Object Oriented Programming  
Tutorial 2 - Introduction to Pointers

## Introduction

When you declare a variable in your program it is allocated a fixed area of the computer memory until it falls out of scope, or the program ends. Whilst obviously useful, on their own they are quite inflexible. For instance, you have to allocate **all** the memory required at **compile-time**, rather than **run-time**. Sometimes it makes more sense to decide how much memory is required at run-time, to cater to the current demands of any given program execution. This **dynamic memory allocation** capability is provided using **pointers**. Pointers also provide the programmer with a very efficient way of accessing and manipulating data.

Whilst in principle the use of pointers is relatively straightforward, they are in fact the source of a wide number of sometimes very difficult to resolve problems in programming. This tutorial will demonstrate the basics of what pointers are and how to correctly use them, and how to avoid the common pitfalls. To this end this tutorial will look at:

* Memory addresses
* Pointers
* Dereferencing a pointer

### Memory Addresses

In the simplest terms, **computer memory** consists of a number of **bytes of data arranged sequentially**. Each of these bytes can be individually accessed by its position in the sequence, and this is called the **memory** **address** (usually just **address** for short). An address is a numerical value, such as 1024, which refers directly to the byte at that location in memory:

|  |  |
| --- | --- |
| **Address** | **Data (bytes)** |
| 0 | byte |
| 1 | byte |
| 2 | byte |
| 3 | byte |
| ... | ... |
| 1023 | byte |
| 1024 | byte |
| 1025 | byte |
| 1026 | byte |
| 1027 | byte |

The data contained in variables declared in a program needs to be stored somewhere in memory. The **address of a variable** is the **place in memory where the data is stored**. Each data type uses a different number of bytes of memory. The fundamental data types are - **integers** which use 4 bytes, **floats** use 4, **doubles** (high precision decimal numbers, used extensively in maths and science) use 8, and **char** in C++ uses 1.

When an int is declared in a program like so:

int x = 2;

the integer uses up 4 bytes of memory. Imagine, for simplicity, that this integer is stored at address **1024**. This diagram shows how the integer variable x is stored in memory:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1023 | ... | ... |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| 1028 | ... | ... |

The **address** of the variable x is the **first** byte of memory it is stored at, i.e. 1024, and because an integer uses 4 bytes it also occupies memory locations 1025, 1026 and 1027. These four bytes store the **value** of the variable, in this case **2**.

Let's assume every time a variable is declared, it is stored in memory **after** the preceding variable that was declared. This isn't always the case, but it'll do for this simple example. Now if a char is declared next:

char b = 'w';

since an integer takes up four bytes of memory, i.e. addresses 1024, 1025, 1026 and 1027, the char variable b will have the address **1028**:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1023 | ... | ... |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| 1028 | char b; | 'w' |
| 1029 | ... | ... |

and only uses that single memory address as a char is just one byte.

If a double is then declared:

double c = 3.141592;

then since a char take up only one byte, the double c will have the address **1029** and use the memory locations **1029 to 1036**. Likewise, if the next variable declared is:

int i = 3;

then it will have the address **1037**, since doubles take up eight bytes.

After all these variable declarations using the simplified memory allocation model, here's how memory would look:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1023 | ... | ... |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| 1028 | char b; | 'w' |
| 1029 | double c; | 3.141592 |
| 1030 |
| 1031 |
| 1032 |
| 1033 |
| 1034 |
| 1035 |
| 1036 |
| 1037 | int i; | 3 |
| 1038 |
| 1039 |
| 1040 |
| 1041 | ... | ... |

### Pointers

As you have seen, the values of normal variables are stored in memory at particular individual addresses, and each different type of data has a specific size. These variables are always in the same place in memory for as long as they exist. So, for instance in the example above, the integer variable called x will **always** refer to the integer data at memory address 1024.

It is often very useful when programming to be able to **create** data more dynamically than with normal variables, you'll explore why in the next tutorial. For now, you'll begin to investigate exactly how to go about **accessing** data in a more dynamic way. What you need is a type of variable that can **point at different memory addresses** at different times as needed, and these types of variable are called **pointers**.

So, given all that, what does this term pointer actually mean? While a normal variable stores the **actual data** for any given data type, e.g. in the above example the integer x contains **2**, a pointer variable **stores a memory address**. **Pointers are variables that contain memory addresses**. The memory address contained in a pointer can be used to look up the **contents of a normal variable at that memory address**. With reference to the example above, imagine a pointer variable,p\_int that has the value **1024**. This could be visualised as:

**POINTER**

**MEMORY**

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1023 | ... | ... |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| 1028 | ... | ... |

|  |
| --- |
| p\_int |
| 1024 |

So in this case the pointer p\_int is pointing at the integer variablexwhich is at the address 1024. Since p\_int points at x, p\_int could be used to access the value of x, assuming that there is a mechanism to do so - more on this later. Usually pointers are specific to a type of variable, e.g. int or float, so in this case p\_int would have been declared as a **pointer to an integer**.

Pointers take up space just like normal variables, but how many bytes a pointer takes up can vary depending on computer architecture and operating system. In a 32 bit Windows computer the size of a pointer is 4 bytes, so this will be used for the following example. Here's what memory would look like in this case if the p\_int integer pointer variable had been declared next and pointed at x:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| ... | ... | ... |
| 1037 | int i; | 3 |
| 1038 |
| 1039 |
| 1040 |
| 1041 | int \*p\_int; | 1024 |
| 1042 |
| 1043 |
| 1044 |
| ... | ... | ... |

The dashed line is to show that p\_int is pointing to x, and can be used to access that integer. **NOTE** int \*, or int\*, is the data type of an integer pointer. This is discussed in the next section, **Pointer Syntax**.

An extremely useful property of pointers is that they are able to point at **many different variables** during the execution of a program. If the value of the address that p\_int points to is changed from **1024** to **1037** then you would see that instead of pointing to the integer variable x, it is now pointing to the integer variable i. This is shown in the diagram below:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1024 | int x; | 2 |
| 1025 |
| 1026 |
| 1027 |
| ... | ... | ... |
| 1037 | int i; | 3 |
| 1038 |
| 1039 |
| 1040 |
| 1041 | int \*p\_int; | 1037 |
| 1042 |
| 1043 |
| 1044 |
| ... | ... | ... |

Now that p\_int is pointing to the integer i it can be used to access the integer value stored in i **instead of** x.

Pointers can also be set to a special value that means they are currently **not pointing at any particular variable**. This value is called NULL:

|  |  |  |
| --- | --- | --- |
| **Address** | **Variable** | **Value** |
| 1041 | int \*p\_int; | NULL |
| 1042 |
| 1043 |
| 1044 |
| ... | ... | ... |

As it is set to NULL**,** p\_int is now not pointing at anything. NULL is a very useful value that is of considerable help when manipulating pointers.

### Pointer Syntax

Now you have seen the basic idea of how memory, variables and pointers work you'll now go into the specifics and the correct syntax to use. An integer pointer is declared like this:

int \*p\_int;

Putting a '\*' after the type here means that a pointer that will **point to a variable of that type** is being declared, in this case an integer pointer called p\_int. It doesn't matter where the '\*' goes in this declaration, the compiler doesn't mind, so all the following are equivalent:

int\* p\_int;

int \* p\_int;

int \*p\_int;

int\*p\_int;

However, you should choose a **single** one of these and consistently use it to make your program more easily readable. You can of course name a pointer variable what you like, within the usual rules, but it is a good idea to use a name that makes it obvious what it is. In this case I've preceded the name with "p\_", this is a fairly standard way to indicate a variable is a pointer. The following pointer declarations are all valid:

int \* p\_counter; // another pointer to integer

float \* p\_distance; // pointer to a float

char \* p\_input\_character; // pointer to a char

int \* x; // valid, but not a good name

Immediately after declaration the pointer is **undefined**, and could be **any** value - in other words it doesn't point to anything meaningful. It is normally a good idea to set a pointer to NULL before it is used, this means it has formally been set as **not pointing to anything**. For instance, you could set any pointer you use to NULL when declaring it:

int \*p\_int = NULL;

The next thing to do therefore is to point p\_int at something. This can be done by:

int x;

p\_int = &x;

The '&' symbol in this context means **'memory** **address of**'. This can look a little confusing at first, but if you read it as "p\_int **equals the address of** x", or "**the value of** p\_int **is set to the address of** x" it might seem clearer. All it really means is that you are pointing the integer pointer p\_int at the address of the integer variable x by storing the address of that variable in the pointer, in exactly the same fashion as the diagrams above. Should you choose to point p\_int at i instead all you need to do is:

p\_int = &i;

i.e. "p\_int **equals the address of** i", and you'll find thatp\_int is now pointing at i. The same sort of syntax applies to other data types:

char mychar;

double mydouble;

char \* p\_charpointer; // declare char pointer **p\_charpointer**

double \* p\_doublepointer; // declare double pointer **p\_doublepointer**

p\_charpointer = &mychar; // **p\_charpointer** equals address of **mychar**

p\_doublepointer = &mydouble; // **p\_doublepointer** equals address of **mydouble**

You can also use the assignment statement to **copy** the value of pointers, i.e. copying memory addresses:

int x, i;

int \*p\_int = &x;

int \*p\_another\_int = p\_int;

This code assigns the value in p\_int to p\_another\_int. Because p\_int contains the address of x, p\_another\_int also will also now contain the address of x, i.e. **both pointers now point at** x. If you subsequently point p\_int at another variable, p\_another\_int still stays pointing at x, e.g.

p\_int = &i; // **p\_int** now points at **i**, but **p\_another\_int** still points at **x**

### Dereferencing Pointers

Once you have a pointer that is pointing at a variable you will want to **access the contents of that variable** using the pointer. Of course the pointer itself is a memory address, so to obtain the contents of what it is pointing at you would need to use the following syntax:

int y;

y = \*p\_int; // Copy contents of what p\_int is pointing at into y

Here a new integer variable y has been declared, then the contents of whatever p\_int is pointing at are copied into y. You can read the second line as "y **equals the contents of what** p\_int **is pointing at**". Notice the different use of the '\*' operator here, as compared to declaring the pointer. This is called **dereferencing a pointer variable**.

Dereferencing a pointer **gives access to the contents** of the variable that is pointed at. Here are a few more lines of code to help show what is happening:

int x = 2; // Declare x. Set x to 2

int \*p\_int; // Declare int pointer p\_int

p\_int = &x; // Point p\_int at x

int y; // Declare y

y = \*p\_int; // Copy what p\_int is pointing at into y, i.e. copy contents of x into y

// same as y = x;

Here x has been set to 2, then the pointer p\_int is pointed at x. Another integer y is declared, and is then set to copy the contents of what p\_int is pointing to. Therefore y will end up equal to the value of x, i.e. **2**.

You can also use the dereferenced pointer to set the value of the variable that its pointing to, so:

int x = 2;

int \*p\_int;

p\_int = &x;

cout << x; // Outputs 2

\*p\_int = 25; // Copy 25 into what p\_int points at

cout << x; // As p\_int is pointing at x, x is set to 25, so this now outputs 25

cout << \*p\_int; // Also outputs 25

shows that \*p\_int can be used in place of x in the assignment and output statements. Be careful to remember to include the dereferencing operator, if you don’t you are trying to directly set the address pointed at by the pointer, which is probably not what you want to do. The following statement misses out the ‘\*’ operator and will not compile:

p\_int = 25; // INCORRECT. Missing \* operator, will not compile

### Uses of the '\*' Operator

You may find at first that it's a bit confusing using the '\*' operator, as it has two distinctly different uses when dealing with pointers:

* On the one hand it's used to **declare** a pointer variable, e.g. int \*p\_int; declares a variable p\_int which is of type int\* (integer pointer)
* On the other hand, it's also used to **dereference** a pointer and obtain the value when placed before a pointer without specifying a type e.g. x = \*p\_int;

Of course the '\*' operator is also used for multiplication. Take the time to understand the differences between these uses, and when each is to be used. Practicing by writing lots of code that uses the operator with pointers, and stepping through with the debugger - this can really help your understanding.

## Exercises

**In order to aid understanding of how the exercises work, use the debugger to step through all of the code you write for the exercises.**

**You are advised to write notes on all aspects of the tutorial and exercises in your notebooks. This can then be used to help with your assignments.**

#### Exercise 01

This exercise will get you used to examining pointers using the debugger. Create a project as usual, and copy the following code into **main.cpp**:

#include <iostream>

using namespace std;

int main()

{

int x; // Declare x

int \*p\_int; // Declare int pointer p\_int

int y; // Declare y

p\_int = NULL; // Set p\_int to NULL

p\_int = &x; // Point p\_int at x

x = 2; // Give x a value of 2

y = \*p\_int; // Copy what p\_int is pointing at into y

cin.get();

}

1. Put a breakpoint **on** the line where p\_int is set to NULL and press **F8** (**F5** in VS) to build and run the app.
2. Examine the **Watches** window in the debugger. What values are all of the variables? What values are all of the pointers? Write them in your notebook. Explain why you think they are these values.
3. Press **F7** (**F11** in VS) to execute the current statement (p\_int = NULL;) and step to the next. How have p\_int and what it is pointing to changed? Discuss what you think this might mean in your notebook.
4. Press **F7** again (p\_int = &x;). How has p\_int changed? Discuss what you think this means in your notebook.
5. Press **F7** again (x = 2;). What happens to p\_int when this is done?
6. Press **F7** again (y = \*p\_int;). What value does y end up with? Explain why in your notebook.
7. Go to the **Watches** window, and add a 'watch' for p\_int (to do this click on an empty box under the **Locals** header and enter the name required).
8. Now add a watch for &p\_int. Examine the watch variable and explain what you think this might mean.
9. Add watches for &x and &y. Explain what these represent in your notebook.
10. Add a new integer z, and assign it with the value of **3** **before** the pointer assignments. Add a watch for &z as well. After y = \*p\_int, point p\_int at z. Execute up to and including this line of code. Does this affect x or y? Discuss why in your notebook.
11. Add a new integer pointer p\_int2. How would you assign p\_int to p\_int2? You need to modify it somehow before assignment. Do this **after** pointing p\_int at z.Explain what happens when you do this in your notebook.
12. Add a line of code that changes z, **after** the assignment above. What happens to the values pointed at by p\_int and p\_int2, and why?
13. Add code, **after** the line added in part 12, to set p\_int to point to x again. After, add the statement:

p\_int = \*p\_int2;

What does this do and why?

1. Next, add code that uses **dereference** operators to set what p\_int2 is pointing at to 11. What else do you think will change? Examine the Locals tab to see if you are correct.
2. Add code at the end to output the values of all of the variables in the program, **don't use any dereferencing or address operators yet**. What values are output for the pointer variables? Explain what they mean.
3. Now add code to output the **address** of x, y and z. Compare the result to the values output for the pointer variables. What do you notice? Can you tell which pointer is pointing at which variable?
4. Finally, add code that uses the **dereferencing** operator to output the values pointed at by the pointer variables. Again, compare these with the values of all the integers, and explain what you find.

#### Exercise 02

Write a program that:

1. Declares a float and a float pointer.
2. Reads in a value from the user and stores it in the float variable.
3. Sets the pointer to the address of the float variable.
4. Prints out the memory address contained in float pointer.
5. Prints out the value stored in the float variable by dereferencing the float pointer.

#### Exercise 03

Copy your work from Tutorial 7, Exercise 11 into a new cpp file. Convert the code to C++. Then add a new function called manInTheMiddle which accepts three float pointers, just as halfDouble did, and then send these to a modified halfDouble to do the calculations. You will need double-dereference operators on one of these functions but which one?

#### Exercise 04

Pointers are one of the most misunderstood elements of C/C++, but understanding them is crucial for more advanced programming. I strongly recommend that you read more on pointers from other sources to reinforce what you have learnt here, as well as doing online tutorials and exercises. Test your knowledge by writing small programs of your own that use pointers, you could use previous tutorials as inspiration.