C++ Pointers

A pointer is a C++ data type. The value that a pointer can hold is a memory location. We say that a pointer variable *points* to the memory location. There are actually many different data types that are pointers, depending on the data type of the value stored in the memory location to which the pointer points. For example, an **`int`** pointer points to an **`int`** value in memory, a **`double`** pointer points to a **`double`** value stored in memory.

Declaring Pointers

Unlike other data types in C++, the pointer types aren't given a specific name. This means that they are declared in a slightly different way. The syntax to declare a pointer is

*dataType *identifier;*

To declare an **`int`** pointer you would have

```
int *k;
```

And to declare a **`double`** pointer, you would have

```
double *y;
```

In the above two examples, the asterisk (`*`) indicates that the identifier is a pointer to the specified type.

All three of the following declarations are equivalent:

```
int *k;

int* k;

int  *  k;
```

In the following statement, only **`i`** is a pointer. Variable **`j`** is an **`int`**, not an **`int`** pointer.

```
int* i, j;
```

To declare both **`i`** and **`j`** as **`int`** pointers, you would have to do this

```
int *i, *j;
```

Assigning Values to Pointers

Since a pointer holds a memory location, we use the address of operator **&** to assign a value to a pointer, like this

```
int x = 7; int *p = &x;
```

Here, **x** is an **int** variable and **p** is a pointer that holds the memory location where the value of **x** is stored. We say the **p** points to **x**. To access the value of **x** using the pointer **p**, we dereference the pointer like this

```
cout << *p;
```

Which causes the value of **x** to be displayed. The program in Listing 8.1 and its output in Figure 8.1 demonstrate how a pointer works.

Programming Example: Pointers1.cpp

Pointers are related to references.

Programming Example: Pointers2.cpp

Reference Parameters Using Pointers

We saw earlier how to use references to pass parameters to functions "by reference". The same thing can be done using pointers.

Programming Example: Pointers3.cpp

There are four ways to pass a pointer to a function:

A non-constant pointer to non-constant data

This is what we've seen so far. The pointer can be dereferenced to modify the object to which it points. The pointer itself can be changed to point to different objects. Programming Example: Pointers4.cpp

A non-constant pointer to constant data

Here the object pointed to can no longer be modify through the use of the pointer, but the pointer can be made to point to different objects.

A constant pointer to non-constant data

The object pointed to can be changed, but the pointer can no longer be made to point to different objects.

A constant pointer to constant data

The object pointed to cannot be changed, nor can the pointer be made to point to different objects.

The `sizeof` Operator

The C++ sizeof operator returns the number of bytes of any valid data type. This can be used to determine the number of elements in an array.

Programming Example: Pointers5.cpp

Pointers and Arrays

As we saw earlier, an array is simply a reference to a memory location. Arrays and pointers are therefore closely related. When a pointer references an array, we can do *pointer arithmetic*.

Programming Examples: Pointers6.cpp and Pointers7.cpp

Casting Pointers

You can cast a pointer of one type to a pointer of another type.

Programming Example: Pointers8.cpp

Function Pointers

C++ allows a pointer to a function. When declaring a pointer to a function the following must be specified:

- The function's return type
- The number of arguments the function takes
- The type of each argument

Programming Examples: Pointers9.cpp and Pointers10.cpp

Void Pointers

As we've seen, every pointer points to a particular *type* of data stored in memory. That is why an **`int`** pointer is of a different type than a **`double`** pointer. C++ also allows for a generic pointer, that can point to any type of data stored in memory. This generic pointer is called a **`void`** pointer. In order to dereference a void pointer you must first cast it to the type of data pointed to. Another feature of void pointer is that you can copy a pointer of any type to a void pointer.

Programming Example: Pointers11.cpp

NULL Pointers

A **`NULL`** pointer is a regular pointer, of any pointer type, which contains the value **`NULL`**. When a pointer contains **`NULL`**, it means it is not pointing to any valid reference or memory address. Do not confuse **`NULL`** pointers with void pointers. A NULL pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

Programming Example: Pointers12.cpp

Dynamic Memory Allocation

In all of the examples we've seen so far, whenever we declared variables, either simple numeric data types or instances of objects, the memory for those variables was automatically allocated for us. Likewise, when that memory was no longer needed for our variables it was automatically deallocated. In this section we are going to learn how to handle these memory management tasks ourselves.

Simple Data Types

The **new** operator allocates memory of the proper size. To dynamically allocate memory for an **int** and a **double** variable, you would do this

```
int *iPtr = new int(5);

double *dPtr = new double;
```

The **int** is assigned an initial value while the **double** is not.

When you manually allocate memory with **new** you are then responsible for releasing that memory by using the **delete** command before your program terminates. Failure to release dynamically allocated memory can lead to a memory leak in your program. To delete the **int** and **double** allocated above, you would do this

```
delete iPtr;

delete dPtr;
```

Programming Example: Dynamic.cpp

Dynamic Arrays

Earlier we created arrays like this:

```
int arr [ARR_SIZE];
```

where **ARR_SIZE** is a constant integer expression.

With dynamic memory allocation, the size of the array doesn't need to be specified at compile time. Rather, it can be specified at run time. This allows for much more flexibility in your programs. The syntax for deleting an array that was created dynamically is

```
delete arr;
```

Programming Example: DynamicArray.cpp

Dangling Pointers

A dangling pointer is a pointer to memory that is no longer allocated. Dangling pointers are a serious problem because they seldom crash the program until long after they have been created. This makes them hard to find. A program that creates dangling pointers may work fine with small inputs, but is likely to crash on large inputs. The example below shows how a dangling pointer can be created.

Programming Example: DanglingPointers.cpp

Pointers to Objects

A pointer can also point to an object. One important thing to remember is that with a pointer to an object, you use **->** to access the objects public members rather than using the "dot" operator as we normally do.

Programming Example: PointerToObject.cpp

Command Line Arguments

When a program is executed, it can be sent optional command line arguments. For example, if we have a program called **`args.exe`**, we can pass arguments to this program when we it from a command prompt like this

```
      args Source.dat dest.txt /t
```

In this example three command line arguments are being passed to the program args.exe. The arguments are (i) "Source.dat", (ii) "dest.txt", and (iii) "/t". Most IDEs can be configured to pass command line arguments to a program when it is executed.

To access the command line arguments from within the program, you must add two arguments to the main function like this:

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

The first argument is an **`int`** named **`argc`**, which stands for "argument count". This shows the number of command line argument. The second argument is an array of C-style strings names **`argv`**, which stands for "argument values". The length of the **`argv`** array is given by the **`argc`** integer.

Programming Example: args.cpp