Chapter 1: Basic Rules

Requirements

- General understanding of topics in Part 1.

Objectives

- Recognize the basic syntax of a C++ program.
- Ability to write simple C++ programs that output text, numbers, mathematical calculations, and new lines.

Breaking into the Circle

The first steps are always the most difficult and learning a programming language is no different. I call the initial step "breaking into the circle" because of the circular construction of these concepts. Every thing seems to be based on other things, which in turn are based on others, and this cycle typically leads back to the beginning. Although it's true that the introductory circle is much smaller than the language itself, it still is a hurdle to be overcome.

Learning C++ for the first time is like trying to decide where to put the door on a fully built house. You can't get to the second floor without going through the first, but you also have a variety of rooms there to start with. Likewise, you'll find a lot of things in each room that you won't ever use.

This chapter and those following will contain a *lot* of information that isn't necessary for proceeding into advanced concepts, just like going through some rooms isn't necessary to going up another floor. These concepts are not superfluous so much as they are a completion of a particular topic.

They are covered where they are, and not in an appendix for example, because they fit in with the surrounding information. You can choose to digest all of this and become a C++ guru, or skip it to learn the concepts well enough to move on. You can always decide to reread the information you left behind.

Unnecessary information can be identified by what objectives I set out at the beginning of the chapter. If concepts are not under this umbrella of goals, you may skip them. Make sure at the end of the chapter that you can check off each objective confidently before moving on. Usually these objectives are fairly simple and you should get them even if parts of a chapter confused you. Remember, not all of these concepts are necessary to move on, but they are included to provide completeness.

The 'Hello World' Program

In writing a book on programming it's expected that the first program explained will be the legendary 'Hello World' program. This program simply has to output the text 'Hello World' to the screen and exit. The C++ source code to do this is below[1]:

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

In all of the source code I will present to you, a column of numbers will be to its left. These numbers represent the lines on which instructions occur. It will make it easier for me to show you which lines I am describing. But these are simply a visual benefit, do not type them in as part of your source code or it will not compile. This source should simply be typed as:

```
#include <iostream.h>

int main()
{
   cout << "Hello World" << endl;
   return 0;
}
```

Type the source code listed above and save it as 'hello.cpp'. Next, build it into a program using your C++ software and run it. You should see the output:

```
Hello World
```

All of the lines, as you've no doubt noticed, are executed in precise order starting from the top-most. The order in which instructions are executed in C++ always follows this top to bottom, left to right, pattern.

On many systems, DOS/Windows for example, there will be an additional blank line after the text 'Hello World':

```
>hello.exe
Hello World
```

[1] You may also see programs using '`void main()`' or just '`main()`'. These are valid as well and will be explored in Chapter 6.

```
>_
```

On other systems there will be none. This is just how these systems deal with programs that output console text. Some will automatically add an extra new line in case the program doesn't, while others may not.

But I digress; first let me break down this program line by line. Afterwards I will explain, in much more detail, the rules for governing this and all C++ programs.

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

This is a *preprocessor statement* because it begins with a pound sign (#); it *includes* the *header file* '`iostream.h`' which contains special C++ source code used for *linking* code based on a common header. The source code in the header is a map of sorts that allows the program to access some basic input and output functionality. In this case it allows the program to output text, specifically 'Hello World', using 'cout'. Chapter 9 has more information on preprocessing and headers.

```
02
```

Blank lines, i.e. *white space*, such as this are skipped. Unlike some other languages, C++ does not rely on having instructions organized based on line, except with preprocessor statements. *Any* white space at all is treated in the same way.

```
03  int main()
```

Here a *function* called '`main()`' is defined which all stand-alone[2] C++ programs must contain this to work correctly. A function is a group of C++ instructions that perform some sort of task. Our task in this case was to output the text 'Hello World'. The 'main()' function has special meaning among C++ programs because it is the *program entry point*, where the program begins execution Chapter 6 is all about functions.

```
04  {
```

The opening curly brace signifies the beginning of the function we just defined called '`main()`'. All of the instructions we want to execute when the program starts will be inside this pair of curly braces. A matched pair of curly braces is known as a *block*, specifically in this case a *function block*; there is more information on this in Chapter 3.

```
05      cout << "Hello World" << endl;
```

This line contains instructions to output the text 'Hello World' to the console you have opened. The output of characters to the command console is also known as *printing* as in

[2] If you are linking to certain libraries that provide this function for your program, you will be required to create some specialized entry-point function; such as WinMain() for Microsoft® Windows™ applications.

printing to the screen and not a printer. The '`cout`' piece is an *object* of a class, also known as a *class instance*, and the less-than sign pairs ('<<') are *operators* that have been defined as part of '`cout`'. Chapter 10 is all about classes, Chapter 13 is about defining operators to work with classes, and Chapter 11 is all about input/output using this and related methods.

```
06      return 0;
```

This sets the exit code of our program to zero and ends our program. The exit code of zero indicates a success in that the program completed without problems. It should always be the last thing before the closing curly brace:

```
07  }
```

Here the function known as '`main()`' ends as well as our program. After all C++ instructions up to this curly brace have been processed, the program will end and control will return back to you at the command console.

Strict C++

The most recent standard to date was published in 1998 and it defines a slightly different way of setting up a C++ program. This "new" way is considered *strict*, *modern*, or *pure* C++, but all in all it acts pretty much the same. If your C++ compiler is in a strict mode then the above program source will not compile correctly for a couple of reasons. You'll have to modify all of my examples slightly so they *will* work. You may also make these changes only if you're interested; most modern (post 1998) compilers support the new standard.

Some header files have been renamed in the published standard. When I say renamed, I mean the old style names are no longer even mentioned in the standard. Most compilers, however, still include the old style names as well and older compilers have them because of their existence before the standard.

The first change you'll want to make, if you're using a strict C++ compiler, is changing the old style header names to the new style. The following conversion table shows all of the headers that will be used in this part of the book and their associated new style.

Old Style	New Style
iostream.h	iostream
fstream.h	fstream
iomanip.h	iomanip
string.h	cstring

Besides naming conventions, the old style headers also dump all the functionality they provide into the *global* namespace. The new style headers, on the other hand, give the

same functionality as the old headers but they place it into a namespace called ‘std’. A namespace is a simple container for all named functionality, such as ‘cout’. Namespaces are covered in detail in Chapter 3.

Therefore to use the new headers easily you have to *use* the namespace known as ‘std’. There are two ways of doing this: explicitly specifying the namespace for each thing it contains or by implying that each thing you use is in that namespace. The latter is easier and is done by writing the following line after the inclusion of all headers:

using namespace std;

Basically the compiler will look inside the namespace anytime you use some functionality, such as ‘cout’. Thus, the ‘hello.cpp’ source should look like so after your changes:

```
01  #include <iostream>
02  using namespace std;
03  int main()
04  {
05      cout << “Hello World” << endl;
06      return 0;
07  }
```

Strict C++ will not be covered at all in later chapters because you make these simple changes yourself. I decided to use the old style headers because it’s easier to upgrade your thinking than downgrade and most compilers have both types of headers.

Lexical Conventions

The exact characters used in a language and how they can be combined to produce meaning, is known as *lexical conventions*. Rules imposed by these conventions are used to separate groups of characters into individual *lexical tokens*. However, the proper arrangement of lexical tokens, to produce accurate meaning, is governed by *syntax* rather than lexical conventions.

In written English we have alphabetic letters, numbers, and punctuation marks which can be combined to produce words and proper emphasis using punctuation. A page of English characters is separated into sentences, words, names, and punctuation using English lexical conventions. Vocabulary, the meaning of individual words, is also governed by the same conventions. However, the grammatically correct organization of English words, sentences, punctuation, etc. is ruled by syntax.

For example, the English lexical conventions determine that “Missy and me” is made of three tokens with the words ‘and’ and ‘me’ as well as the name ‘Missy’. These

conventions do *not* rule that the grammar incorrect, that is a completely different facet covered by syntax.

C++ has its own lexical conventions: characters akin to our alphabet, numbers, and punctuation marks, keywords akin to our vocabulary, and operators akin to our punctuation marks.

The valid characters used in C++ source code are known as *basic source characters*. Other characters may be used in the data of C++ programs, but not legally in its instructions. In an English essay, for example, you may use a foreign character as a title, reference, or whatever as long as it doesn't interfere with the grammar and is isolated from the legal *alphanumeric*, or alphabetic and numeric, characters. The basic source characters include invisible *control characters* space, tab, form feed, and new-line as well as the following visible, also known as *graphical*, characters:

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	_	{	}
[	]	#	(	)	<	>	%	:	;	.	?	*
+	-	/	^	&	|	~	!	=	,	\	“	‘

Characters beyond those listed above may appear within the data utilized in a C++ program, such as within the double quotes of a text string like ‘Hello World’. Some C++ sources, particularly those written in a different language such as French, may contain characters other than those above. Unfortunately those are not legal as per the C++ standard, but still supported by many compilers. The above characters can be seen of a base guarantee, not necessarily a limit.

The source character set is the character set that the source code was written in. Specifically it would indicate which code represents each of the basic source characters in the table above. However, a compiler on a system using a certain character set *may* accept source code written in a different character set as long as it was properly specified. At the same time, source code written in the native character set for the current platform (operating system and compiler) may be compiled into code that is intended for a different character set.

In C++, the *basic execution character set* is the output character set intended. Most of the time both the source and execution character sets are the same; usually both are ASCII-based. If you ever intend to cross compile, however, you may have to deal with this in a bit more detail. The basic execution character set should contain the same characters as the basic source character set *as well as* the control characters for alert, backspace, carriage return, and NULL[3]. This means those characters are valid in the code generated by a C++ compiler. Code generated by a C++ compiler may contain characters

[3] This special character is explained in detail in Chapter 8.

other than this, but these are something of a guarantee because certain *escape sequences*[4] evaluate to them.

Foreign characters not included in this set can be placed into source code using a *universal character name*. ***This isn't that important, but it needs to be done.***

Trigraph Sequences

??= → #	??(→ [	??< → {
??/ → \	??) →]	??> → }
??' → ^	??! → |	??- → ~

Some rare, or maybe very old, computer systems can't actually handle all of the characters in the source character set shown above. C++ source code on these systems must use *trigraph sequences* to represent certain characters. Anytime a trigraph sequence is encountered in the source it is interpreted as the character it represents.

Author's Opinion: I'd be interested in hearing of *anyone* having to fiddle much with execution character sets and trigraph sequences. My explanation was short due to their rareness, especially where PC's are concerned.

Translation Phases

C++ source code goes through basically three phases[5] *before* it is ready to be compiled into object code: replacement, preprocessing, and parsing. These three phases rely on lexical conventions rather than syntax.

The replacement phase simply translates all named universal characters and trigraph sequences into their corresponding character codes in the source character set. Secondly, all *comments* are removed. Another replacement is each new line control character that is preceded by a backslash is removed along with the backslash. That is, if you place a backslash at the end of a line this stage will make sure the physical line break is removed. This is useful in *macro functions* which are explained in Chapter 9.

The next stage, preprocessing, translates and processes each *preprocessor command* in the source code. Preprocessor commands are represented by *preprocessor statements* which are individual lines in the source code that begin with a pound sign (#). The line '#include <iostream.h>' in 'hello.cpp' is a preprocessor statement that evaluates to the preprocessor command 'include'. The anatomy of these statements is explored in detail in Chapter 9. For now, just realize that they occupy a single, whole line each.

Lastly, just before compilation occurs, the source code is parsed into *tokens* using lexical conventions. Each token is associated with a line number (row) and column as it relates cto the source file it was contained in. These values are shown to you by the compiler when a syntax error is encountered that seems to originate around that particular position.

For example, line 5 of 'hello.cpp' will be parsed into the following individual tokens:

---

[4] Escape sequences are explored later on in this chapter.

[5] The standard lists seven (7) explicit steps to do this, but I think some of them are redundant; I believe it's easier to see three distinct stages.

cout	<<	“Hello World”	<<	endl	;

The compiler now no longer sees each character individually, it sees whole words. When we speak to each other in our native tongue, we speak in whole words rather than spelling out each letter of every word. How many people do you know that say “Hello” by saying “Aech Eee Ell Ell Oh”? We divide what we’re going to say or write into words, our own version of tokens.

Parsing

The parsing stage of source code works from top to bottom, left to right. Characters are picked up in that order and tokens are generated as sequences are recognized. For example, in English we know a word begins with an alphabetic character and ends when we reach any sort of punctuation symbol besides a hyphen or space. Similarly in C++, tokens are recognized based on how they begin. A token is known to end when any amount of *white space* is encountered *or* the first character of a different *token type* is encountered.

It is important to remember that the parsing of **all** characters is *case sensitive*. This means that lower-case letters are seen as being completely separate characters from upper-case ones; they are different character codes to begin with. Thus, the character ‘A’ is not the same as ‘a’. Keep this in mind anytime you are dealing with any kind of name.

White space is a term coined originally from documents composed on typewriters and includes any kind of character whose display glyph is invisible. This includes blanks or spaces, horizontal and vertical tabs, newlines, formfeeds, and *comments*. Simply put, you can separate tokens in C++ by pressing the space bar, tab key, or enter key. All white space in C++ source is essentially *ignored* except to separate tokens.

Intertwined with white space there are five types of tokens that are recognized in C++: *identifiers*, *keywords*, *literals*, *operators*, and *punctuators*. These relate to things we understand in English: identifiers and literals are like nouns, keywords and operators are like verbs, and punctuators are like punctuation. Each of these tokens are summarized below and then explained in more detail later on.

Identifiers are named things in a C++ program. Think of pronouns, or *named* nouns, in English: “Neil C. Obremski” rather than “author” for example. Identifiers in C++ are used to *identify* something such as functionality, data storage, or both. The identifiers in ‘hello.cpp’ are ‘cout’ and ‘main’. Valid identifiers can contain any amount alphanumeric characters and underscores, but *must* begin with an alphabetic character *or* an underscore. Identifiers are explored in more detail in Chapter 2.

Keywords are the special words of a language and each one has a special purpose and use. C++ contains the following sixty-three (63) keywords, all of which will be covered through the course of this book:

C++ Keywords

asm	auto	bool	break	case	catch	char
class	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

Literals are explicit, name-less pieces of data that represent exactly what they are seen as. If an identifier is a coupon for a pizza, then a literal is simply having the pizza itself. With an identifier, you must go get the pizza using the coupon, but with a literal you already have it. The literals used in 'hello.cpp' are '0' (for 'return 0;') and 'Hello World'. The concept of literals becomes clearer during Chapter 2 where identifiers are explored in detail.

Operators represent built-in functionality that is applied to data through *expressions*. An operator can be seen as a mathematical transformation. Operators are implied through special symbols and alternatively special word tokens. The only operator used in 'hello.cpp' is '<<'. Operators can be customized in C++, through a process known as *operator overloading*, to do special things in certain situations. The operator in 'hello.cpp' has been customized to work with 'cout'.

Lastly punctuators are terminating tokens which signify the end of *code statements* which are something like sentences in English. The only punctuator in C++ is the semi-colon while in English we have many punctuation marks.

Literals

The WYSWIG[6] concept is true of literals in C++ source. They are *literally* the data they represent and there are five kinds: *integer*, *character*, *floating*, *string*, and *boolean*. How each of these is represented in memory was covered in Part 1, Chapter 3. This section covers how to write them into C++ source, not what they are and how they are stored.

Boolean literals are the exception. These types of literals represent a value that is either on or off, yes or no, flagged or not flagged, set or reset, and lastly true or false. Those keywords, **true** and **false**, are the only two boolean literals there are. They are used in boolean logic which is covered in Chapter 3.

[6] What You See Is What You Get.

Integer literals are whole numbers that are represented as integers in digital storage. Any decimal number without a decimal point that you write in C++ is an integer literal. All literals, including integers, can be printed to output using 'cout' and '<<' as was done with 'Hello World' above. For example, to output the integer five (5):

```
cout << 5 << endl;
```

By default 'cout' will print all integers in decimal form. However, you *can* write integer literals using the hexadecimal and octal number systems as well. A hexadecimal integer literal is written by adding the prefix '0x' where the 'x' can also be upper-case. Octal integer literals are written by prefixing the number with any amount of zeroes (0), though just one will work fine. The following lines of code will all output fifteen (15):

```
cout << 15 << endl;
cout << 0x0F << endl;
cout << 017 << endl;
```

Writing negative integers involves using sign operators which will be explained later on.

There are two suffix letters you can append to the end of an integer literal. Both can be either upper or lower-case and can be combined in any order. The first is 'L' which specifies placing the literal explicitly in a double word storage unit. The second is 'U' which specifies the integer is *unsigned*. Thus, any of the following suffixes for integer literals are valid: "l", "L", "u", "U", "ul", "lu", "uL", "Lu", "Ul", "UL", and "LU".

Floating point literals can be written in two forms: as a decimal number with a decimal point or as a whole number with an exponent. ***Explain both forms of floating point literals.***

Lastly, character literals are display representations of character codes and string literals are sequences of character literals. Remember that each character in memory is actually a number in memory which represents that character. Rather than writing out a series of numbers for each character (imagine what 'hello.cpp' would have looked like) we can write what we want to see as a single character or string of characters.

Individual character literals are simply surrounded by single quotes or apostrophes. These literals represent a single byte of storage and a single character code. A string literal is a list of characters that is surrounded by double quotes and represents one or more bytes in storage. The following would produce 'Hello World' just as well:

```
cout << 'H' << 'e' << 'l' << 'l' << 'o' << ' ' << 'W' << 'o' << 'r' << 'l' << 'd' << endl;
```

Only a single character can be represented by a character literal. A string literal, on the other hand, may have an arbitrary amount of characters within it. Each character in a string, be it visible or invisible, graphical or control, represents a single byte in memory.

A list of characters also has an additional invisible character at the end, the terminating NULL character, which allows the end of the string to be detected.

Control characters cannot be typed into a string or character literal, they should be inserted using *escape sequences* also known as *escape characters*. Escape sequences are like trigraph sequences are to source code. All of these begin with a backslash character which means to use a backslash character in a character or string literal, you need to write it twice. That is, the escape sequence to insert a backslash is *two* backslashes.

Escape Sequences

name	character	sequence
new-line	NL	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	ooo	\ooo
hex number	hhh	\Xhhh

Thus, you could change the primary line of 'hello.cpp' to the following:

cout << "Hello\n \\\n \"World\"" << endl;

And the output from this would be:

Hello
\
"World"

The '\n' escape sequence is ubiquitously synonym to '\f', because both produce a character code of 10 (0xA) on most compilers. It is recognized as a way to begin a new-line and doesn't always correspond to the new-line sequence necessary on a specific platform. The functionality processing '\n' knows to convert it to such at the correct times. The escape sequences '\r' and '\f' shouldn't be used unless you specifically want the codes of 13 (0xD) or 10 (0xA) respectively. Proper use of new-line sequences is explored in Chapter 10.

Insert more information on escape sequences.

Character literals and string literals cannot span multiple lines, but two strings back to back will be automatically *concatenated* somewhere during the parsing phase, before actual compilation. This means that if you have two distinctly separate string literals, one after another, they will be turned into one:

cout << “Hello “ “World” << endl;

The two string literals above, ‘Hello ‘ and ‘World’, are automatically concatenated into the one ‘Hello World’. Concatenation is a fancy word for saying “attach” or “append”. Because C++ treats white space indifferently, you can organize two string literals on separate lines and they will still be combined in the same way:

cout << “Hello “
“World” << endl;

Automatic concatenation will occur between any amount of string literals that are written after one another. Since character literals can only contain a single character this will *not* work for them.

Syntax

The grammar of a language is defined by its syntactical rules, known as *syntax*, which can be construed as the correct way to combine lexical tokens so as to produce the desired result. Think of the English language and its grammatical rules, which include punctuation and the correct organization of word types such as verbs and nouns. For example, “Bob up ! ball the picked” is *lexically* correct, but not *syntactically* correct. That is, the lexical tokens are correctly formed, but they are *not* correctly combined to produce meaning. The same tokens could be correctly combined if they were in the following order: “Bob picked up the ball!”.

Source code in C++ is made up of different types of *statements* that can contain *expressions*. A statement states a particular way to organize and utilize the expressions that express instructions and their corresponding values. In comparison to English, an expression is somewhat equivalent to a combination of words representing an action while a statement is like a structure of these actions: sentence, paragraph, list of sentences, etc. A statement is a way to correctly form an instruction or set of instructions using expressions, just like there is a correct way to form a sentence in English to get your meaning across.

Expression syntax governs how pieces of source code, known as lexical tokens, represent an instruction or value, just like how words in English represent a verb or noun. Statement syntax governs how these expressions and other tokens are combined for proper organization. The whole combination of these is the overall syntax that C++ source code is ruled by. When syntax is mentioned without an explicit context, it refers to this overall syntax.

All statements of a particular type have a specific syntax that refers to how they begin, end, and what they contain. Expressions determine the actual instructions *and* values, but the statements they are contained in determine *if*, *when*, and *how* many times they are executed. The relationship between expressions and statements is akin to that of structure and content.

Statements

There are basically three types of statements: *declaratory*, *instructional*, and *flow control*. There are eight explicit categories, but each of these has a purpose equivalent to the three types I listed: *labeled statements*, *expression statements*, *compound statements*, *selection statements*, *iteration statements*, *jump statements*, *declaration statements*, and *try-catch statements*. Each of these will be explored in utter detail throughout the later chapters of this book, but for now I'll simply go over the basic categories I use.

Declaratory statements *declare* things in a C++ program and typically associate them with an identifier, something along the lines as declaring who you are: "I am Neil". Instructional statements execute instructions, including setting, modifying, and removing values. For example, an instructional sentence in English might be: "Neil, write a book with your laptop". Flow control statements alter the flow of execution within the program. Thus, these statements can determine when or if other statements are actually performed. For example, "If you have time, write a book with your laptop". In that sentence, the first part "If you have time" would be flow control.

C++ is different from English in that it is a language explicitly intended for *programming*. Because of that, some things are difficult to equate to our common written/spoken languages, like English. Another big difference is it is extremely logical. Lastly it must be mentioned that there is a certain amount of nesting allowed whereby some statements are made up of one or more other statements. For example, in "If you have time, write a book with your laptop", the sentence as a whole is considered to be flow control even though it's latter part is instructional. Thus, that instructional piece is "nested". Please note this is a rude analogy compared to C++'s statement nesting.

The 'hello.cpp' program contains three statements and each is of a different category. It's worth mentioning that the first line is a *preprocessor instruction* and not a C++ statement. Anytime a line begins with a pound sign (#), it's a preprocessor instruction and follows different lexical and syntactical rules than normal C++. Preprocessing is covered in detail in Chapter 9.

The first statement of 'hello.cpp' begins on line 3 with 'int main()' and it is *declaratory*. It declares "main" as a function and is followed by a pair of curly braces ('{' and '}') in which the rest of the program is contained. That pair of braces and their contents are known as a *compound statement* or *block*. In this case, that statement block has been

declared as being associated with ‘main’. The whole program (it’s small) could be summed up by: “I am ‘main’ and these are my instructions.”

Although the block of instructions is associated with ‘main()’ it is not nested. A nested statement must occur *within* another statement while in this case the block is simply trailing it. Think of a truck pulling a trailer. The trailer is *not* nested within the truck, but it is still associated with it; the same is the case with the block following ‘main()’. Chapter 3 contains a multitude of information on blocks, nesting, and flow control.

The first statement within the block is instructional. It’s telling ‘cout’ to use “Hello World” and ‘endl’. Something like, “Hey ‘cout’, do something with these will ya?”. The ‘cout’ functionality responds by outputting the text “Hello World” followed by moving to a new line.

The last statement in the block is for flow control; it tells the caller that the function is done and to return to where it was started. Since this is the only function in the program, this return causes the program to exit and control to *return* to the operating system.

If you are using strict C++ then you’ll also have an additional declaratory statement above ‘main()’ which reads ‘using namespace std;’.

Our whole ‘hello.cpp’ program, as a movie script, might look like so (set aside reality and imagine the operating system having the source code rather than the machine code):

Operating System: [runs ‘hello.cpp’] Hey ‘main’, do your thing.
Main: I am a function called ‘main()’. I will process my instructions. First, ‘cout’ you must do something with “Hello World” and ‘endl’.
Cout: Alright, I’ll output “Hello World” and then move to a new-line.
Main: Thanks, now I see that my job is done and I should exit with a code of zero.
[Gives zero to Operating System]
Operating System: Thanks ‘main’, you have executed successfully.

All C++ programs are built from statements, but most of the actual instructions are done through *expressions*.

Expressions

Beyond flow control, all instructions in C++ are achieved through *expressions*. Similarly where in mathematics an expression is a formula for finding an answer, in C++ they are a process for producing a result. Not only that, an expression wholly represents a value because all values are a result of something. For example, you represent yourself simply by “being”. That is something you can do, because you may not “be”, and therefore the value of yourself is expressed through the action of “being”.

Expressions that directly represent something, as in you "being" yourself, are known as *primary expressions*. This means the only action needed to find the result is to look exactly at what is "being" used. For example, all literals are primary expressions. They literally represent a value and therefore require no additional action to produce the result of themselves. Although this is confusing, you don't have to remember it in terms of philosophy. Just know that any value in C++, whether literal or identified, is a primary expression. For example, whether or not you have an egg already in your hand (i.e. literal) or a carton of eggs from which you must pick one (i.e. identified), both are considered primary expressions of an egg.

Easily identified expressions are ones that actually use values to produce a result. Machines are submerged completely within the confines of numbers, so it's no wonder that C++ is math-oriented and it's expressions follow that tradition. There are some slight differences between math expressions and those found in C++. For one, a math expression is basically a declaration of how to achieve a value, such as:

5 + 5 = 10

This declaration might be useful to know, but it's not commanding enough to be useful in programming of which C++ expressions are geared towards. For example, saying that 'cout << "Hello World" << endl;' will produce the output we want is different than actually producing that output. The act of changing something using an expression is known as a *side effect*. It isn't necessarily something bad, it's just something that happens. When 'Hello World' is used with 'cout' as we have done, the side effect is the text appearing in the output.

Author's Note: If you are weak in the mathematics arena, don't fret or put down this book. I've only recently begun furthering my education there and I've been able to program in C++ effectively for years without it. I've even toyed with the idea of writing a book on mathematics geared to C++ programmers.

An expression is usually defined as any sequence of *operators* and *operands* that produces a value or generates a side effect. An operand is a value, which may be another expression known as a *subexpression*, while an operator is a command to be done with the operands. Operators are also used to separate the list of operands; that is, no two operands will ever be written next to each other. The math expression given earlier has two operands ('5' and '5') and one operator ('+'). Expressions break down into simple actions based on a single operator and it's surrounding operands. These *simple expressions* are combined to form *complex expressions*, also known as *compound expressions*. I refer to each individual simple expression as an operation since it's based on an operator.

Operators and Operands

An operator is a symbolic sequence of characters that represents an operation to be performed. Many operators are a single character, such as '+' in Math, which is known as the 'plus symbol' and it's operation is the addition of it's two surrounding operands. The operators in C++ are similar to math in that regard, but some of them are multiple symbols that represent a single operator (such as '++') and others are character sequences that look like actual English words (such as the 'sizeof' operator).

Operands are values that are used with an operator. For example, the number '5' is just a value, but if it is used with an operator it becomes an operand. The operator's action is based on the operand. Similarly, you are a person, but if you go into the Hospital you become a patient. The operations performed in the hospital are based on you as the patient.

When a C++ compiler is parsing a source file, it gathers expressions by looking for known operators and properly formed literals and identifiers. In this way it is agnostic to white space, because it looks for individual pieces of expressions, not necessarily where they appear, as long as they are in order.

Operations only have up to three operands, and all but one have two or less. That means that most operations boil down to two operands and an operator, e.g. two values and an action. The combination of these is what I refer to as an operation and each one results in some value[7]. These values are known as *intermediate values* or *intermediate results* when they are used as operands for other operations. Consider the following:

5 * 5 + 5

The first operation is '5 * 5' which results in the intermediate value of '25'. This intermediate value is then used in the second operation which would be '25 + 5'. The second operation isn't executed until the first completes, because it needs the result value in its own equation.

Author's Note: A computer stores intermediate results in a CPU register. So, instead of seeing this as a single instruction it sees it as two. Something like 1.) Multiply five by five and put the result into a register; 2.) Add the register value to 5.

C++ Operators

Category	Operator	Type	Syntax	Summary	Notation	Precedence	Associativity
logical	++	unary	l++;		postfix		left-to-right
binary							

The symbolic character sequences that distinguish operators are many times used to represent multiple, and sometimes completely different, operations. The compiler determines the operator actually used, based on the context in which it appears, e.g. the syntax or operands associated with the operation. For example, if you place the

[7] Some operations result in *nothing*, but that is seen as a special value known as 'void'. When using some functions you can see how considering this an actual value makes a difference.

increment (‘++’) symbol *before* a value, it knows you want to use the prefix increment operator as opposed to the postfix increment operator.

In ‘hello.cpp’, you may notice that ‘<<’ is used with ‘cout’, but the description of that operator doesn’t really fit what we’re doing. This is because that operator has been *overloaded* to work differently with ‘cout’. The term “overloaded” actually refers to *overriding* default functionality. In this case the default operation of “bit shifting” is overridden with the operation of “pushing data to output” using ‘cout’. The overloaded functionality is attached to the *operand* or operands using the operator, which means anytime you use ‘<<’ with ‘cout’ it will always be overridden.

To be done: explain categories, etc.

Blarg.

Precedence and Associativity

The order of which operations in an expression are performed is determined by operator precedence. If you remember basic math and “My Dear Aunt Sally” you’ll remember what order math operations are performed in. C++ uses this same order, but it also contains more operators than that.

To be done: explain associativity, dispel insertion operator associativity myth, etc.

Blarg.

Notation Fixes

Operators have three different written notations, which also refer to when side effects occur: infix, prefix, and postfix.[8]

To be done: explain the different notation “fixes”, etc.

Blarg.

Result Values

An expression always results in a value, but there are two actual categories of values: lvalue’s and rvalue’s. There is also a special value known as “void” which doesn’t, in my opinion, fall into either category.

[8] You may also see these terms associated with expressions, but they have the same meaning.

Lvalues can be used in place of rvalues, but not the reverse. Think of an lvalue as a coupon for a free pizza, and an rvalue as an actual pizza. You can use the coupon to get the pizza, but you can't use the pizza to get a coupon. Lvalues are basically like tags or coupons, they identify a value that exists somewhere but you don't necessarily have. Rvalues, on the other hand, represent the actual value in a way where you can't identify where it comes from or have no need to.

To be done: explain different value categories and significance, lvalues can be rvalus but not reverse, etc.

Blarg.

Expressions of 'hello.cpp'

To be done: analyze expression statement in 'hello.cpp' as well as the return statement.

Blarg.

Basic Output

To be done: output artithmetic results, etc.

Let's get back to the output. You can plainly see 'Hello World', but what of the strange '`endl`' following it and what of all these '<<'? The 'endl' means 'end line' or 'go to the next line here'. For example, try changing line 5 to the following:

```
05      cout << "Hello" << endl << "World" << endl;
```

After compiling and running this you will see the output:

```
Hello
World
```

The words are now on separate lines. This is because of the 'endl' we put in between them. Notice for each new thing that you want to output, you need to put a '<<' in front of it. When you want to output text, as we have done, you must surround it with double quotes. The double quotes will not be printed along with the text; they simply tell exactly what you want to output.

Comments

Author's Opinion: There are a lot of opinions on how comments should be used effectively. Personally, I've seen two extremes at either end of the spectrum that I

dislike. On the one hand there are people who put absolutely *no* comments in their source. It's like wading through alphabet soup trying to form words or looking to the clouds to see shapes that aren't really there. And on the other side there are those that put so many inane comments in their code you feel like you're reading a book on baby names. My advice is to use what you will, but if you're inline comment lines exceed your code lines by at least 3:1, then you're doing too much. And please, don't comment on the obvious.

Summary