

# Pointers in C/C++

## Contents

<b>1</b>	<b>Memory Addresses</b>	<b>2</b>
<b>2</b>	<b>Pointers and Indirection</b>	<b>3</b>
2.1	The & and * Operators . . . . .	4
2.2	A Comment on Types - Muy Importante! . . . . .	6
2.3	Pointers and const . . . . .	7
2.4	The sizeof Operator . . . . .	8
<b>3</b>	<b>Passing Pointers to Functions</b>	<b>9</b>
<b>4</b>	<b>A Deeper Look at Arrays</b>	<b>10</b>
<b>5</b>	<b>Pointer Arithmetic</b>	<b>12</b>
<b>6</b>	<b>C-style Strings</b>	<b>13</b>
<b>7</b>	<b>Function Pointers</b>	<b>14</b>



## 2 Pointers and Indirection

- Variable that stores the location of another variable are called *pointers*
- They are declared by preceding the variable name with a `*` in the declaration

```
int *numPtr;
```

- In this case, this tells C++ that `numPtr` will hold the memory location of an `int`
- If we wanted to create a pointer to a `double`, we would do something like

```
double *otherPtr;
```

- As the book mentions, the `*` in the variable name is *not* part of the type. . . as such, the `*` only applies to the variable whose name immediately comes after
- For example, the following

```
int *numPtr, num;
```

defines `numPtr` to be a pointer to an `int`, and `num` is defined as an `int` (*not* a pointer!)

- If you wanted to define two pointers to integers, you would have to do

```
int *numPtr1, *numPtr2;
```

- ***Make no mistake!*** In the above example. . .
  - `numPtr` simply stores some memory location
  - It's *value* is just a number indicating *where* something is stored

- As a good practice, pointer names contain the suffix “Ptr” in their name, as this makes it immediately obvious that the variable is a pointer

## 2.1 The & and \* Operators

- So now that we can create pointers? How do we set them to specific memory locations?
- How do we retrieve the memory locations of other variables?
- As we saw in the previous examples, we can use the & operator

- For any variable `v`, the & operator returns the numeric memory location of `v`
- E.g. given

```
int v;
```

`&v` returns the numeric version of the memory location for `v`

- *revisit old examples...*

- So, now that we have that...
- We can assign a pointer to *point to* another variable (that is, set the pointer to the memory location of the variable) by doing the following...

```
int v;

int *vPtr = &v;    // now "vPtr" contains the location of "v"
                  // (we say "vPtr points to v")
```

- Given two pointers, we can copy one into another through the usual assignment...

```
int v;

int *vPtr1, *vPtr2;

vPtr1 = &v;        // now vPtr points to v
vPtr2 = vPtr1;     // now vPtr2 also points to v
```

- But what can we *do* with a pointer?
- This is where the `*` operator (sometimes called the *dereferencing* operator) comes in...
- Given a pointer/memory-location, the `*` operator dereferences the pointer, and refers to the actual *value* in the memory location
- That is, `*vPtr` is the value in the memory location of `vPtr`
- *examples...*

- Note that we can also define pointers *to pointers*...

```
int v = 10;
int *vPtr = &v; // point vPtr to v

int **vPtr2 = &vPtr; // point vPtr2 to vPtr
```

- Note that the type of `vPtr2` is `int**`, since it's a *double pointer* (a pointer to a pointer)
- There is no limit to the level of indirection

## 2.2 A Comment on Types - Muy Importante!

READ THIS. RE-READ THIS. LEARN THIS.

- It's exceedingly important that you understand the following...

- Given the following two declarations...

```
int v = 10;          // set v to 10
int *vPtr = &v;     // point vPtr to v
```

- Do *NOT* mistake the types of the following expressions...

Expression	Type
v	int
vPtr	int*
&v	int*
*vPtr	int
&vPtr	int**

- Notice that the above expressions can be build upon indefinitely
- That is, *\*&\*&v* is a *valid expression!* What type is it?

- *MOST* pointer errors are due to a misunderstanding of something above

- *examples...*

## 2.3 Pointers and const

- There are two different ways to make pointers **const**

1. The first creates a pointer to *constant* data

```
const int *xPtr;
```

Think of this as follows. . . `xPtr` is a pointer to a constant integer (`const int`). We can tell `xPtr` to point anywhere, but whatever it points to, it *must* be a constant integer. Note that the value of `xPtr` can change (it can point to different things).

2. The second creates a constant pointer, which means it must always point to the same thing (i.e. its *value* is constant)

```
int * const xPtr = &x
```

Here, `xPtr` is fixed to point to `x`, and it can never point anywhere else. That does not mean, however, that the value of `x` is constant! Note that like a normal constant, one *must* give `xPtr` a value (assign it to something) in its declaration!

- *examples. . .*
- So, for pointers to constant data, the **const** qualifier comes **before** the **\***
- For pointers which are constant in what they point to, the **const** qualifier comes **after** the **\***

- The two methods of **const** pointers above can also be combined
- I.e. we can create a pointer which points to constant data, and can never point to anything else. . .

```
const int * const xPtr = &x;
```

- *examples. . .*

## 2.4 The sizeof Operator

- C++ provides an operator called `sizeof(x)`, which evaluates to the size of a data type, variable/constant, or an array. It measures the size in the number of bytes
- So you can use `sizeof` to determine how many bytes are used for specific data types, which may change from system to system
- *examples...*

### 3 Passing Pointers to Functions

- Reference parameters, which we covered earlier, are specific to C++
- They don't exist in C
- In C, if you wanted to pass something by reference, you would pass it *by pointer*
  
- As you've seen with pointers, manipulating them can be a little tricky at times
- C++ compilers really just translate these reference parameters to *pointer parameters*, and the compiler does all of the tricky manipulation for you
  
- Passing by pointers still have their place, but they're few and far between (and certainly not necessary for anything in this class)
  
- Like reference parameters, if you are passing something by pointer for the sole purpose of reducing overhead, *you should make the pointer parameter constant!*

## 4 A Deeper Look at Arrays

- As the book covers in §8.9, there is a close relationship between arrays and pointers
- An array such as

```
int foo[10];
```

really just allocates a chunk of memory to store 10 integers, and the array name “foo” is simply a pointer to the start of this piece of memory

- How much memory is allocated? Enough for 10 integers. How much for each integer? Simply `sizeof(int)` bytes!
- So, an array declaration like `type arrayName[arraySize];`, will allocate

```
sizeof(type) * arraySize
```

bytes, and set `arrayName` to *point* to the first element

- *examples...*

- **So an array name is just a pointer!**

- What about brackets? If the name is just a pointer, what do the brackets do?
- Brackets are an *operator* in C++, just like `+`, `-`, or even `sizeof()`
- Specifically, they translate as follows. Given something like

```
array[index]
```

this is translated as

```
*(array + index)
```

- What does this do?
- *examples...*

- So the brackets are just operators for doing simple pointer arithmetic!

## 5 Pointer Arithmetic

- *many many examples...*

## 6 C-style Strings

- §8.13 in the book covers pointer-based strings, which were the original way strings were stored and worked with (and still are in C)
  - Strings are stored essentially as arrays of `chars`, which as we saw earlier, are just pointers to chunks of memory
  - They have the property that a *proper* C-style string *must* end in a null character (`'\0'`)
  - To determine the length of such a string, you essentially iterate through the array until you find the null character, and keep an ongoing count of each character you pass
  - There are functions for all of this (`strlen` for calculating the length), which is pretty standard across systems
- 
- You should read this chapter and be aware of what it talks about, but this is the most we'll cover in class
  - If you work with any older software (or *any* software in C), you *will* see these
- 
- Any questions on the quiz on Friday or on the Final Exam regarding this section will be *very* high-level (read: easy), such as *How are C-style strings stored and manipulated?*

## 7 Function Pointers

- When a program containing functions is compiled, every function is compiled as well (obviously) and when the resulting binary is loaded into memory, each compiled function is loaded as well
- Each compiled function can be referenced in memory by the location in memory where its code begins
- As such, we can *point* to functions just like we can point to variables
  
- This can be *extremely* useful when you don't just want to pass values to functions, but also *other functions*
- *examples*
  
- You can also create *arrays* of function pointers, where each element in the array points to a different function (each of the same type)
- This can be extremely handy for menu driven applications, event handling, and a number of other things
  
- Covering function pointers completely would take *at least* an entire class, simply because the syntax for them can be quite difficult at first
- I encourage you to read §8.12 of the book for examples
- Also, a great reference for function pointers (using them in virtually *every* situation you could possibly think of), is...

<http://www.function-pointer.org/>