the code below we have given values to the two variables that we allocated on the heap and printed them out. Below the code you will find a picture of the situation created by the program. Students are sometimes confused by the fact that stringArray is not dereferenced in the last line of code below. The reason is that the square brackets in C++ actually cause the pointer to be dereferenced. In other words, the expression "stringArray[2]" actually means "dereference the pointer variable stringArray, and then move over 2 elements in the array." If stringArray was dereferenced (using the expression *stringArray), the expression would refer to only the first element in the array, since stringArray is a pointer to that element. (This is illustrated in the program below.)

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

int main()
{
    int* intptr = new int;

    // new does 2 things: (1) allocate a variable of the
    // specified type on the heap (2) return a pointer
    // to that variable.  Put another way:  create a new
    // variable on the heap and make intptr point to it.

    cout << "Enter the size of the array: ";
    cin >> arraySize;
    string* stringArray = new string[arraySize];

    *intptr = 392;
```

```
        stringArray[0] = "Nicole";
        stringArray[2] = "Ryan";

        cout << *intptr << " "
             << *stringArray << " "
             << stringArray[2] << endl;
    }
```

```
    // OUTPUT

    392 Nicole Ryan
```



12.3.3 The delete operator

When a variable is allocated using the new operator, it is the programmer's responsibility to also deallocate the variable when it is no longer needed, so that the memory can be used for other purposes. Whenever you use the new operator in your code, you should make sure to check that there is a delete operator somewhere that deallocates the variable. This is done using the delete operator. This is particularly tough to get right, since with any program that is not huge (by our standards), you could never use the delete operator and your program will almost certainly run just fine, because it doesn't require nearly as much memory as is available on the heap. So, unlike most of the concepts you learn in computer programming, the computer will not typically tell you if you got this one wrong. Be very careful to use the delete operator correctly.

Here are the statements that will deallocate the memory that we have allocated so far:

```
delete intptr;
delete [] stringArray;
```

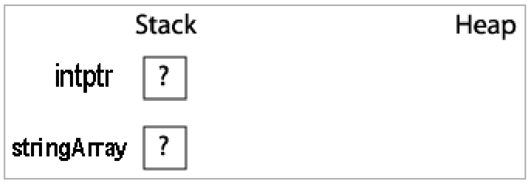
Consider the first of the two statements for a moment. The syntax can be a bit misleading since it looks like we are deleting the variable intptr, when actually we are deleting the variable that intptr is pointing to. This delete statement should be pronounced "delete the variable that intptr is pointing to". After the statement is executed, the variable intptr is considered uninitialized. If you attempt to dereference it with the dereference (*) operator, you will get an unhandled exception.

A second reason that the syntax can be misleading is that nothing is actually being "deleted". What is happening is that the variable that intptr is pointing to is being ***deallocated***. In other words, we are telling the runtime environment that we are done with that variable and it can now be used for some other purpose.

Now let's look at the second statement. When the new operator is used to allocate an array of variables, this form of the delete statement must be used in order to deallocate the entire array. If you leave out the square brackets, only the first element of the array will be deallocated.

It is ok to delete a nullptr pointer. The statement will be treated as a no-op (that is, it will have no effect). If you try to delete an uninitialized pointer, however, you will get an unhandled exception.

Here is a picture of the situation after the two delete statements above are executed:



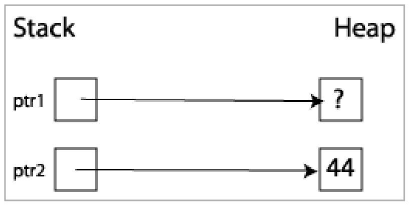
12.3.4 Memory Leaks

A memory leak is what we call the situation where a piece of memory on the heap becomes inaccessible; that is, it somehow ends up with no pointers pointing to it. This piece of memory, then, cannot be used by our program, but it also cannot be allocated for another use. It is just wasting space. It is important to avoid this situation.

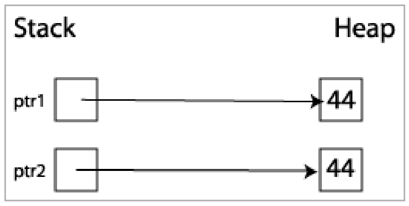
Consider the following code:

```
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;
*ptr1 = *ptr2;

ptr1 = ptr2;
```

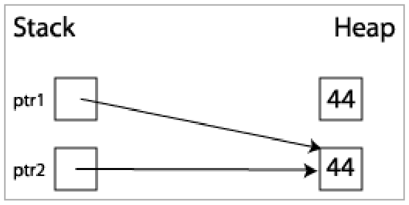
You might want to see if you can draw a picture of what happens with the execution of each of these statements before reading on. Here is a picture of the situation after the execution of the first three statements:



Consider the fourth statement. This could be read "the variable that ptr1 points to gets the variable that ptr2 points to". The result is that the value stored in the variable ptr2 is pointing to, namely 44, gets stored in the variable that ptr1 is pointing to, resulting in this situation:



The last statement should be pronounced "make ptr1 point to the variable that ptr2 is currently pointing to". The result is this:



Notice that in this picture the variable formerly pointed to by ptr1 is now inaccessible. This is a memory leak. To avoid this situation, we should have deleted ptr1 before assigning it to point to a different variable. If we insert the statement

```
delete ptr1;
```

in the code above right before the reassignment of ptr1, we end up with the same situation we did before except that the variable formerly pointed to by ptr1 will be missing from the picture -- indicating that it is now free to be allocated for some other variable.