**Chapter 10: Pointers**

Table of Contents

[Pointer to void 3](#_Toc49597987)

[Pointers and Arrays 4](#_Toc49597988)

[Pointers to String Constants 5](#_Toc49597989)

[const Modifiers and Pointers 6](#_Toc49597990)

[Pointers and Objects 7](#_Toc49597991)

[Memory Management 8](#_Toc49597992)

Pointers hold the address of a particular variable.

int \*ptr; *//declaring the pointer*int p = 10;  
ptr = &p; *//assigning memory address*cout<<ptr; *//prints address;*cout<<\*ptr; *//prints value at address, i.e. value of p*

C++

The & operator is called the address-of operator, and is used to access the address of a variable. The \* operator is called the pointer to operator when used in the declaration, and the dereference operator, or indirection operator, or contents of operator, and essentially means 'the value of the variable pointed to by', when used to access the value of the variable being pointed to.

Variables of different types store data in different ways. Thus, we need to specify the type while declaring a pointer, so that the memory address it will store can be interpreted in the correct manner.

## Pointer to void

It is possible to create a pointer with no data type, that can point to any data type.

void \*ptr;

C++

It is also possible to convert data of one type into another type before setting its address to a pointer.

ptrint = reinterpret\_cast<int\*>(flovar);  
ptrflo = reinterpret\_cast<float\*>(intvar);

C++

## Pointers and Arrays

Arrays take up consecutive spaces in memory. Thus, if a pointer is set to the first element, we can simply increment it to get the second element like this:

int ar[10];  
int \*ptr = ar; *//points to ar[0];*ptr++; *//points to ar[1];*

C++

Incrementing a pointer like this does not necessarily mean increasing the address by 1, say from 100 to 101. Integers for example, take up 4 bytes of memory, so incrementing the pointer would increase the address by 4 bytes, say from 100 to 104.

## Pointers to String Constants

Writing this:

char str1[] = "Hello";

C++

is the same as writing this:

char \*str2 = "Hello";

C++

The difference is that the array form is a pointer constant, an address itself, and thus cannot be changed, while the pointer form is a pointer variable, directions to an address, and can be changed. If we simply write str2++, the ‘H’ will disappear. Every other function, including accessing characters at specific indexes, is the same.

## const Modifiers and Pointers

const int \*ptr;

C++

Here, ptr points to a const integer, so we cannot change the value of what ptr points to, but we can change what ptr points to.

int \* const ptr;

C++

Here, ptr itself is a const, so we cannot make it point to something else, but we can change the value of what it is pointing to.

## Pointers and Objects

When we create a pointer to an object, it points to the memory address of the object. The data contained within the object has other, separate memory addresses.

MyClass \*ptr;  
ptr = &obj; *//ptr now points to an object, obj, of MyClass*

C++

An important thing to note about pointers, related to classes and structures, is that while they can hold pointers to objects of their own type, they cannot hold their own objects.

class SomeClass  
{  
 SomeClass \*ptr; *//this is fine*};  
class SomeClass  
{  
 SomeClass obj; *//this is not; error*};

C++

## Memory Management

The C language had a very difficult process to allocate memory dynamically. C++ makes it much easier.

MyClass \*ptr;  
ptr = new MyClass();

C++

The keyword new is used to allocate memory for an object of MyClass and ptr is made to point to it. Notice that the object has no name and cannot be accessed without ptr. Thus, if we made ptr point to something else, we would lose access to this object. This creates a very dangerous situation, since we allocated memory and then lost track of the object. This can cause a lot of problems. Thus, whenever we allocate memory like this, we need to make sure to remove the memory space before ending the program, since it will not be removed automatically. We can free the memory space like this:

delete ptr;

C++

Note that this does not clear the memory space, but makes the data there invalid. If we somehow printed the data in that specific memory space, there is a good chance that we will no longer get the data we originally had since it has been overwritten.

Destructor functions come in handy in this scenario. We can use them to automatically clear up memory used by our objects. Consider the following class:

class String  
{  
 char \*ptr;  
public:  
 string (const char \*ch)  
 {  
 int len = strlen(ch);  
 ptr = new char[len+1]; *//allocating memory* strcpy(ptr, ch);  
 }  
 ~String()  
 {  
 delete[] ptr; *//remove memory* }  
};

C++

Notice that delete[] is being used here instead of delete. This is done since ptr points to an array of characters. delete[] is used to delete arrays while delete is used to delete non-array objects.