Addresses and Pointers

Matt Warner

Contents

1 Number Systems

The decimal number system we normally use for representing numbers is a ***positional number system*** in which any natural number may be uniquely represented by use of the ten symbols 0,1,2...,9.

In this system these then symbols, also referred to as decimal digits, represent the numbers zero, one, two,..., nine, respectively. A unique representation of any natural number m can be given in the form

$$d_n d_{n-1} d_{n-2} \dots d_1 d_0$$

where $m \geq 0$. The same natural number m can also be represented in the form

$$d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + d_{n-2} \cdot 10^{n-2} + \ldots + d_1 \cdot 10^1 + d_0 \cdot 1-^0$$

For example, the number one hundred twenty-three may be represented by

$$123$$

Or by

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

The decimal number system is also called the ***base ten system*** since ten digits are utilized in the number representations in this system.

There is, however, nothing sacred about the base ten since the notion of a positional number system can easily be generalized to any given base b where b is a natural number greater than or equal to two.

For example, we can also represent the number one hundred twenty-three in the ***base two*** or ***binary number system***. The symbols 0 and 1 are chosen to represent zero and one, just as then symbols were selected in the base ten system to represent zero, one, two, ... nine. Then, since

$$123 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

The number one hundred twenty-three would be represented in the binary number system by

$$1111011$$

So, any natural number that we can represent in base ten can also be represented in base two. It's also **much** easier to represent *two* distinct states in a phyiscal system like a computer than it is to represent *ten* distinct states. You can use the two stable states of a flip-flop, two positions of an electrical switch, two distinct voltage or current levels allowed by a circuit, two distinct levels of light intensity, two directions of magnetization or polarization, etc.

The fact that it's easy to represent binary numbers with hardware which has made base two the number system of choice in digital devices such as computers

2 Bits and Bytes

Computer memory (often called RAM - "random access memory") is measured using different units (like inches, feet, yards). Most commonly, we use

- **bit** (binary digit): Each bit can hold a 0 or a 1, nothing more. A 2 is too big.
- **byte**: Can hold 8 bits, or a decimal value from 0 to 255.
- **word**: Size depends on the system. Usually 2,4, or 8 bytes. 2 bytes can hold a value from 0 up to about 65,000. 4 bytes can hold a value up to about 4 billion. 8 bytes can hold a value up to about $1.8 \cdot 10^{19}$

We can represent many different types of data using just groups of bits:

- As outlined above, groups of bits can be used to represent the larger integer numbers we might want to store in a computer program.
- One bit can be used for the number's sign (0 = positive, 1 = negative), which allows us to represent negative values.
- Character data can be represented using a numeric code such as ASCII or Unicode that assigns a distinct integer representation to each character.
- Boolean values can also be easily represented as an integer (0 = false, 1 = true).
- Floating-point numbers can be represented (with some loss of precision) as two integer values packed together, one for the fraction and one for the exponent (similar to scientific notation).

The variables you define in a C++ program and the code for the functions you write all occupy space in the computer's memory when the program is run. That means they occupy some number of bytes. FOr a variable, the number of bytes occupied depends on the variable's data type and the system the program is compiled and run on. Here are the sizes of some different data types on our Unix system:

- *char* = 1 byte (0 to 255)
- *bool* = 1 byte
- *short int* = 2 bytes (-32,768 to 32,767)
- *unsigned short int* = 2 bytes (0 to 65,535)
- *int* = 4 bytes (-2,147,483,648 to 2,147,483,295)
- *unsigned int* = 4 bytes (0 to 4,294,967,295)
- *long int* = 8 bytes (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- *unsigned long int* = 8 bytes (0 to 18,446,744,073,709,551,615)
- *float* = 4 bytes (plus or minus 10^{38}, limited to $\approx$ 6 significant digits)
- *double* = 8 bytes (plus or minus 10^{308}, limited to $\approx$ 12 significant digits)
- *long double* = 16 bytes (plus or minus 10^{308}, limited to $\approx$ 31 significant digits)
- C string = depends on number of chars in the array
- object type = sum of the sizes of the individual data members

Note:-

These sizes may be different on a different computer. To find the size of a data type or a variable if you don't know it, use the `sizeof` operator:

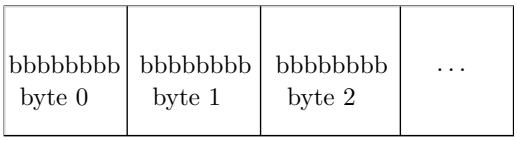
```
sizeof(int)     // An expression that evaluates to 4 on our Unix system.
sizeof(x)     // Evaluates to the amount of memory that x occupies
```

Also, sizeof looks a bit like a function, but it's not, really. It's and operator built-in to the C++ language.

The uncertainty of the size of various data types in C and C++ is a problem. It was never defined as part of the language and it's far too late to change now. More modern languages such as Java standardize the size (and therefore the range) of numeric data types so that there is no uncertainty.

3 Addresses

Bytes in the computer's memory are asigned consecutive increasing mumbers starting with the number 0, Thus, storage may be pictorially represented as



Where each of the b's represents a bit and the number assigned to a given byte is called the ***address*** of that byte. Addresses range from 0 to the maximum amount on the computer. Addresses are binary numbers, but are often printed in hexadecimal (base 16) to save space. You can print an address in decimal by type casting it to an integer.

The ***address of a variable*** is the address of its first byte of storage that it occupies. Similarly, the address of a function is the address of the first byte of storage that the function's code occupies.

We rarely have to know the actual address of a variable or function, but we do need to understand the idea of addresses and the fact that variables take up a certain amount of space in memory.

To obtain the address of a non-array variable, we can use the & operator.

This is not the same operator as the & when declaring a reference variable. It also has nothing to do with the && operator used in compound conditions. This is confusing, but you just have to keep in mind the context in which you're using the &.

Symbol	**Context**	**Means**	**Example**
&	In the declaration of a data type (variable declaration, function return data type, function parameter)	This data type is a reference type	`int& x = num;`
&	As a unary operator (variable or function name to the right of the operator, no whitespace), usually in an assignment statement or function call	"Address of" operator	`cout << &num;`
&	As a binary operator (variable or literal on both sides of the operator), usually in an assignment statement	Bitwise AND operator	`num = num & 5;`
&&	As a binary operator, usually in a decision or loop condition	Logical AND operator	`if (num >= 5 && num <= 10)`
&&	In the declaration of a data type (variable declaration, function return data type, function parameter)	This data type is an "r-value reference" (an advanced data type used in C++ "move semantics")	`string&& other`

We can use the & operator to obtain the address of a variable and print it (in either hexadecimal or decimal) in a program:

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{

  int num  = 5;

  cout << "Value of num is " << num << endl;
  cout << "Address of num (hexadecimal) is " << &num << endl;          // prints base 16
  cout << "Address of num (decimal) is " << (long int) &num << endl;  // Prints base 10

  return 0;
}
```

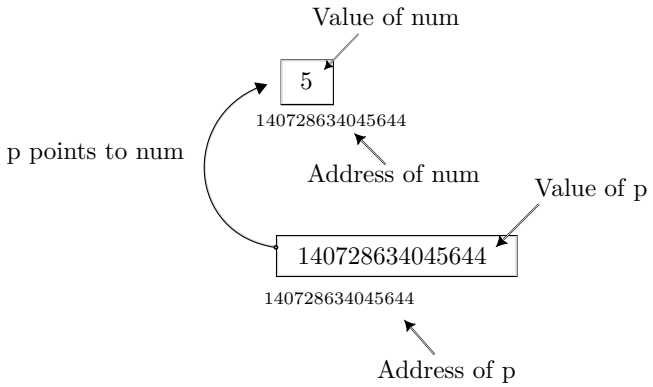
We can declare a pointer variable to any non-array data type. C++ considers all of these pointer types to be different data types; a pointer delcared as `int*` and a pointer delcared as `char*` are not the same data type.

```
float* floatPtr;    // floatPtr is a pointer to a float.

char* first;         // first is a pointer to a char.

Date* dataPtr;      // dataPtr is a pointer to a Date object.

double* x, * y;     // x and y are both pointers to double.
```

We can use the & operator to put the address of a variable into the appropriate type of pointer variable.

```
int num = 5;

int* p = &num;
```

Now we can say “p contains the address of num” or “p points to num”. The figure below illustrates the relationship we’ve established with these two lines of code.



Note that since p is itself a variable, it occupies some number of bytes in the computer's memory and has its own address (14072863404556 in this example)

What good does this do? Some of the most important uses will come later, but for right now, we can use this together with one more new idea to create a new way to access the value in a variable.

We know we can access the value in `num` by using `num` itself; for example:

```
1  cout << num << endl;
```

Now we can use the pointer variable to get to `num's` value (assuming as above that p points to `num`). This will prove very useful soon.

But first we have to know *how* we can access the value stored in `num` by using the pointer p? We ***dereference*** the pointer. "Dreference a pointer" means "access the value of the variable that the pointer points to."

The ***dereference operator*** is the *. Write the * before the pointer and you have an expression that refers to the "value pointed to" by the pointer.

So given the declarations and assignments above, we can code:

```
1  cout << num << endl;     // Prints 5.
2  cout << *p << endl;      // Also prints 5.
```

Note:-

In the first line of code, we print the value stored in the variable `num`

In the second line of code, we print the value stored in the *int* variable pointed to by p (which is the value in `num`)

They are the same thing

Notice another possible source of confusion:

In a declaration, you write:

```
1  int* p;
```

This declares p to be a variable that can hold the address of an integer variable. The data type of p is *int** (pointer to an integer).

In contrast, in an **executable statement** you might write:

```
1  x = *p;
```

Here, *p refers to "the value in the variable whose address is stored in the pointer variable p" or more briefly, "the value pointed to by p"

In all, there are three different contexts in which you might use the character * in C++.

Symbol	Context	Means	Example
*	In the declaration of a data type (variable declaration, function return data type, function parameter)	This data type is a pointer type	`int* p;`
*	As a unary operator, with a pointer variable name or pointer arithmetic expression to the right of the operator	Dereference operator	`cout << *p;`
*	As a binary operator	Multiplication operator	`num = num * 5;`

4 Pointers to Objects

We can create pointers to objects in the same fashion as pointers to built-in types like *int* and *double*. Pointers to objects have some additional syntax associated with them when it comes to accessing members of the object:

Expression	Meaning
`pointer-name`	Address of the object pointed to by `pointer-name`
`*pointer-name`	Value of the object pointed to by `pointer-name`
`(*pointer-name).member_name`	Syntax to access the data member `member_name` of the object pointed to by `pointer-name`
`pointer-name->member_name`	Alternative syntax to access the data member `member_name` of the object pointed to by `pointer-name`
`(*pointer-name).member_function_name(arguments)`	Syntax to call the member function `member_function_name()` of the object pointed to by `pointer-name`
`pointer-name->member_function_name(arguments)`	Alternative syntax to call the member function `member_function_name()` of the object pointed to by `pointer-name`