Chapter 3: Variables

Notes

- Casting does not modify original variable.
- Difference between static and reinterpret cast.
- User-defined constants are known as “symbolic” constants.
- enum

Introduction

In the last chapter I talked about numbers and gave a very vague explanation about where and how they are stored. In this chapter you’ll learn how to set aside storage for these numbers and even change them with mathematical formulas. And I’ll tickle your interest with some information on what we’ve been working with all along but didn’t have a name for.

What is a Variable

I’m pretty sure you’ve heard this term thrown around; maybe in math class ten years ago! Variable means something that can change. In C++ a *variable* is a thing that can hold a value, but whose value can be changed. There are five attributes of a variable: name, value, type, address, and lifetime.

A variable’s *identifier*, or name, is how you refer to its *value* in your C++ program. It is the same with people. We identify each other by our names. The value of a variable is comparable to the person of a name. The value to us is ourselves, but the value of a C++ variable is *always numeric*. I don’t mean sometimes, I mean always. This is because, as I’ve explained, a value in a computer system equates to a set of bits stored somewhere.

A variable’s type determines the storage unit that the variable uses, or in other words, the number of bits it represents and how they are used. This is also known as a data type. The usage I speak of is what the sequence of bits means to a particular type. In most types the bits represent either an integer or float number; both of which I discussed in the last chapter. Until much later we will not deal with types whose bits are used otherwise.

The address of a variable is its location in memory. Memory in C++ is a giant linear sequence of bytes and a variable will occupy one or more of these bytes. The location of the bytes in this huge “list”, an integer value, is the address of a variable. An address, unlike a name, is retained when a C++ program is compiled into machine language. This is only a brief explanation of something very *huge* in C++ that people have difficulty mastering.

Lastly the lifetime of a variable is how long it exists. A variable cannot be used until it is created or after it is destroyed. The time from the creation *to* the destruction is the variable's lifetime. Only in its lifetime can a variable be used.

Data Types

The type of a variable is also known as a data type and it involves the two things I went over briefly: storage unit and bit usage. In all simple C++ data types the bit usage is limited to either integer or floating-point, which is good for you because it makes things easier. Below is a list of the basic C++ data types:

Data Type	Format	Storage Unit
bool*	integer	
char	integer	byte
wchar_t*	integer	word
short	integer	word
int	integer	varies
long	integer	double-word
longlong*	integer	quad-word
float	floating-point	double-word
double	floating-point	quad-word
long double*	floating-point	

* Still not widely supported.

The '`int`' type has a varying storage unit depending on the platform the C++ program is compiled on. If you're compiling for a 32-bit CPU, this type will have a size of 32 bits, i.e. a double-word storage unit. Most 32-bit CPU's have 16-bit capability and must be configured by the operating system to run in 32-bit mode. Thus, operating systems that do not do this, like Windows 3.1 or DOS[1], will use 16-bit integers. The '`int`' is typically the fastest type because it is usually identical to the CPU's registers which are variables that are stored in the CPU itself rather than the computer's memory. The 64-bit CPU is on the horizon as of writing this book, and the storage unit of an '`int`' on these systems is a quad-word.

Declaring Variables

Before using a variable in your program you must create it and begin its lifetime. To create a variable in your program you must write a *variable declaration statement*. This is a relatively simple statement that begins with the type you want to create, followed by

[1] DOS *can* be configured to enter protected mode, i.e. use the CPU's 32-bit capabilities, using a DPMI server. Compilers that build programs specifically for DOS protected-mode, like DJGPP, will thusly use 32-bit integers.

the name you want to use, followed optionally by an initializer expression, and ended (like most statements) with a semi-colon:

```
type name [ = initializer ] ;
```

The program below creates an integer, '`int`', named '`x`':

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x;
06      return 0;
07  }
```

We could initialize '`x`' with a value of five by modifying the program as so:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x = 5;
06      return 0;
07  }
```

This is known as *initializing* a variable. You are giving it a value to start with. The value you initialize a variable with is known as an *initializer*. In the above program '5' is the initializer. If you do not initialize a variable, it will usually be filled with *garbage data*. When you declare a variable you set aside memory for it, but that memory is not normally reset or cleared for you. So whatever was in memory before is what your variable starts with unless you initialize it.

After a variable has been created, it is "alive" so to speak and can be used. In the context of the programs in this chapter a variable will live until the closing brace, '`}`', is reached. At that point the variable is destroyed and can no longer be used. Because all of our code at this point will exist between these curly braces, this does not hinder us. The next chapter will expand on variable lifetime.

Signed and Unsigned

Most of the integer-based types are *signed* by default, but on some compilers (and/or when certain options are set) they are *unsigned*. Quite simply, a signed integer can be negative and has a lower numeric range than an unsigned integer. The maximum value of an unsigned variable is always two times plus one the maximum value of a signed variable of the same type.

If you wish to specify a variable as either signed or unsigned, you may put that before the type. For example, if we wished to specify that '`x`' is signed, the program would look like so:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      signed int x;
06      return 0;
07  }
```

A signed variable may be negative while an unsigned variable may not. Only integer variables may be one of these. Floating-point variables always support negative values. The *signedness* (as it is called) of a variable will effect its range as well, because one of the bits must be used to represent the sign of the variable.

Compound Declaration

You can declare multiple variables of the same time in a single *compound* statement. To do this you begin as you normally would for the first variable, but rather than end with a semi-colon, use a comma. For the remaining variables you only have to type their name and optionally their initial value; and you must separate each with a comma. The last variable declaration must be terminated by a semi-colon. The following would declare three variables, all of type '`int`':

```
int x, y, z;
```

We could initialize each of these variables as well:

```
int x = 1, y = 2, z = 3;
```

The names of variables are known as *identifiers* and follow a strict set of rules.

Identifiers

An identifier is a name, plain and simply. It is how you *identify* something. Later on you will see how to create more generic names liken to things: plant, animal, etc. But for now we will deal with specific names liken to the names we give each other "Bob Brown", "John Smith", etc. A variable's name is specific because it applies to that one variable, just as our names apply to us individually.

In a C++ program all *user-defined* things are identifiers. This would be all named things save for C++ *keywords* which are words reserved by the C++ language. Operators are

not C++ identifiers, they are a separate thing entirely and their meaning always remains the same.[2]

Identifiers are single words that can be anything except one of the C++ keywords. The words can include any alphanumeric character plus underscores. However, they can*not* begin with a number. The following are all *invalid* identifiers:

```
hello world
10things
one-twenty
```

The first, '`hello world`', is not a single word because there is a space between the two. The second, '`10things`', begins with a number (`1`). And lastly, '`one-twenty`', contains a dash. The following are all valid incarnations of the same names:

```
hello_world
_10things
one2twenty
```

Identifiers are case sensitive in C++. This means that the difference between two variables might be the casing of a single character. All of the following are seen as different identifiers:

```
Neil_Obremski
neil_obremski
Neil_obremski
NEIL_OBREMSKI
```

Lower-case characters, thus, are different than their uppercase counterparts.

Another thing to look-out for is *name-clashing*. This occurs when you try to create an identifier that already exists. One form of name-clashing, as mentioned, is trying to use an identifier that is the same as one of C++'s keywords. Another form is creating two identifiers yourself, like two variables, both with the same name. It *is* possible to create two identifiers with the same name as long as they exist in different spaces. This will be explored more next chapter.

Identifiers carry part of the attributes of a variable. A variable is simply a named value, a quantity that can be gotten or changed by referring to its *identifier*. The identifier is a name for the storage used by the variable. When a variable is destroyed the memory where it once lay still exists, but the identifier no longer does. Thus lifetime applies to a variable, but more explicitly it applies to the variable's name or identifier.

C++ Keywords

asm	auto	break	case	catch	char	class

[2] Though operators *can* be customized to do things other than they were intended!

const	const_cast	continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export	extern	false
float	for	friend	goto	if	inline	int
long	mutable	namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short	signed	sizeof
static	static_cast	struct	switch	template	this	throw
true	try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while	

Above is a list of all the current keywords currently in the C++ language. Depending on the date of your compiler you may not even be able to use all of these. New ones were added back in 1996 as part of the new ANSI Standard C++. Luckily, things don't change too much and all the really useful, and *basic*, ones are in all decent C++ compilers from 1990 to the present.

A keyword is a word that is reserved by the C++ language for a specific, built-in purpose. Believe it or not, keywords are the only things that are built into C++. Everything else you do requires including new modules in your program. You've already done this for 'cout' which is *not* a keyword. To use 'cout' you must have the line '#include <iostream.h>'. Keywords, on the other hand, need nothing like this to work … they are *part of the language itself*. All other things I teach you are part of the *C++ standard libraries*.

An identifier can*not* be one of these words.

Literals

This is a term for data in a C++ program that is not named. In other words it is a value without a name. However, just like a variable, a literal has a type, address, and value. A literal has no name because you use a literal by its value.

You've been using literals all a long, they are just difficult to see until you can compare them to variables. In the 'Hello World' program '`"Hello World"`' was a literal. In the program above, where we assigned '`x`' the value of five, '`5`' was a literal. And in the last chapter, you dealt exclusively with constant values for '`cout`'. They have no names because what you see is what they are. And unlike variables they are *constant*, meaning their value never changes.

Constants

Things that do not change are *constant*. In C++ you can have these which have all the same properties as variables but cannot have their values changed. A constant is created in the same way a variable is, except for one difference. The declaration begins with the

'const' keyword. If we wanted to make 'x' a constant rather than a variable, the program would look like so:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      const int x = 5;
06      return 0;
07  }
```

Using constants is a thing of etiquette rather than raw function, so it will not be covered in detail. A constant must be assigned a value when it is declared or you will get a compiler error. This is because although constants never change value, they must have a value to begin with. Notice that you can use a constant in the same ways you can use a variable and I encourage you to experiment with them.

Assignment

Let's not forget our operators! The most common operations you'll use with any variable are *assignment* and basic arithmetic: addition, subtraction, division, and multiplication.

An assignment operation refers to setting a variable's value. The assignment operator is the equal sign: '='. In a program above we created a variable called 'x' and *assigned* it the value five. Technically speaking the assignment operation places the value of the right operand into the variable named by the left operand. The right operand can be a literal, constant, or variable but the left operand *must* be a variable. The result of an assignment operation is the right operand.

The following program shows how you can use literals, constants, or other variables in an assignment operation:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x;
06      int y;
07      const int z = 7;
08      x = 5;
09      y = x;
10      x = z;
11      return 0;
12  }
```

Let's look at some of these statements individually. Here is the first assignment:

```
08      x = 5;
```

The literal value '5' is placed into the variable 'x'.

```
09      y = x;
```

Here the value of the variable 'x', which is '5', is placed into the variable 'y'.

```
10      x = z;
```

Lastly the value of the constant 'z', which is '7', is placed into 'x'. It is important to remember the obvious fact that *a variable cannot be used until it is declared*. Thus the following program would not compile:

It's like trying to drive a car when you don't own one. Really, there is no complexity; it just *doesn't* exist until it is declared. How can you rightly say "Excuse me" without producing the appropriate bodily output? ☺

Basic Arithmetic

The result of an arithmetic operator is the arithmetic operation it defines, obviously. The operands of these operations can be literal, constant, or variable. If you wish to place the result somewhere it must be *assigned* to a variable. The following demonstrates an addition of two values which is placed into the variable 'x':

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x;
06      x = 2 + 2;
07      return 0;
08  }
```

In the above program, the addition operation is performed and the result is assigned to the variable 'x'. Here is the operation break-down:

```
x = 2 + 2;
x = 4;
```

This is a common thing. Most arithmetic operations will have their result stored in a variable. Remember that a variable must be on the *left* side of the assignment operation for it to compile properly. This is a bit different than basic math where you would see:

$$2 + 2 = x$$

The addition operator is the plus sign: '+'. The others are the dash, '-', for subtraction, the asterisk, '*', for multiplication, and the forward slash, '/', for division. It is known as

the forward slash because the top end leans to the right which is generally acknowledged as moving forward (remember how all those old scrolling games favored going right).

Remember that the result of an operation can become the operand of another. Such is common the case with arithmetic. Many times you will be writing complex expressions with arithmetic operators.

Increment and Decrement

Since the result of most mathematical operations end up in a variable, there is a set of operators that combine the assignment and arithmetic into a single operation. These operators are the arithmetic operator followed by the plus '+' sign. All of these operators perform the specified arithmetic operation and the result is stored in the left operand. Explaining each of these is useless since I've already explained arithmetic operations. So, here's an example:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x = 8;
06      x += 3;
07      x -= 2;
08      x /= 3;
09      x *= 2;
10      return 0;
11  }
```

These are still known as assignment operators, they just perform a mathematic operation before assigning a value to the left operand. Their individual names are their mathematic operation followed by the word 'assignment'. Therefore '+=' is the *addition assignment* operator, '/=' is the division assignment operator, and so on and so forth.

These are not just faster to write, but they are handled quicker when your program is running. For example, the following would add three (3) to the current value of 'x':

```
x = x + 3;
```

That requires two operations:

```
x = x + 3;
x = sum;
```

Whereas if we had used the addition assignment operator it would simply have *incremented* the current value of 'x' by three (3). There would be no need for two operations or an intermediate value.

When you add to a variable's value it is known as *incrementing* it; likewise to *decrement* a variable is to subtract from its current value. Usually you will increment or decrement by one. If this is the case you can use the special increment/decrement operators. The increment operator is two plus signs: '`++`'. Of course the decrement operator, then, is two dashes: '`--`'. These are unary operators and there are two ways to use each that result in two different operations. To explain this we'll deal with just one, the increment operator.

If the increment operator is used with a left operand the result is the left operand *before* the increment. If the increment operator is used with a right operand the result is the right operand *after* the increment. Here's an example:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x = 6;
06      cout << x++ << endl;
07      cout << ++x << endl;
08      return 0;
09  }
```

The output of this program is:

```
 6
 8
```

The output of the first increment operation is the left operand before the increment. The output of the second increment operation is the right operand *after* the increment. Since the first increment has already increased '`x`' by '`1`', it is '`7`' in the second operation, and therefore ends up being '`8`'.

Casting

All operations expect specific data types. Usually the operands in an operation must be of the same type, as is the case with arithmetic. However, assigning a float value to an integer variable or vice versa is perfectly acceptable. In this case the compiler will automatically *cast* the operands to a specific type. To cast is basically to convert[3].

In the assignment operation and the basic arithmetic operations I have described, the compiler will automatically covert the right operand to the type of the left if they are not the same. This is generally the case with most automatic conversions. Take the following program:

```
01  #include <iostream.h>
02
```

[3] Some casting cases do not require a conversion because the types are already like enough.

```
03  int main()
04  {
05      float x;
06      int y;
07      x = 5;
08      y = 6.2;
09      return 0;
10  }
```

The float variable '`x`' is assigned an integer value and the integer variable '`y`' is assigned a float value. The values assigned to both are automatically *cast* to their respective types before being used.

You can explicitly cast a variable to specific type yourself. C++ provides several methods for doing this. One of them is the traditional C-style cast, a legacy of the C programming language from which C++ was born. The other methods are using one of the C++ casting keywords known as *casting operators*:

```
static_cast
reinterpret_cast
dynamic_cast
const_cast
```

To use any of these, you use the syntax:

```
cast<type>(expression)
```

The '*`cast`*' would be the casting keyword you wish to use. Following this is '*`type`*' within the greater than/less than signs which is the data type you wish to cast or covert to. Lastly the '*`expression`*' would be the thing you want to convert. The expression can be any valid expression that has a result. Preferably you will use a single identifier or literal that you want to cast or convert.

Now let's return to the previous program in which the values assigned to '`x`' and '`y`' were automatically cast. Let's cast them explicitly:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      float x;
06      int y;
07      x = static_cast<float>(5);
08      y = static_cast<int>(6.2);
09      return 0;
10  }
```

The difference between using '`static_cast`' and '`reinterpret_cast`' is, I'm afraid, fairly subtle. The rule is that if it can be cast implicitly then '`static_cast`' is

the one to use. Otherwise you'll need to use '`reinterpret_cast`' which is a "forced cast".

The other two casting operators: '`dynamic_cast`' and '`const_cast`', will not be covered until later. The first, '`dynamic_cast`' is used in conjunction with switching between advanced data types. The latter, '`const_cast`' is for toggling the 'const' attribute of a variable.

You may have been wondering about the operator side of these keywords. They are truly considered operators, but they certainly don't look like any you've seen before. I consider these special *language constructs*. I use that term in places where the programming language, C++ in this case, something which bends the normal rules of things. These are truly operators, though, and should be considered as such.

Author's Preference: I actually prefer C-style casting because it is a no-nonsense approach. I have yet had the experience where the different C++ casting types were really useful.

Enumerations

The 'enum' keyword allows you to create a list of named numeric values. The term enumeration loosely translates to "list" so this kind of makes sense. The enumeration itself can have a name as well. Using this name you can create variables based on the enumeration; that is, variables whose values are expected to be listed within the enumeration.

To declare an enumeration, write the keyword 'enum' followed by an identifier of your choosing. Next the list itself will be enclosed in curly braces and ended with a semi-colon. The members of the list are identifiers separated by commas:

```
enum <name>
{
    <member1>,
    <member2>
};
```

Each member you list in the enumeration will automatically be assigned some integer value. In fact they will usually be 'int' type constants. The members you list cannot be modified to represent a different value later on, thus they are constants. You can decide what value an enumeration member represents by assigning it a value within the list:

```
enum <name>
{
    <member> = <integer value>,
    <member>
};
```

You can create a variable based on an enumeration, simply by using the name of the enumeration as the type for the variable. A variable that is based on an enumeration will usually be the same as an 'int', but conceptually should only ever contain a value relating to a member in the list.

No two members in an enumeration may contain the same value. Typically the first member, if not assigned a value, will begin at one (1). However, if you assign the second member to one (1), but don't do anything with the first, the system will figure out some other number to assign the first member. Members following one that has been assigned a value will typically use that value incrementally. That is, if you assign a member the value three (3), the next member after that will be four (4), and so on. You can choose to assign values selectively and the gaps will be filled in for you.

The most common example I see for practicing your first enumeration is a list of the months:

```
01  #include <iostream.h>
02
03  enum Month
04  {
05      January = 1,
06      February,
07      March,
08      April,
09      May,
10      June,
11      July,
12      August,
13      September,
14      October,
15      November,
16      December
17  };
18
19  int main()
20  {
21      Month m;
22      m = January;
23      return 0;
24  }
```

I named the enumeration 'Month' because it represents what each member *is*. For example 'January' is not a *months* (plural), it's a *month*. By assigning 'January' the value of one (1), the next member 'February' should contain two (2), and so on. The last month, 'December', will represent the value twelve (12). Lastly, I created the variable 'm' based on the 'Month' enumeration and I assigned it the value of the member 'January' or one (1).

Output

You can output the value of a variable or constant with '`cout`' by specifying its name with the insertion operator. Remember the insertion operator is '<<' and that operation. Up until now we've done all sorts of things to variables and their values, but we can't see the results when running the examples.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x = 5;
06      cout << "x = " << x << endl;
07      return 0;
08  }
```

The program above places the value '`5`' into '`x`' and then outputs the value of '`x`'. The output for this program is:

```
x = 5
```

You can even combine operations like we did in the last chapter with numbers. Say you want to assign the result of a complex expression to 'x' and output the result. This could be combined into a single line:

```
cout << "x = " << x = (x + 3) / 4 * 6 << endl;
```

Math operators, including the assignment operator, have a higher precedence than the insertion operator ('<<') so they area all done first. Breaking down this program, the operations would be done as follows (assuming '`x`' is '`5`'):

```
cout << "x = " << x = (x + 3) / 4 * 6 << endl;
cout << "x = " << x = 8 / 4 * 6 << endl;
cout << "x = " << x = 2 * 6 << endl;
cout << "x = " << x = 12 << endl;
cout << "x = " << x << endl;
cout << x << endl;
cout << endl;
```

Input

Up until now our programs have had no user intervention. They simply run. And because there is no user or no randomization, they always incur the same results. Now is the time to put that to an end. I present to you 'cin' and the *extraction* (>>) operator:

```
01  #include <iostream.h>
02
03  int main()
04  {
```

```
05      int x;
06      cout << "Enter a number and press enter: ";
07      cin >> x;
08      cout << "You entered " << x << endl;
09      return 0;
10  }
```

If you compile and run this program you will be asked to enter a number and press enter. Once you have done this it will tell you what you entered and then end. Let's look at lines 6 and 7:

```
06      cout << "Enter a number and press enter: ";
```

First off I did not include an '`endl`' at the end of the first '`cout`' statement because I wanted the input cursor to be on the same line as that text. Had I put an '`endl`' at the end, the number would've appeared *below* the text as it was typed in.

```
07      cin >> x;
```

Here's the real gem. This extracts the characters that the user types in and puts them into the variable 'x'. Extraction seems like an odd term for this, but deal with it because I didn't make it up so I can't change it. ☺

That's about all I want to get into with input at this point. You can now use this to get user-entered data; currently only numeric. You'll get to the input of words and such later on, so chill.

Land Plot Analogy

The best way I have thought of to explain variables is by plots of land. Here's how it goes. Think of a long, possibly endless, street with no turns or stop signs or whatever. On one side of the street is nothingness and we won't concern ourselves with it. On the other side of the street are perfect square plots of land. Each of these can have a building, but in order to be used the used plots of land must be marked off and reserved using a name.

In this, each plot of land is a byte in memory. A used portion of land is a used portion of memory and equates to a variable. The name of the variable is the name of the reservation for one or more plots of land; just as a variable can utilize one or more bytes of memory. The building on the land represents the value of a variable. This can be practically anything, but its size is constrained to the accumulative size of the plots reserved. Such is the case with variables. You can store many different values, always numeric, but their range is limited to the amount of memory they occupy.

<analogy drawing>

Size Of

One useful keyword you'll want to know how to use, especially with variables, is '`sizeof`' which is actually a unary operator. The result of its operation is the size, in bytes, of the right operand. The right operand can be a variable, literal, constant, or even a variable type! To use the '`sizeof`' operator, just follow it with one of the fore mentioned things. Typically the thing is enclosed in parenthesis, but this is not required (though highly suggested). A good reason for using this is seeing how large '`int`' variables are on your platform:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x;
06      cout << "sizeof(x) is " << sizeof(x) << endl;
07      cout << "sizeof(int) is " << sizeof(int) << endl;
08      cout << "sizeof(5) is " << sizeof(5) << endl;
09      return 0;
10  }
```

The format of the '`sizeof`' result is an integer of type '`size_t`' which is actually just an '`unsigned int`'. Later on I'll cover this as it pertains to the keyword '`typedef`'. Please realize that the '`sizeof`' operator is a special language construct because it can accept a variable type. No normal expressions of any kind can use a variable type as a valid operand.

Summary

You may not have realize, but this chapter delved into three of the four key things that the stuff of programs are made of: storage (variables), output (printing the variables and other stuff), and input (getting a user to do a machine's job ☺). Logic is, of course, the essence of programming and is the instructions that tell the computer what to do. But actually utilizing logic and bending a computer to some purpose is the real task. You will learn important aspects to logic in the next chapter.

In *this* chapter I explained variables in fair depth, showed you how to create them using different types, how to change their values through assignment or basic arithmetic, how to print them to the screen, and how to have a user enter their value. I also explained identifiers and introduced you to casting which we will come back to later on.

You should be able to create variables of your own, give and change their values, output them to the screen, and input their value. You should recognize C++ keywords and casting, even if you don't yet know how to use them. Lastly the concept of memory and the data stored there should be getting a bit clearer. You will need this knowledge in some heavy chapters to come.