Chapter
2

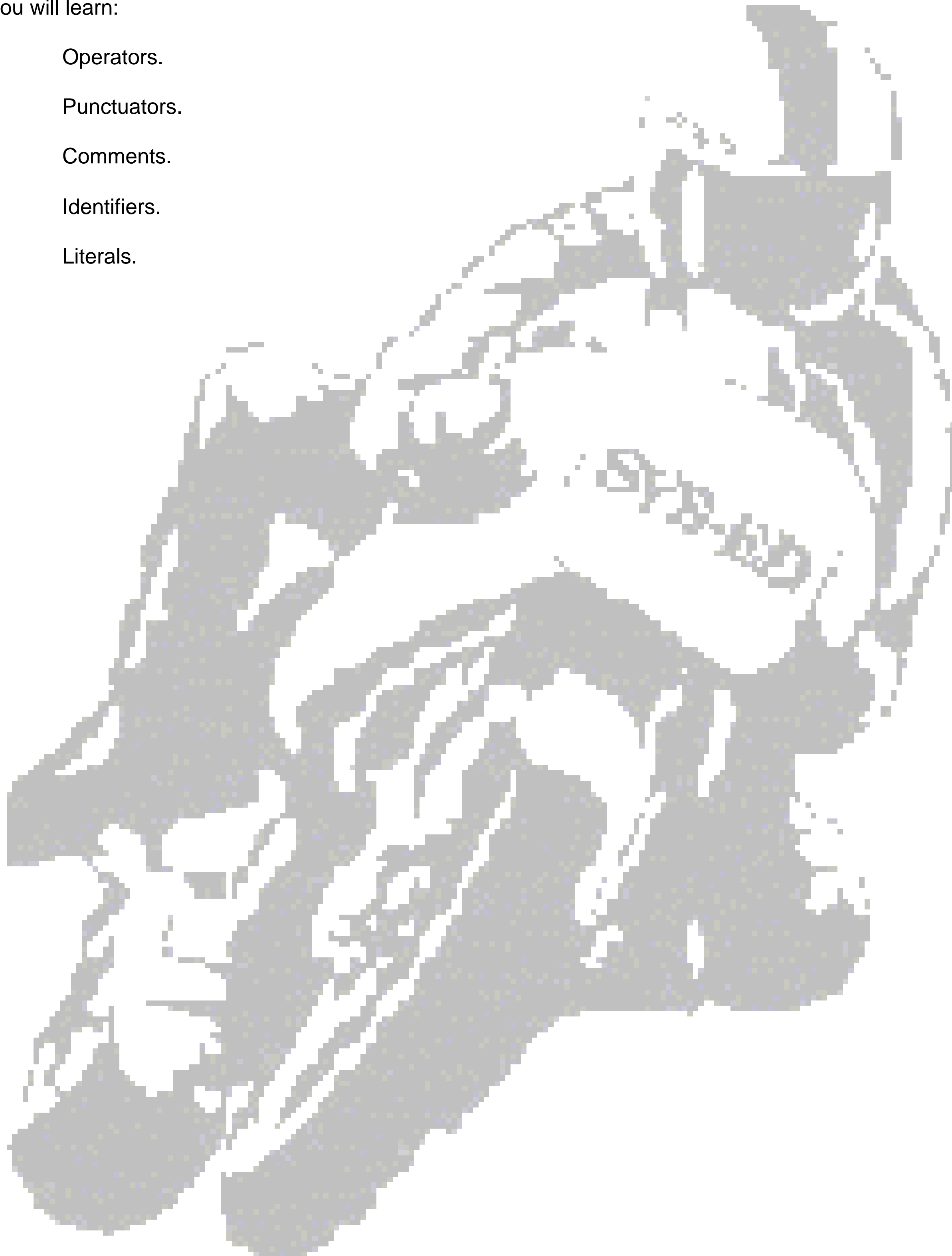
LEXICAL CONVENTIONS

SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.

Objectives

You will learn:

- Operators.
- Punctuators.
- Comments.
- Identifiers.
- Literals.



1 Operators

The operators in C++ are defined as being one of the following:

<table>
<tr><td>[ ]</td><td>( )</td><td>.</td><td>-></td><td>++</td><td>--</td></tr>
<tr><td>&</td><td>*</td><td>+</td><td>-</td><td>~</td><td>!</td></tr>
<tr><td>sizeof</td><td>/</td><td>%</td><td><<</td><td>>></td><td><</td></tr>
<tr><td>></td><td><=</td><td>>=</td><td>==</td><td>!=</td><td>^</td></tr>
<tr><td>|</td><td>&&</td><td>||</td><td>?:</td><td>=</td><td>*=</td></tr>
<tr><td>/=</td><td>%=</td><td>+=</td><td>-=</td><td><<=</td><td>>>=</td></tr>
<tr><td>&=</td><td>^=</td><td>|=</td><td>,</td><td>#</td><td>##</td></tr>
</table>

And the following operators specific to C++:

<table>
<tr><td>::</td><td>.*</td><td>->*</td></tr>
</table>

The operators # and ## are used only by the preprocessor.

Depending on context, the same operator can have more than one meaning.

Example:

The ampersand (&) can be interpreted as:

- a bitwise AND (A & B)
- an address operator (&A)
- in C++, a reference modifier

In the first case, the & is a binary operator; in the second, the & is a unary operator.

1.1 Unary operators

Unary operators operate on one operand.

Turbo C++ provides the following unary operators:

&	Address
!	Logical negation
*	Indirection
++	Increment
~	Bitwise complement
--	Decrement
-	Unary minus
+	Unary plus

The syntax for unary operators is as follows:

<unary-operator> <unary expression>

OR

<unary-operator> <type><unary expression>

1.2 Overloaded Operators

Overloaded operators are those that have been redefined within a C++ class using the keyword operator followed by an operator symbol.

When an operator is thus overloaded, the resulting symbol is called the operator function name.

1.3 Precedence of Operators

In the following table of operator precedence, the Turbo C++ operators are divided into 16 categories.

The #1 category has the highest precedence; category #2 (Unary operators) takes second precedence, and so on to the Comma operator, which has lowest precedence.

The operators within each category have equal precedence.

The Unary (category #2), Conditional (category #14), and Assignment (category #15) operators associate right-to-left; all other operators associate left-to-right.

# Category	**Operator**	**What it is (or does)**
1. Highest	()	Function call
	[]	Array subscript
	->	C++ indirect component selector
	::	C++ scope access/resolution
	.	C++ direct component selector
2. Unary	!	Logical negation (NOT)
	~	Bitwise (1's) complement
	+	Unary plus
	-	Unary minus
	++	Preincrement or postincrement
	--	Predecrement or postdecrement
	&	Address
	*	Indirection
	sizeof	(returns size of operand, in bytes)
	new	(dynamically allocates C++ storage)
	delete	(dynamically deallocates C++ storage)

<table>
<tr><th># Category</th><th>Operator</th><th>What it is (or does)</th></tr>
<tr><td rowspan="3">3. Multiplicative</td><td>*</td><td>Multiply</td></tr>
<tr><td>/</td><td>Divide</td></tr>
<tr><td>%</td><td>Remainder (modulus)</td></tr>
<tr><td rowspan="2">4. Member access</td><td>.*</td><td>C++ dereference</td></tr>
<tr><td>->*</td><td>C++ dereference</td></tr>
<tr><td rowspan="2">5. Additive</td><td>+</td><td>Binary plus</td></tr>
<tr><td>-</td><td>Binary minus</td></tr>
<tr><td rowspan="2">6. Shift</td><td><<</td><td>Shift left</td></tr>
<tr><td>>></td><td>Shift right</td></tr>
<tr><td rowspan="4">7. Relational</td><td><</td><td>Less than</td></tr>
<tr><td><=</td><td>Less than or equal to</td></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td rowspan="2">8. Equality</td><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>9.</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>10.</td><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>11.</td><td>|</td><td>Bitwise OR</td></tr>
<tr><td>12.</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>13.</td><td>||</td><td>Logical OR</td></tr>
<tr><td>14. Conditional</td><td>?:</td><td>(a ? x : y means "if a then x, else y")</td></tr>
</table>

# Category	Operator	What it is (or does)
15. Assignment	=	Simple assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
	<<=	Assign left shift
	>>=	Assign right shift
16. Comma	,	Evaluate

All of the operators in this table can be overloaded except the following:

.	C++ direct component selector	.*	C++ dereference
::	C++ scope access/resolution	?:	Conditional

2 Punctuators

The Turbo C++ punctuators (also known as separators) are:

[ ]	( )	{ }	,	;
:	...	*	=	#

Most of these also function as operators.

2.1 Braces ({ })

The { } braces indicate the start and end of a compound statement.

2.2 Semicolon (;)

The semicolon is a statement terminator.

Any legal C expression (including the empty expression) followed by ; is interpreted as a statement, known as an expression statement.

The expression is evaluated and its value is discarded. If the expression statement has no side effects, Turbo C++ can ignore it.

Semicolons are often used to create an empty statement.

2.3 Colon (:)

Use the colon (:) to indicate a labeled statement:

```
start:    x=0;
...
goto start;
...
switch (a) {
case 1: puts("One");
      break;
case 2: puts("Two");
      break;
...
default: puts("None of the above!");
      break;
}
```

2.4 Ellipsis (...)

An ellipsis (...) consists of three successive periods with no whitespace intervening.

You can use an ellipsis in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types.

Example:

```
void func(int n, char ch,...);
```

This declaration indicates that func will be defined in such a way that calls must have at least two arguments, an int and a char, but can also have any number of additional arguments.

In C++, you can omit the comma preceding the ellipsis.

2.5 Equal Sign (=)

The = (equal sign) separates variable declarations from initialization lists.

```
char array[5] = { 1, 2, 3, 4, 5 };
int  x = 5;
```

- Declarations of any type can appear, with some restrictions, at any point within the code.
- Function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }
//  parameter i has default value of zero
```

The equal sign is also used as the assignment operator in expressions:

```
a = b + c;
```

2.6 Pound Sign (#)

The # (pound sign) indicates a preprocessor directive when it occurs as the first non-whitespace character on a line.

It signifies a compiler action, not necessarily associated with code generation.

3 Comments

A comment is text that the compiler ignores. Comments are normally used to annotate code for future reference. The compiler treats them as white space.

Comments can be used by programmer in testing to make certain lines of code inactive.

In most cases, however, #if/#endif preprocessor directives should be used because although comments cannot benested.

3.1 ANSI C Style Comment

The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters.

Comments using this syntax cannot be nested.

Example:

```
/* Intent:  Comment out this block of code.
Problem: Nested comments on each line of code are illegal.
FileName = String( "hello.dat" ); /* Initialize file string */
cout << "File: " << FileName << "\n"; /* Print status message */
*/
```

3.2 Single Line Comment

A single line commets consists of // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a “single-line comment.”

The single-line form (//) of a comment followed by the line-continuation token (\) can have surprising effects.

Consider this code:

```
1 #include <stdio.h>
2
3 void main()
4 {
5       printf( "This is a number %d", // \
6       5 );
7 }
```

After preprocessing, the preceding code contains errors and appears as follows:

```
1 #include <stdio.h>
2
3 void main()
4 {
5       printf( "This is a number %d",
6 }
```

4 Identifiers

An identifier is a sequence of characters used to denote one of the following:

- Object or variable name.
- Class, structure, or union name.
- Enumerated type name.
- Member of a class, structure, union, or enumeration.
- Function or class-member function.
- typedef name.
- Label name.
- Macro name.
- Macro parameter.

4.1 Naming Rules

1. An identifier can contain:

   C upper or lower case alphabet

   C digit

   C underscore

2. The first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore (_).

3. An identifier must not be a C/C++ keyword.

Use of two sequential underscore characters (__) at the beginning of an identifier, or a single leading underscore followed by a capital letter, is reserved for C++ implementations in all scopes.

You should avoid using one leading underscore followed by a lowercase letter for names with file scope because of possible conflicts with current or future reserved identifiers.

4.2 Case

C++ identifiers are case sensitive: fileName is different from FileName. Identifiers cannot be exactly the same spelling and case as keywords. Identifiers that contain keywords are legal.

Example:

Pint is a legal identifier, even though it contains int, which is a keyword.

4.3 Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.

The following keywords are reserved for C++:

asm	auto	bad_cast	bad_typeid
break	case	catch	char
class	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	except
extern	finally	float	for
friend	goto	if	inline
int	long	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	try	type_info	typedef
typeid	union	unsigned	using
virtual	void	volatile	while
xalloc			

5 Literals

Invariant program elements are called "literals" or "constants".

Literals fall into four major categories:

C integer	C character	C floating-point	C string literals

5.1 Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. You can specify integer constants in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

C To specify integer constants using octal or hexadecimal notation, use a prefix that denotes the base.

C To specify an integer constant of a given integral type, use a suffix that denotes the type.

Decimal

To specify a decimal constant, begin the specification with a nonzero digit.

Example:

```
int i = 157;   // Decimal constant
int j = 0198;  // Not a decimal number; erroneous octal constant
int k = 0365;  // Leading zero specifies octal constant, not decimal
```

Octal

To specify an octal constant, begin the specification with 0, followed by a sequence of digits in the range 0 through 7. The digits 8 and 9 are errors in specifying an octal constant.

Example:

```
int i = 0377;     // Octal constant
int j = 0397;     // Error: 9 is not an octal digit
```

Hexadecimal

To specify a hexadecimal constant, begin the specification with 0x or 0X (the case of the “x” does not matter), followed by a sequence of digits in the range 0 through 9 and a (or A) through f (or F).

Hexadecimal digits a (or A) through f (or F) represent values in the range 10 through 15.

Example:

```
int i = 0x3fff;   // Hexadecimal constant
int j = 0X3FFF;   // Equal to i
```

To specify an unsigned type, use either the u or U suffix. To specify a long type, use either the l or L suffix.

Example:

```
unsigned uVal = 328u;   // Unsigned value
long lVal = 0x7FFFFFL;  // Long value specified
                        // as hex constant unsigned long ulVal = 0776745ul;
                        // Unsigned long value
```

5.2 Characters Constants

Character constants are one or more members of the “source character set,” the character set in which a program is written, surrounded by single quotation marks ('). They are used to represent characters in the “execution character set,” the character set on the machine where the program executes.

There are three kinds of character constants:

- Normal character constants
- Multicharacter constants
- Wide-character constants

Use wide-character constants in place of multicharacter constants to ensure portability.

Character constants are specified as one or more characters enclosed in single quotation marks.

Example:

```
char ch = 'x';                // Specify normal character constant.
int mbch = 'ab';              // Specify system-dependent
                              // multicharacter constant.
wchar_t wcch = L'ab';         // Specify wide-character constant.
```

The only difference in specification between normal and wide-character constants is that wide-character constants are preceded by the letter L.

Example:

```
char schar = 'x';             // Normal character constant
wchar_t wchar = L'\x81\x19';  // Wide-character constant
```

C++ Reserved or Nongraphic Characters

Character	ASCII Representation	ASCII Value	Escape Sequence
Newline	NL (LF)	10 or 0x0a	\n
Horizontal tab	HT	9	\t
Vertical tab	VT	11 or 0x0b	\v
Backspace	BS	8	\b
Carriage return	CR	13 or 0x0d	\r
Formfeed	FF	12 or 0x0c	\f
Alert	BEL	7	\a
Backslash	\	92 or 0x5c	\\
Question mark	?	63 or 0x3f	\?
Single quotation mark	'	39 or 0x27	\'
Double quotation mark	"	34 or 0x22	\"
Octal number	ooo	—	\ooo
Hexadecimal number	hhh	—	\xhhh
Null character	NUL	0	\0

5.3 Floating-point Constants

Floating-point constants specify values that must have a fractional part.

These values contain decimal points (.) and can contain exponents.

Floating-point constants have:

mantissa	specifies the value of the number.
exponent	specifies the magnitude of the number.
optional suffix	that specifies the constant's type.

The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number.

Example:

```
18.46
38.
```

The exponent, if present, specifies the magnitude of the number as a power of 10.

Example:

```
18.46e0          // 18.46
18.46e1          // 184.6
```

If an exponent is present, the trailing decimal point is unnecessary in whole numbers such as 18E0.

5.4 String Literal

A string literal consists of zero or more characters from the source character set surrounded by double quotation marks (").

A string literal represents a sequence of characters that, taken together, form a null-terminated string.

C++ strings have these types:

- Array of char[n], where n is the length of the string (in characters) plus 1 for the terminating '\0' that marks the end of the string.
- Array of wchar_t, for wide-character strings

The result of modifying a string constant is undefined.

Example:

```
char *szStr = "1234";
szStr[2] = 'A';          // Results undefined
```