NOTE A.1

Operator Precedence

<table>
<tr><th>Default operation</th><th>C and C++</th><th>C++ Only</th><th>Associativity</th></tr>
<tr><td>scope</td><td></td><td>::</td><td>left to right</td></tr>
<tr><td>primary</td><td>( ) [ ] -> .</td><td>type( ) const_cast dynamic_cast reinterpret_cast static_cast typeid</td><td>left to right</td></tr>
<tr><td>unary</td><td>++ -- ! ~ (type) + - * & sizeof</td><td>new delete</td><td>right to left</td></tr>
<tr><td>select pointer</td><td></td><td>.* ->*</td><td>left to right</td></tr>
<tr><td>multiplicative</td><td>* / %</td><td></td><td>left to right</td></tr>
<tr><td>additive</td><td>+ -</td><td></td><td>left to right</td></tr>
<tr><td>shift</td><td><< >></td><td></td><td>left to right</td></tr>
<tr><td>relational</td><td>< <= > >=</td><td></td><td>left to right</td></tr>
<tr><td>equality</td><td>== !=</td><td></td><td>left to right</td></tr>
<tr><td>bitwise</td><td>&</td><td></td><td>left to right</td></tr>
<tr><td>bitwise</td><td>^</td><td></td><td>left to right</td></tr>
<tr><td>bitwise</td><td>|</td><td></td><td>left to right</td></tr>
<tr><td>logical</td><td>&&</td><td></td><td>left to right</td></tr>
<tr><td>logical</td><td>||</td><td></td><td>left to right</td></tr>
<tr><td>conditional</td><td>?:</td><td></td><td>right to left</td></tr>
<tr><td>assignment</td><td>= += -= *= /= %= <<= >>= &= |= ^=</td><td></td><td>right to left</td></tr>
<tr><td>throw</td><td></td><td>throw</td><td>left to right</td></tr>
<tr><td>comma</td><td>,</td><td></td><td>left to right</td></tr>
</table>

NOTE A.2

Default Operator Meanings

<table>
<tr><th colspan="6">Operators Common to Both C and C++</th></tr>
<tr><td>( )</td><td>Function Call</td><td>[ ]</td><td>Array access</td><td>−></td><td>Struct/Union Ptr</td></tr>
<tr><td>.</td><td>Struct/Union Mbr</td><td>++</td><td>Increment</td><td>- -</td><td>Decrement</td></tr>
<tr><td>!</td><td>Logical Negation</td><td>~</td><td>One's Complement</td><td>(type)</td><td>Typecast</td></tr>
<tr><td>+</td><td>Unary Plus</td><td>-</td><td>Unary Minus</td><td>*</td><td>Indirection</td></tr>
<tr><td>&</td><td>Address</td><td>sizeof</td><td>Byte Count</td><td>*</td><td>Multiplication</td></tr>
<tr><td>/</td><td>Division</td><td>%</td><td>Modulus</td><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Subtraction</td><td><<</td><td>Left Shift</td><td>>></td><td>Right Shift</td></tr>
<tr><td><</td><td>Less Than</td><td><=</td><td>Less or Equal</td><td>></td><td>Greater Than</td></tr>
<tr><td>>=</td><td>Greater or Equal</td><td>==</td><td>Equality</td><td>!=</td><td>Inequality</td></tr>
<tr><td>&</td><td>Bitwise And</td><td>^</td><td>Exclusive Or</td><td>|</td><td>Bitwise Or</td></tr>
<tr><td>&&</td><td>Logical And</td><td>||</td><td>Logical Or</td><td>?:</td><td>Conditional</td></tr>
<tr><td colspan="3">= += -= *= /= %=
<<= >>= &= |= ^=</td><td>Assignment</td><td>,</td><td>Comma</td></tr>
</table>

<table>
<tr><th colspan="4">C++ Only Operators</th></tr>
<tr><td>::</td><td>Scope Resolution</td><td>typeid</td><td>Type Identification</td></tr>
<tr><td>type( )</td><td>Typecast</td><td>new</td><td>Allocate Memory</td></tr>
<tr><td>const_cast</td><td>Typecast</td><td>delete</td><td>Deallocate Memory</td></tr>
<tr><td>dynamic_cast</td><td>Typecast</td><td>.*</td><td>Member Dereference</td></tr>
<tr><td>reinterpret_cast</td><td>Typecast</td><td>−>*</td><td>Indirect Member Dereference</td></tr>
<tr><td>static_cast</td><td>Typecast</td><td>throw</td><td>Throw Exception</td></tr>
</table>

NOTE G.1

C/C++ Coding Style Guidelines

1. Introduction

The C and C++ languages are free form, placing no significance on the column or line where a token is located. This means that in the extreme a program could be written either all on one line with token separators only where required, or with each token on a separate line with blank lines in between. Such variations in programming style can be compared to accents in a spoken language. As an accent gets stronger the meaning of the conversation becomes less understandable, eventually becoming gibberish. This document illustrates some of the most commonly accepted conventions used in writing C/C++ programs, providing the consistent guidelines needed to write "accentless" code.

2. Implementation Files and Header Files

Implementation files typically have a *.c* or *.cpp* extension and will minimally contain any needed non-inline function definitions and external variable definitions, although they often also contain many of same types of things contained in header files. Header files typically have a *.h* or no extension and are included in other files via a #include directive. They typically contain items like macro definitions, inline function definitions, function prototypes, external variable referencing declarations, templates, typedefs, and type definitions for classes, structures, unions, and enumerations. They also contain #include directives when necessary to support other items in the file. Header files must never contain function definitions, external variable definitions, "using" directives or statements, or run time code that is not part of a macro or an inline function. In applications where multiple implementation files include many of the same header files it is sometimes acceptable to place all of these #include directives in a single header file and include it instead.

Standard header files are typically supplied with the compiler, while 3rd-party library vendors provide custom header files to support their products. In addition, programmers are always free to create their own custom header files to meet their needs. Header files must always be logically organized, includable in any order, and each must implement an "Include Guard" to prevent the multiple inclusion of its contents in another file, even if the header is included multiple times.

3. File Organization

The suggested order for some common file items is as follows, with a blank line placed between each group. The most important consideration is that a something that relies on something else be placed after it. Except for item 1, which is always required, not all files will contain all of these while some files may contain items not shown:

1. A block comment containing information about the contents and functionality of the file, the developer, the revision history, and anything else that might be helpful in understanding and maintaining it;
2. #include directives;
3. "using" statements/directives (C++);
4. #define directives (C);
5. **typedef**s and custom data type definitions;
6. **const** variable declarations (C++);
7. External variable declarations;
8. Function prototypes;
9. Function definitions in some meaningful order, with a block comment before each describing its syntax and purpose.

NOTE B.1

The ASCII Character Codes

Control Characters

DEC	OCT	HEX	CHR	DEC	OCT	HEX	CHR	DEC	OCT	HEX	CHR	DEC	OCT	HEX	CHR
000	000	00	NUL	032	040	20	<sp>	064	100	40	@	096	140	60	`
001	001	01	SOH	033	041	21	!	065	101	41	A	097	141	61	a
002	002	02	STX	034	042	22	"	066	102	42	B	098	142	62	b
003	003	03	ETX	035	043	23	#	067	103	43	C	099	143	63	c
004	004	04	EOT	036	044	24	$	068	104	44	D	100	144	64	d
005	005	05	ENQ	037	045	25	%	069	105	45	E	101	145	65	e
006	006	06	ACK	038	046	26	&	070	106	46	F	102	146	66	f
007	007	07	BEL	039	047	27	'	071	107	47	G	103	147	67	g
008	010	08	BS	040	050	28	(	072	110	48	H	104	150	68	h
009	011	09	HT	041	051	29	)	073	111	49	I	105	151	69	i
010	012	0a	LF	042	052	2a	*	074	112	4a	J	106	152	6a	j
011	013	0b	VT	043	053	2b	+	075	113	4b	K	107	153	6b	k
012	014	0c	FF	044	054	2c	,	076	114	4c	L	108	154	6c	l
013	015	0d	CR	045	055	2d	-	077	115	4d	M	109	155	6d	m
014	016	0e	SO	046	056	2e	.	078	116	4e	N	110	156	6e	n
015	017	0f	SI	047	057	2f	/	079	117	4f	O	111	157	6f	o
016	020	10	DLE	048	060	30	0	080	120	50	P	112	160	70	p
017	021	11	DC1	049	061	31	1	081	121	51	Q	113	161	71	q
018	022	12	DC2	050	062	32	2	082	122	52	R	114	162	72	r
019	023	13	DC3	051	063	33	3	083	123	53	S	115	163	73	s
020	024	14	DC4	052	064	34	4	084	124	54	T	116	164	74	t
021	025	15	NAK	053	065	35	5	085	125	55	U	117	165	75	u
022	026	16	SYN	054	066	36	6	086	126	56	V	118	166	76	v
023	027	17	ETB	055	067	37	7	087	127	57	W	119	167	77	w
024	030	18	CAN	056	070	38	8	088	130	58	X	120	170	78	x
025	031	19	EM	057	071	39	9	089	131	59	Y	121	171	79	y
026	032	1a	SUB	058	072	3a	:	090	132	5a	Z	122	172	7a	z
027	033	1b	ESC	059	073	3b	;	091	133	5b	[	123	173	7b	{
028	034	1c	FS	060	074	3c	<	092	134	5c	\	124	174	7c	|
029	035	1d	GS	061	075	3d	=	093	135	5d	]	125	175	7d	}
030	036	1e	RS	062	076	3e	>	094	136	5e	^	126	176	7e	~
031	037	1f	US	063	077	3f	?	095	137	5f	_	127	177	7f	del

The values of all successive members of the numeric and alphabetic character sets are sequential. Thus, there is a fixed difference between corresponding members of the upper and lower case character sets and a similar fixed difference between the numeric and alphabetic character sets. The first 32_{10} characters are called "control characters" because they have been historically used for message control rather than for displaying actual printable characters. Such message control can range anywhere from moving the screen cursor to the next line to implementing complex data communication protocols. Two different names are commonly associated with each control character:

A. The first is based upon how the character is commonly used in data communication protocols. In this case the character's name appears as an abbreviation in the table. For example, BS stands for "backspace", LF stands for "line feed", SYN stands for "synchronize", ESC stands for "escape", etc.
B. The second is based upon the character's physical position in the table itself. That is, BS, LF, SYN, and ESC from the previous paragraph become "control-H", "control-J", "control-V", and "control-left-bracket". These are often abbreviated ^H, ^J, ^V, and ^[for ease of documentation. The correct name for any control character is easily determined by simply adding 64_{10} (same as 40_{16} and 100_8) to the value of that character and then putting the word "control-" in front of the resulting character.

WHAT IS UNICODE?

Unicode is an internationally standardized encoding scheme intended to provide a unique platform-independent numeric representation for every approved character in every approved written language in the world. It is compatible with ASCII in that the first 128 characters of each have the same numeric values (ASCII only has 128 characters).

NOTE 0.3A

Part 1 of the "How Not to Program" Series

```
/*  Program to reverse and output the digits of a number entered by the user  */

#include <stdio.h>
#define Twas int
#define the
#define night main()
#define before {
#define Christmas int number, rightDigit, sign = 0;
#define And
#define all printf("Enter your number: ");
#define through scanf("%d", &number);
#define house if (number < 0)
#define Not
#define a
#define creature {
#define was number = -number;
#define stirring sign = 1;
#define even }
#define mouse do
#define The {
#define stockings rightDigit
#define were = number
#define hung %
#define By 10;
#define chimney printf("%d", rightDigit);
#define with number /=
#define care 10;
#define In }
#define hopes while
#define that (number);
#define Saint if (sign)
#define Nicholas puts("-");
#define Soon else
#define would putchar('\n');
#define be return 0;
#define there }

/*  Begin actual program  */

Twas the night before Christmas
And all through the house
Not a creature was stirring
Not even a mouse

The stockings were hung
By the chimney with care
In hopes that Saint Nicholas
Soon would be there
```

NOTE 0.3B

Part 2 of the "How Not to Program" Series

/* Program to reverse and output the digits of a number entered by the user */
/* Identically the same program as in Part 1 */

```
int cdecl printf(const char *format,...);int cdecl scanf(const char *format,...);int main(){int number,
rightDigit,sign=0;printf("Enter your number: ");scanf("%d", &number);if(number<0){number=-number
;sign=1;}do{rightDigit=number%10;printf("%d",rightDigit);number/=10;}while(number);if(sign)puts
("-");else putchar('\n');return 0;}
```

/* Sample program runs - Part 1 or Part 2 */

```
Enter your number: -123
321-

Enter your number: 5678
8765
```

Part 3 of the "How Not to Program" Series

```
/*
 * Reproduced from "The New Hacker's Dictionary" from the entry 'Obfuscated C Contest'.
 * Program to compute an approximation (3.141) of pi - by Brian Westley, 1988.
 * Accuracy of approximation increases as the physical size of the "pie" increases and becomes
 * "rounder".
 */

#include <stdio.h>
#define _ 0xF<00? --F<00||--F-OO--:-F<00||--F-OO--;
int F=00,OO=00;
main() {F_OO();printf("%1.3f\n",4.* -F/OO/OO);}F_OO()
{
               _-_-_-_-_-_
            _-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_
        -_-_-_-_-_-_-_-_-_-_-_-_-
           -_-_-_-_-_-_-_-_-_-
              -_-_-_-_-_-_-
}
```

NOTE C.1

Number Systems For Computer Use And Conversions Between Them

"Never trust a man who can count to 1023 on his fingers"

Hexadecimal, Decimal, Octal, and Binary

The most common number systems used in computer environments are as follows:

System	Base/Radix	Character Set
HEXADECIMAL (HEX)	16	0-9, A-F/a-f
DECIMAL	10	0-9
OCTAL	8	0-7
BINARY	2	0-1

The terms *base* and *radix* refer to the number of unique digits available. For example, decimal numbers are expressed using combinations of the ten digits 0-9. Hex, on the other hand, uses 16 digits where the first six letters of the alphabet are used in addition to 0-9. To avoid confusion when working with multiple bases, numbers are commonly written with a subscript denoting their base. For example, 456.7_{10} is decimal while 456.7_{16} is hex.

The C and C++ programming languages permit numbers to be written in all of these systems except binary. The following table shows the equivalence of the first 16 numbers in all four systems:

Decimal	Hexadecimal	Octal	Binary
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

NOTE 1.1

C/C++ Miscellany

Miscellaneous C Facts

- Developed at Bell Labs in early 1970s by Dennis Ritchie;
- Based upon Algol 60 (1960) and B (1967);
- More similar to PL/1, Pascal, and Ada; less similar to BASIC, FORTRAN, and Lisp;
- 90% of the popular "UNIX" operating system is written in C.

Miscellaneous C++ Facts

- Developed at Bell Labs in early 1980s by Bjarne Stroustrup;
- Is a superset of C;
- Contains many type-safe functions;
- Contains object-oriented programming (OOP) features.

Benefits of C and C++

- High level and structured; has all the useful data types including pointers and strings;
- Run-time library includes functions for I/O, storage allocation, string manipulation, etc;
- Designed with systems programming in mind (i.e., UNIX, Linux, Windows, Mac OS, etc.);
- Permits low level access to the underlying hardware not possible in many other languages;
- Programs are efficient; small "semantic gap" between the language and the computer hardware;
- Language standardization permits portability between computers and operating systems;
- Lack of confining rules permits versatility;
- C++ is designed to permit object-oriented programming.

Pitfalls of C and C++

- Programs can be written cryptically;
- Lack of confining rules provides less protection against programming errors;
- Some symbols are used for multiple purposes and differentiated by context;
- Has no built-in array bounds checking.

C and C++ Language Standards (ANSI/ISO/IEC)

- Standardized definitions of the C and C++ languages prepared and published by one or more of the American National Standards Institute (ANSI), the International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC).
- The C versions are commonly referred to as C89 (also called "ANSI C"), C90, C99, and C11;
- The C++ versions are commonly referred to as C++98, C++03, C++11, C++14, and C++17;
- "Standard Code" / "Standard C" / "Standard C++" is code that adheres to an appropriate language standard.
- All code in this book is compatible with all versions of the standards unless noted otherwise.

Compiling Programs

- Program code to be compiled is called "source code" and is created and edited as a standard text file;
- C source file names conventionally end in *.c* while C++ names end with *.C*, *.cpp*, or *.cxx*;
- Compiler may generate assembly code then invoke assembler, or directly generate an "object" file;
- Object files typically end in *.o* or *.obj*;
- Object files must be "linked" to produce an executable file, which typically ends in *.out* or *.exe*;
- Many compilers compile, assemble, and link as part of a single compiler invocation;
- Early C++ compilers were merely front ends that turned C++ code into C code, then passed it to a C compiler.

Language Interpreters

- Do not compile, assemble, or link the source code;
- Analyze and execute source code instructions as encountered;
- May be implemented for most programming languages;
- Program execution is slower than with compiled code.

NOTE 1.2

Functions and Structured Programming

Functions

- Encapsulated constructs in which all run-time code is contained:

  Run-time refers to activities that occur as program runs, as opposed to load-time, which refers to when a program is initially loaded into memory in preparation for running.

Required Function

- *main* is the only required function and is where program execution always begins;
- For small programs (small is in the eye of the beholder) *main* might be the only function;
- For large programs, *main* primarily calls other functions that do the real work.

Structured Programming

- Functions perform logically independent tasks:

  Such tasks can consist of displaying on the console screen, reading and writing disk files, performing mathematical operations, etc. If these tasks are of general interest, they are usually collected in a special form called a library where they may be used by more than one program.
- Functions can be developed independently of each other:

  Since functions are normally designed to perform only a small portion of the overall program task, each one can be developed independently of the others without concern for the details of the entire program.
- Creating the entire program:

  Once the individual functions are written they are all linked together to form the complete program. If a problem arises it is relatively easy to locate and fix the offending function.

Structured Program Example

```
int main(void)
{
    do
    {
        OpenNeededFiles(fileList);
        GetDataFromReadFiles(whereToPutIt, howMuchData);
        ModifyDataFromReadFiles(newValues);
        WriteModifiedDataIntoWriteFiles(newValues);
        CloseAllFiles();
        DoThis();
        DoThat();
        CheckSomething(expectedResult);
        PrintSomething();
    } while (!Terminate());
    return(exitCode);
}
```

NOTE 1.3

The Syntax of *main*

```
   Pre-ANSI C          Standard C                 Standard C++

6  main()              int main(void)             int main()
7  {                   {                          {
8          ...                 ...                        …
9                              return 0;                  return 0;
10 }                   }                          }
```

A common question concerns the differences between the C *main* functions appearing throughout this book and those in some older sources. Many older sources used original (pre-ANSI) syntax simply because of “tradition”, while the code in this book maintains compatibility with the newest C and C++ language standards. Function syntax will be fully explained later so the following information is simply given “for what it's worth”.

Constraints for *main* are stated in the language standards (e.g., ISO/IEC 9899:2011 section 5.1.2.2.1.1 for C and ISO/IEC 14882:2014 section 3.6.1.2.1-2 for C++). They say in effect that all implementations shall allow *main* to be declared in two ways.

With no parameters:

```
int main(void)     {  /* ... */  }          /*  C and C++  */
int main()         {  /* ... */  }          /*  C++  */
```

or with two parameters (referred to here as *argc* and *argv*, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[])    {  /* ... */  }          /*  C and C++  */
```

The remaining considerations apply to all functions including *main*. If full standards compatibility is checked while compiling a “traditional” *main* such as the one at the beginning of this note, up to three compiler-dependent warnings can typically be generated, which are paraphrased and explained below:

Line 6 Warning: “Return type specifier omitted in function *main*; **int** assumed”
Although pre-ANSI C allowed the type specifier in a function declaration to be omitted, in which case it defaulted to **int**, this is at best a bad programming practice.

Line 6 Warning (C only): “Obsolescent function declarator”
From ISO/IEC 9899:2011 section 6.11.6.1: “The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature”. (In C++ an empty parameter list is the same as **void** and is perfectly acceptable.)

Line 10 Warning: “Function should return a value in function *main*”
Since an omitted function return type defaults to **int**, such a function declaration tells the compiler that an **int** will be returned, but where is it? From ISO/IEC 9899:2011 section 6.9.1.12: “If the **}** that terminates a function is reached and the value of the function call is used by the caller, the behavior is undefined.” In C++, however, an omitted **return** statement in *main* is legal and is treated as **return 0**, although this is not a good practice and a warning may still be generated.

NOTE G.2

4. Comments

Comments should be used wherever there might be a question about the workings or purpose of an algorithm, statement, macro definition, or anything else that would not be obvious to a programmer unfamiliar with the code. It should never be necessary to "reverse-engineer" a program. Comments should explain algorithmic operations, not semantics. For example, an appropriate comment for the statement *distance = rate * time;* might be "position of projectile", while something like "multiply rate by time and assign to distance" says nothing useful. With a wise choice of identifiers code can often be made somewhat self-documenting, thereby reducing the need for as many comments.

The format of a comment is normally determined by its length. Comments occupying more than one line, such as a file title block or algorithm description, should be in block comment form. Except for "partial-line" comments, comments should be placed prior to and aligned with the code they comment as follows. Depending upon the compiler, C++ style comments could cause compatibility issues in C89/C90 code.

A C style block comment (also acceptable in C++):

```
/*
 *   Use a block comment whenever comments that occupy more than one line are needed.  Note
 *   the opening / is aligned with the code being commented, all asterisks are aligned with each
 *  other, and that all comment text is left aligned.
 */
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

An equivalent C++ style block comment (also acceptable in C since C99):

```
//
//   Use a block comment whenever comments that occupy more than one line are needed.  Note
//   the opening / is aligned with the code being commented, all // are aligned with each other, and
//   that all comment text is left aligned.
//
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

C and C++ style "full-line" comments:

```
/* This is a full-line C style comment. */
// This is a full-line C++ style comment.
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

It is possible to over comment a program to the point of making it unreadable. This is especially true when code lines and comment lines are intermixed. Many comments can be reduced to the point of being small enough to fit to the right of the code being commented while still conveying meaning. To be most readable code should reside on the left side of a page with comments on the right, opposite the code they comment. Whenever possible all such comments should start in the same column as each other, as far to the right as possible. Deciding on the appropriate column is a compromise and there will usually be exceptions in any given program. Using tabs instead of spaces when positioning such comments reduces the effect of making minor code changes but may not be interpreted correctly by the printer. The following illustrates these "partial-line" comments:

```
while ((nextValue = getchar()) != EOF)        /* while source file has characters */
{
    if (nextValue == '.')                     /* got sentence terminator */
        break;                                /* don't read any more characters */
    else                                      // must continue reading sentence
        ++charCount;                          // update characters read in sentence
}
```

NOTE G.3

The following examples use exactly the same code but with the comments (if any) placed differently. Which programmer would you hire or be most willing to maintain code for?

WRONG – NO COMMENTS

```
while ((nextValue = getchar()) != EOF)
{
    if (nextValue == '.')
        break;
    else
        ++charCount;
}
```

WRONG – CLUTTERED & HARD TO READ

```
while ((nextValue = getchar()) != EOF)/* while source file has characters */
{
    if (nextValue == '.')/* got sentence terminator */
        break;              /* don't read any more characters */
    else/* must continue reading sentence */
        ++charCount;                    /* update characters read in sentence */
}
```

CORRECT – FULL LINE COMMENTS

```
/* while source file has characters */
while ((nextValue = getchar()) != EOF)
{
    /* got sentence terminator */
    if (nextValue == '.')
        /* don't read any more characters */
        break;
    /* must continue reading sentence */
    else
        /* update characters read in sentence */
        ++charCount;
}
```

CORRECT – PARTIAL LINE COMMENTS

```
while ((nextValue = getchar()) != EOF)  /* while source file has characters */
{
    if (nextValue == '.')               /* got sentence terminator */
        break;                          /* don't read any more characters */
    else                                /* must continue reading sentence */
        ++charCount;                    /* update characters read in sentence */
}
```

NOTE G.5

8. Placement of Braces { }

A point of contention among some C/C++ programmers is the placement of the braces used for compound statements and initializer lists. The two most popular formats are shown on the left and right below and are known as the "brace under" and "brace after (K&R)" styles, respectively. Choose one and use it exclusively and consistently. Do not intermix the two or use a different variation. The braces are sometimes omitted if only one statement is needed with **if**, **else**, **for**, **while**, or **do**:

```
int SomeFunction(void)
{
     declarations/statements
}
```

```
int SomeFunction (void)  (Both forms are the same!)
{
     declarations/statements
}
```

```
if (...)
{
     declarations/statements
}
else if (...)
{
     declarations/statements
}
else
{
     declarations/statements
}
```

```
if (...) {
     declarations/statements
} else if (...) {
     declarations/statements
} else {
     declarations/statements
}
```

```
for or while (...)
{
     declarations/statements
}
```

```
for or while (...) {
     declarations/statements
}
```

```
do
{
     declarations/statements
} while(...);
```

```
do {
     declarations/statements
} while (...);
```

```
switch (...)
{
     declarations
     case ... :
          statements
          break;
}
```

```
switch (...) {
     declarations
case ... :
     statements
     break;
}
```

```
struct or class or union or enum tag
{
     declarations
};
```

```
struct or class or union or enum tag {
     declarations
};
```

```
double factors[] =
{
     initializer, initializer, ...
};
```

```
double factors[] = {
     initializer, initializer, ...
};
```

NOTE G.6

struct/**class**/**union**/**enum** type definitions and initialized variable declarations may be placed entirely on the same line if they will fit, but this is not appropriate for any of the other constructs:

```
struct or class or union or enum tag { declaration, declaration, … };
double factors[] = { initializer, initializer, ... };
```

9. Indenting

Indenting is used strictly to enhance a program's human readability and is totally ignored by the compiler. An indent signifies the association of one or more statements with a previous, less indented construct, as in the following examples:

```
if (...)
{
    x = 1;
    printf(...);
}
else if (...)
    z = y;
else
{
    int t = 2;

    ++v;
}
```

```
for (...)
    while (..)
    {
        x = 2;
        ++d;
    }
do
{
    y = 3;
    x = y + 9;
} while (...);
```

```
int main(void)
{
    int ch = 'A';

    putchar(ch);
    return EXIT_SUCCESS;
}
```

The rules for how and when to indent are simple:

> **1. Braces must be aligned according to the placement formats previously discussed.**
>
> **2. All declarations and other statements associated with a construct must be indented equally from the left edge of that construct.**
>
> **3. The width used for all indents must be consistent throughout any program.**

One final consideration is the width of each indent. A default tab stop is 8 spaces wide on some systems, but this is too wide for most programs because the code ends up going off the right edge of the screen/paper or wrapping around. Because of this an indent width of 3 or 4 spaces (or at least 1/4 inch) is recommended instead. Widths smaller than this tend to be hard to discern while larger values run out of room more quickly. Actual spaces should be used instead of hard tab characters because the system printer often has no way of knowing the editor's tab setting. Thus, it often assumes that it is 8 and produces a program listing whose indenting does not match that seen within the editor itself.

10. Multiple Statements on One Line

Do not put more than one statement on a line. Similarly, put the body of an **if**, **else**, **for**, **while**, **do**, or **case** on the next line. Chained assignments such as *x = y = z = 0;* are permissible as long as all operands are logically related. Multiple variables of the same type may be declared on the same line, comma separated, unless a comment is required. The following examples illustrate some acceptable and non-acceptable formats:

NOTE 1.4

Some Elementary Language Concepts

Comment
- Used to explain and clarify program code;
- C style comment: Starts with **/*** and ends with ***/** and may extend over as many lines as desired;
- C++ style comment: Starts with **//** and ends at the end of the line;
- Either style may be used in either language (C++ style not supported in C89/C90);
- Comments may not be nested; the following is illegal: */* ABCD /* EFG */ */*
- The compiler treats each comment as if it were a single whitespace;
- <u>Poorly-commented code = poor code = poor programmer.</u>

Whitespace
- Any of *blank space*, *end-of-line*, *vertical tab*, *horizontal tab*, *form feed*, *comment*;
- Used to separate tokens and make a program more readable.

Token
- A meaningful group of one or more inseparable non-whitespace characters;
- *goto >> += printf* are tokens. *−234+key* consists of 4 tokens.

Keyword (**reserved word**)
- Words with special meaning to the compiler, e.g., **int for goto sizeof class**, **namespace**, **void**, etc.
- May not be redefined by the programmer.
- There are 44 C kewords in ISO/IEC 9899:2011 and 73 C++ kewords in ISO/IEC 14882:2014;
- Some implementations implement additional non-standard, non-portable keywords;

Identifier
- Name used to designate a variable, a function, a macro, etc. (use meaningful names);
- Any combination of letters, digits, and underscores, not a keyword and not starting with a digit;
- Upper and lower case sensitive; at least the first 31 characters are significant;
- *N9__5 _ ___ Abc5n28 Int* are identifiers. *5ABC ax$ int double &ax* are not;
- Applications programmers should avoid combinations starting or ending with an underscore.

Operator
- Token that specifies an evaluation to be performed, or yields a designator, or produces a side effect, or a combination thereof. An operand is the entity upon which an operator acts;
- Examples: **+** (addition), **/** (division), ***** (multiplication or indirection), etc.

Punctuator
- Token used to form syntactically correct code rather than perform an operation;
- Some tokens are used as both punctuators and operators; usage is differentiated by context.

Object (data object)
- A region of data storage whose contents can represent values.

Variable
- An identifier that designates an object;
- A variable must always be declared before being used.

Expression
- …a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

Lvalue • An expression that designates an object.
Rvalue • The value of an expression.

NOTE 1.5

String Literals and Character Literals

String Literal	Character Literal
A double-quote enclosed sequence of *zero* or more members from the 4 categories listed below	A single-quote enclosed sequence of *one* or more members from the 4 categories listed below

- *Category 1:* Any character from the source character set, except:
  The backslash \ (use \\ to produce \);
  The newline character (the ENTER key), (use **\n** to produce the newline character);
  The single-quote **'** in character literals (use \' to produce **'**);
  The double-quote **"** in string literals (use \" to produce **"**).

- *Category 2:* Simple escape sequence
  Certain "non-printing" characters are so frequently required that special escape sequences have been instituted to represent them:

\a	alert (beep)	*\f*	form feed	*\r*	carriage return	*\v*	vertical tab
\b	backspace	*\n*	newline	*\t*	horizontal tab		

  The following escape sequences are required only as indicated. The desired character may be used alone in all other cases:

\'	single-quote	(needed only in character literals)
\"	double-quote	(needed only in string literals)
**	backslash	(always needed)
\?	question mark	(needed only for trigraph escapes)

- *Category 3:* Octal escape sequence:
  Up to three octal digits preceded by a backslash: *\123* *\7* *\007* *\46*

- *Category 4:* Hexadecimal escape sequence (not available in pre-ANSI C):
  Any number of hexadecimal digits preceded by **\x**: *\x1a* *\xab2cd7ef* *\x007a*

String Literals

Treated as tokens by the compiler and may be continued onto next line

- by using the line continuation character, \, as the last character on the line, or
- by realizing that string literals separated by only whitespace get concatenated

Three of Many Ways to Output "Hello World!"

```
printf("Hello World!\n");                                    /* totally portable */
printf("\x48\145\154\154\x6f\x00020\x57\157\x72\x6c\144\41\xa");   /* ASCII only */
printf("He\154\154\x6f\x00020Worl\144!\xa");                 /* ASCII only */
```

The Value of a Character Literal

The value of a character literal containing more than one character, or containing a character or escape sequence not represented in the basic execution character set or representing a value not within the range of **unsigned char**, is implementation-defined. Otherwise, its value is the value of:

- the single character or character represented by a simple escape sequence, as found in an ASCII, EBCDIC, etc., code chart. For example, assuming ASCII, the value of *'A'* is 65_{10}, the value of *'\a'* is 7_{10}, the value of *'\\'* is 92_{10}, etc.
- the octal or hexadecimal number in an octal or hexadecimal escape sequence.

The Data Type of a Character Literal: (in C it is <u>always type int</u>; in C++ it is <u>always type char</u>)
Character literals are so called not because of their data type but because they are used to represent the values of specific characters.

NOTE 1.6

Expressions

Definition

From ISO/IEC 9899:2011 section 6.5.1: "An **expression** is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof." (clarification: an operator is optional)

There are three main questions that should be answered about every expression used in a program, the first two of which will be discussed here:

1. What does each operator (if any) do?
2. What is the value (if any) of the expression?
3. What is the data type of the expression?

Assuming *y* is a variable of appropriate type with a value of *12*:

Expression	*Operator*	*What does operator do?*	*Expression value?*	*Value*
y	*none*	Nothing	Value of constant or identifier	12
y + 3	+	Add operands together	Sum of operands	15
y / 3	/	Divide left operand by right	Quotient of operands	4
y == 3	==	Compare operands	1 if operands are equal; 0 if not	0
- -y	- -	Decrement operand	Current value minus 1	11
printf("H")	( )	Call a function	Function return value	1
myArray[2]	[ ]	Access an array element	Value of array element	?
y = 3	=	Assign right operand to left	Assigned value of left operand	3

Using the Value of an Expression

The value of an expression may be used wherever needed, such as for assignment to a target variable. Note how the last case above and below illustrates that the assignment operator produces an assignment expression (*y = 3*) that itself has a value that may be used for assignment to another variable.

```
targetVariable  =  y;
targetVariable  =  y + 3;
targetVariable  =  y / 3;
targetVariable  =  y == 3;
targetVariable  =  --y;
targetVariable  =  printf("H");
targetVariable  =  myArray[2];
targetVariable  =  y = 3;            /* chained assignment - are parentheses needed?  (no) */
```

NOTE 1.7

Compound Assignment
Increment/Decrement Operators

Compound Assignment

If the value of an object is to be replaced by its current value modified in some way, most languages express the operation in a way similar to the first column below. Although such expressions are valid in C and C++, they are considered cumbersome and are almost never used. The preferred expressions are shown in the second column and use what are called *compound assignment operators*.

Other Languages	C/C++	Never Use
x = x * 25	x *= 25	
y = y + -s	y += -s	
z = z / (a + b)	z /= a + b	z /= (a + b)
t = t / (q + 7 * (-3 + x))	t /= q + 7 * (-3 + x)	t /= (q + 7 * (-3 + x))

In general, the syntax *object* = *object operator expression* becomes *object operator*= *expression*, where the operation must be between *object* and everything else on the right side of the assignment. Thus, the expression *z* = *z* / *a* + *b* cannot be simplified because *a* and *b* are not involved in a common operation with *z*. No parentheses are ever needed around *expression* in the compound form because that form is independent and <u>does not get converted</u> to the non-compound form. Compound assignment operators are pronounced “plus equals”, “minus equals”, “times equals”, “divide equals”, etc. and can use any of the arithmetic operators:

+=   −=   *=   /=   %=   <<=   >>=   &= |=   ^=

Increment/Decrement Operators

The increment and decrement operators, ++ and --, are used to change an appropriate object's value by 1. The following uses ++ as an example but also applies to --. If it is desired to increment an object's value and that value is not used as part of a larger expression, it makes no difference value-wise whether pre-incrementing, ++*x*, or post-incrementing, *x*++ is used. The only effect in either case is that the object's value increases by 1. If used as a sub-expression within a larger expression, however, there is a difference. Pre-incrementing indicates that the object's incremented value will be used to calculate the value of the larger expression while post-incrementing indicates that the object's non-incremented value will be used. The following examples illustrate the point:

```
int x, y;

x = 1;
y = ++x;                /* pre-incrementing assigns 2 to y;  x == 2 */
            OR
x = 1;
y = x++;                /* post-incrementing assigns 1 to y;  x == 2 */
```

In both cases *x* ends up containing 2 but the value of *y* depends upon whether the value of *x* was incremented before or after it was used in the larger expression. Note that if an object's value is to be changed by 1, the increment/decrement operators should always be used. The following are considered cumbersome and are almost never used: x = x + 1 and x += 1

Note: The cost in computation resources for post-incrementing/decrementing some C++ objects in certain situations can be substantially greater than pre-incrementing/decrementing them.

NOTE 1.8

An Elementary C Program

```c
/*
 *      This comment is a “block” comment.  This program is over-commented for the purpose of
 *      explaining some basic concepts.
 */

#include <stdio.h>                                      /* standard I/O library "header" file */
#include <stdlib.h>                                     /* standard general "header" file */


int main(void)                                          /* declare function main */
{                                                       /* start body of function main */
        int product;                                    /* declare type int variable */
        double sum = 6.08, theDifference;               /* declare two type double variables */
        float value;                                    /* declare type float variable */
                                                        /* blank line for readability */
        puts("hello world");                            /* output message to console */
        puts("goodbye\nworld");                         /* output message to console */

        product = 2 * 16000;                            /* multiply then assign */
        printf("The product of 2 and 16000 is %d\n", product);    /* output message & value */

        product -= 26;                                  /* same as "product = product - 26" */
        printf("The product minus 26 is %d\n", product);          /* output message & value */

        printf("The product plus 1 is %d\n", ++product);          /* math on a function's argument */
        printf("theDifference = %e\n", theDifference = sum);      /* assignment expression as argument */

        value = 3.1416f;                                /* assign to the variable value */
        sum = 4. + value;                               /* add then assign to sum */
        printf("        The sum of 4. and %f is %f\n", value, sum);   /* spaces after the first " */

        printf("Enter an integer and a floating point value: ");  /* prompt user */
        scanf("%d %lf", &product, &sum);                /* console input into product and sum */
        printf("You entered %d %f\n", product, sum);    /* outputs the values back to console */

        return(EXIT_SUCCESS);                           /* return success to calling function */
}                                                       /* end function main */
```

Console Screen After Program Run

```
hello world
goodbye
world
The product of 2 and 16000 is 32000
The product minus 26 is 31974
The product plus 1 is 31975
theDifference = 6.080000e+000
        The sum of 4. and 3.141600 is 7.141600
Enter an integer and a floating point value: 23 67.334
You entered 23 67.334000
■
```

NOTE 1.9

An Elementary C++ Program

```
//
//      This comment is a "block" comment.  This program is over-commented for the purpose of
//      explaining some basic concepts.
//

#include <iostream>                                     // standard I/O stream "header" file
#include <cstdlib>                                      // standard general "header" file
using namespace std;          // Make all names in "std" namespace visible; See Note 1.18 for alternatives.

int main()                                              // declare function main
{                                                       // start body of function main
    int product;                                        // declare type int variable
    double sum = 6.08, theDifference;                   // declare two type double variables
    float value;                                        // declare type float variable
                                                        // blank line for readability
    cout << "hello world" << endl;                      // message to console – use \n instead of endl
    cout << "goodbye\nworld\n";                         // output message to console

    product = 2 * 16000;                                // multiply then assign
    cout << "The product of 2 and 16000 is " << product << '\n';     // output message & value

    product -= 26;                                      // same as "product = product - 26"
    cout << "The product minus 26 is " << product << '\n';   // output message & value

    cout << "The product plus 1 is " << ++product << '\n';   // math on a function's argument
    cout << "theDifference = " << (theDifference = sum) << '\n';

    value = 3.1416f;                                    // assign to the variable value
    sum = 4. + value;                                   // add then assign to sum
    cout << "       The sum of 4. and " << value << " is " << sum << '\n';   // spaces after the first "

    cout << "Enter an integer and a floating point value: ";   // prompt user
    cin >> product >> sum;                              // console input into product and sum
    cout << "You entered " << product << ' ' << sum << '\n';   // outputs the values back to console

    return(EXIT_SUCCESS);                               // return success to calling function
}                                                       // end function main
```

Console Screen After Program Run

```
hello world
goodbye
world
The product of 2 and 16000 is 32000
The product minus 26 is 31974
The product plus 1 is 31975
theDifference = 6.08
       The sum of 4. and 3.1416 is 7.1416
Enter an integer and a floating point value: 23 67.334
You entered 23 67.334
■
```

NOTE 1.11

Outputting Data to the Console Screen in Different Formats in C

The following examples demonstrate typical ways of outputting to the console screen using four different C library functions. *printf* is the most versatile while *putchar*, *puts*, and *fputs* can provide maximum efficiency and smallest code size. The newline character, **\n**, has special meaning and causes the display cursor to move to the first column of the next line. *puts* automatically appends a newline to each string it outputs while *fputs* does not. The examples given for *printf* use default formatting and are representative of many possible combinations, but there are many more. The official documentation for each function should be consulted for complete details.

```
#include <stdio.h>                          /* needed for printf, putchar, puts, and fputs */

/* Strings */
    signed/unsigned char *cp = "Hello World";    or    signed/unsigned char cp[] = "Hello World";

    printf("%s", "Hello World\n");
    printf("Hello World\n");
    fputs("Hello World\n", stdout);
    puts("Hello World");
    printf("%s", cp);
    printf(cp);
    fputs(cp, stdout);
    puts(cp);

/* Characters Individually */
    signed/unsigned char/short/int x;
        printf("%c", x);

    any_arithmetic_type x;
        putchar(x);

/* Numbers */
```

Integer types may be output in decimal, hex, or octal:
(i or d=>signed decimal, u=>unsigned decimal, o=>octal, x=>hexadecimal)

```
int format for:
signed/unsigned char/short/int x;      printf("%i  or  %d  or  %u  or  %o  or  %x", x);

short format for:
signed/unsigned char/short/int x;      printf("%hi  or  %hd  or  %hu  or  %ho  or  %hx", x);

signed/unsigned long x;                printf("%li  or  %ld  or  %lu  or  %lo  or  %lx", x);

signed/unsigned long long x;           printf("%lli  or  %lld  or  %llu  or  %llo  or  %llx", x);

size_t x;   (since C99)                printf("%zd  or  %zu  or  %zo  or  %zx  or  %zi, x);
```

Floating types may only be output in decimal:

```
float/double x;                        printf("%e  or  %f  or  %g", x);

long double x;                         printf("%Le  or  %Lf  or  %Lg", x);
```

NOTE 1.12

Outputting Data to the Console Screen in Different Formats in C++

The following examples demonstrate typical ways of outputting to the console screen using the C++ *cout* ostream class object. In C++ *cout* takes the place of all four C console output functions (*printf*, *putchar, puts*, and *fputs*) and unless non-default formatting is needed, is much easier to use. Note the simplicity of *cout* compared to *printf*. When non-default formatting is desired the code can become somewhat “wordy”. The official documentation for *cout* should be consulted for complete details.

```
#include <iostream>                      // for cout, hex, dec, and oct
using namespace std;

// Strings
    signed/unsigned char *cp = "Hello World";    or
    signed/unsigned char cp[] = "Hello World";
        cout << "Hello World\n";
        cout << cp;                      // no automatic newline

// Characters Individually
    signed/unsigned char x;
        cout << x;

    any_arithmetic_type x;
        cout << char(x);
        cout.put(x);

// Numbers
```

Integer types may output in decimal, hex, or octal. The default output format for integer types upon program startup is decimal and remains so until changed by the program. Once changed, it remains changed until the program changes it again. A common way to change it is to use the *dec*, *hex*, and *oct* manipulators as follows:

```
        cout << hex;                     // change to hex integer output format
        cout << oct;                     // change to octal integer output format
        cout << dec;                     // change to decimal integer output format

        signed/unsigned char x;
            cout << int(x);              // if signed char
            cout << unsigned(x);         // if unsigned char
            cout << +x;                  // both signed and unsigned char

        any non-char integer signed/unsigned x;
            cout << x;
```

Floating types may only be output in decimal:

```
        any_floating_type x:
            cout << x;
```

NOTE 1.13

Inputting Data from the Console Keyboard in Different Formats in C

The following examples demonstrate four different library functions used for reading data from the console keyboard into program variables. *scanf* is the most versatile while *getchar*, *gets*, and *fgets* can provide maximum efficiency and smallest code size. Literal non-conversion characters in the *scanf* control string are not stored but must exactly match input. The **\n**, **\t**, and *space* characters have special meaning and cause *all* whitespace up to the next non-whitespace character to be skipped. All conversion specifications except *%c*, *%[...]*, and *%n* automatically skip all leading whitespace. The examples given for *scanf* use default formatting and are representative of many possible combinations, but there are many more. The official documentation for each function should be consulted for complete details.

#include <stdio.h>   /* needed for *scanf*, *getchar*, *gets*, and *fgets* */
#define LINE_LENGTH 256   // define maximum characters in any input line

/* Strings */

signed/unsigned char cp[LINE_LENGTH];

Skip leading whitespace; store up to next whitespace in *cp* (255 chars max).

    scanf("%255s", cp);

Store everything up to **\n** in *cp*, discard **\n** – Good practice dictates that you NEVER use gets.

    ~~gets(cp);~~

Store everything up to and including **\n** in *cp*, or a maximum of LINE_LENGTH-1 characters.

    fgets(cp, LINE_LENGTH, stdin);

Store everything up to scan set mismatch/match in *cp* (255 chars max).

    scanf("%255[...]", cp);

/* Characters Individually */

signed/unsigned char x;

    scanf("%c", &x);   /* get next character, even if a whitespace */
    scanf("\n%c", &x);   /* skip leading whitespace; get next non-whitespace */

any_arithmetic_type x;

    x = getchar();   /* get next character, even if a whitespace */

/* Numbers */

Integer types may input in decimal, hex, or octal:

(d=>signed decimal, u=>unsigned decimal, o=>octal, x=>hexadecimal, i=>any)

signed/unsigned short x;	scanf("%hd *or* %hu *or* %ho *or* %hx *or* %hi, &x);
signed/unsigned int x;	scanf("%d *or* %u *or* %o *or* %x" *or* %i", &x);
signed/unsigned long x;	scanf("%ld *or* %lu *or* %lo *or* %lx *or* %li", &x);
signed/unsigned long long x;	scanf("%lld *or* %llu *or* %llo *or* %llx *or* %lli", &x);
size_t x; (since C99)	scanf("%zd *or* %zu *or* %zo *or* %zx *or* %zi, &x);

Floating types may only be input in decimal:

float x;	scanf("%e *or* %f *or* %g", &x);
double x;	scanf("%le *or* %lf *or* %lg", &x);
long double x;	scanf("%Le *or* %Lf *or* %Lg", &x);

NOTE 1.14

Inputting Data from the Console Keyboard in Different Formats in C++

The following examples demonstrate typical ways of reading data from the console keyboard using the C++ *cin* istream class object. In C++ *cin* takes the place of all four C console input functions (*scanf*, *getchar*, *gets*, and *fgets*) and unless non-default formatting is needed, is much easier to use. Note the simplicity of *cin* compared to *scanf*. When non-default formatting is desired the code can become somewhat “wordy”. The official documentation for *cout* should be consulted for complete details.

```
#include <iostream>                    // for cin, cout, hex, dec, and oct
#include <iomanip>                     // for setw
using namespace std;

const int LINE_LENGTH = 256;           // define maximum characters in any input line

// Strings
    signed/unsigned char cp[LINE_LENGTH];
```

Skip leading whitespace; store up to next whitespace in *cp* (LINE_LENGTH-1 chars max).

```
        cin >> setw(LINE_LENGTH-1) >> cp;
```

Store everything up to **\n** in *cp*, discard **\n**, or a maximum of LINE_LENGTH-1 characters.

```
        cin.getline(cp, LINE_LENGTH);
```

Store everything up to **\n** in *cp*, leave **\n**, or a maximum of LINE_LENGTH-1 characters.

```
        cin.get(cp, LINE_LENGTH);

// Characters Individually
    signed/unsigned char x;
        cin >> x;                      // skip leading whitespace; get next non-whitespace
        cin.get(x);                    // get next character, even if a whitespace

    any_arithmetic_type x;
        x = cin.get();                 // get next character, even if a whitespace


// Numbers
```

The default integer input format upon program startup is decimal and remains so until changed by the program. Once changed, it remains changed until the program changes it again. A common way to change it is to use the *dec*, *hex*, and *oct* manipulators as follows:

```
        cin >> hex;         // change to hex integer input format
        cin >> oct;         // change to octal integer input format
        cin >> dec;         // change to decimal integer input format
```

Floating types may only be input in decimal.

```
    any_non-char_arithmetic_type x;
        cin >> x;
```

NOTE 1.15

Inputting/Outputting Single Characters in C

Because *scanf* and *printf* ultimately call some form of *getchar* or *putchar* for each character they read or write, there can be some efficiency advantages to calling those functions directly when handling console characters one at a time. When deciding between *scanf* and *getchar* several things must be kept in mind:

scanf:

- Preceding a %c with any of **\n**, **\t**, or *space* will skip all leading whitespace of any kind up to the next non-whitespace character. **\n** is preferred. For readability, avoid a literal *space*;
- Target variables corresponding to *%c must* be one of the **char** types.

getchar:

- Does not skip leading whitespace although skipping is easily achievable using a **while** loop;
- If the return value of *getchar* is to be assigned to a variable, the type of that variable should be **int** or a wider **signed** type to facilitate testing for EOF.

```
#include <stdio.h>                              /* needed for all outputs and inputs */
#include <stdlib.h>                             /* needed for EXIT_SUCCESS */
#include <ctype.h>                              /* needed for isspace */

int main(void)
{
    char myChar = 'A';                          /* declare and initialize myChar */
    int myInt;                                  /* declare but don't initialize myInt */

    printf("%c", myChar);   or  putchar(myChar);  /* output the letter A */
    printf("%c", 'j');      or  putchar('j');     /* output the letter j */
    printf("%c", '\n');     or  putchar('\n');    /* move cursor to next line, column 1 */

    /* Single character input that does not skip leading whitespace */
    scanf("%c", &myChar);              /* input a character using scanf -- myChar must be type char */
    myInt = getchar();                 /* input a character using getchar -- myInt should be type int */

    /* Single character input that does skip leading whitespace */
    scanf("\n%c", &myChar);            /* input a character using scanf -- myChar must be type char */
    while (isspace(myInt = getchar()))  /* input a character using getchar -- myInt should be type int */
        ;
    return(EXIT_SUCCESS);
}
```

%c and %d are Different

%c instructs *printf* to take the value of the corresponding argument, look up the character that the value represents in the ASCII, EBCDIC, etc. character table, then *output that character*.

%d instructs *printf* to *output the value* of the corresponding argument as a decimal integer.

```
printf("%c", 'A');          /* outputs the character whose value is 'A' (which is the letter A) */
printf("%d", 'A');          /* outputs the value of 'A', which is 65 (ASCII character set) */
```

%c instructs *scanf* to input the next character, look up the value of that character in the ASCII, EBCDIC, etc. character table, then store that value in the corresponding argument.

%d instructs *scanf* to keep inputting characters as long as they continue to form a decimal integer, then store the value of that decimal integer in the corresponding argument.

```
scanf("%c", &myChar);   /* if -1234 is entered, the - is read and 45 is stored in myChar */
scanf("%d", &myInt);    /* if -1234 is entered, the value -1234 is read and stored in myInt */
```

NOTE 1.16

Inputting/Outputting Single Characters in C++

The appropriate functions of the C++ *cin* and *cout* objects may be used to input and output individual characters as follows:

```
#include <iostream>                    // standard I/O stream "header" file
using std::cin;                        // declare that cin is in "std" namespace
using std::cout;                       // declare that cout is in "std" namespace

int main()                             // declare function main
{
    char myChar = 'A';                 //declare and initialize myChar
    int myInt;                         //declare but don't initialize myInt

    cout << myChar;                    // output the letter A - myChar must be type char
    cout.put(myChar);                  // output the letter A
    cout << 'j';    or   cout.put('j');    // output the letter j
    cout << '\n';   or   cout.put('\n');   // move cursor to next line, column 1

    /* Single character input that does not skip leading whitespace */
    cin.get(myChar);                   // input a character using get -- myChar must be type char
    myInt = cin.get();                 // input a character using get -- myInt should be type int

    /* Single character input that does skip leading whitespace */
    cin >> myChar;                     // input a character using cin -- myChar must be type char

    return 0;
}
```

NOTE 1.17

A Simple Program Debugging Technique

Most non-trivial programs do not work perfectly the first time. Although IDE's provide built-in easy to learn and use debugging tools, programmers sometimes opt for simply putting *printf* (C) or *cout* (C++) statements at strategic points in the code to keep track of the values of important expressions. In the examples below the appropriate header file inclusions are assumed.

Version 1: The following bug ridden C program is intended to get two values from the user, pass them as arguments to function *Add* (which adds them and returns the result), then output the value *Add* returns.

```
float Add(int x, float y)
{
    return(x + y);                                           /* return sum of arguments */
}
int main(void)
{
    int iInput;
    float fInput, sum;

    printf("Enter an integral and a fractional number: ");   /* prompt user for input */
    scanf("%d %d", &iInput, &fInput);                        /* get user input */
    sum = Add(iInput, fInput);                               /* add the input values */
    printf("The sum is %d\n", sum);                          /* output the sum */
    return 0;
}
```

Version 2: This is the same program with debugging code added. (See notes D.5 & D.6 for __LINE__)

```
float Add(int x, float y)
{
/*B*/printf("Line %d: x = %d, y = %f , x + y = %f \n", __LINE__, x, y, x + y);
    return(x + y);                                           /* return sum of arguments */
}
int main(void)
{
    int iInput;
    float fInput, sum;

    printf("Enter an integral and a fractional number: ");   /* prompt user for input */
    scanf("%d %d", &iInput, &fInput);                        /* get user input */
/*A*/printf("Line %d: iInput = %d, fInput = %f\n", __LINE__, iInput, fInput);
    sum = Add(iInput, fInput);                               /* add the input values */
/*C*/printf("Line %d: sum = %f \n", __LINE__, sum);
    printf("The sum is %d\n", sum);                          /* output the sum */
    return 0;
}
```

Results: Note that before running the debug version the programmer must step through a "paper" listing of the code by hand, writing down the expected values of the important expressions. These are then compared with the values printed by the debugging code when the program is actually run. This comparison exposes the problems listed below. Note that bugs must always be fixed in the order of program execution:

A. Variable *fInput* does not contain the value input by the user;
B. Variable *y* in function *Add* does not contain the value passed to it;
C. Variable *sum* does not contain the value returned by *Add* and the *printf* on the next line is bad.

NOTE 1.18

What is a Namespace? (C++ Only)

In the beginning days of C (way back before running water, indoor plumbing, and electricity) there was the standard library. It came with most C compilers and contained functions and other things to perform many of the common tasks that programmers would want to do, such as printing, file I/O, mathematical computations, etc. As time progressed and programmers demanded more, 3rd party software vendors began to create custom libraries to handle a more sophisticated assortment of tasks such as hi-precision math, specialized I/O, multimedia, graphical user interfaces (GUI), and a multitude of other things. This produced a dilemma in which one vendor might intentionally or inadvertently choose the same name for a function or other entity as that chosen by another vendor. When a programmer using both libraries would attempt to use such a duplicate, which one should be used? The compiler/linker would typically answer this question with a resounding error message, then terminate. This is commonly referred to as a "namespace" problem.

To alleviate this problem C++ introduced the concept of a namespace. This facility lets software vendors and application programmers alike associate a unique identifier, called a namespace, with various entities that could potentially be involved in namespace problems. For example, the contents of the standard C++ library are part of a namespace called ***std***, while a vendor named "Fly-By-Night Software" might choose ***flybynight*** for its namespace. Entities that are part of a namespace may not be used alone but instead, the appropriate namespace must also specified in some way. Assuming that no two vendors choose the same namespace, this mechanism can relegate namespace problems to mere memories.

How do I specify which namespace I want when calling a function, using a class, etc?
There are three ways to do this:

1. The "**scope resolution operator**": *namespaceName::entityName*;
   This technique permits a programmer to individually specify the namespace to which an entity belongs each time it is used. Although verbose, it is the method preferred by namespace proponents and purists.

   Example using *cout* and *cin*, which are both in the *std* namespace:
   ```
   std::cin >> someVariable;
   std::cout << "Hello world\n";
   ```

2. The "**using-declaration**": **using** *namespaceName::entityName*; // NEVER use in a header file!
   This technique permits a programmer to associate one or more entities with a namespace before using them, negating the need to re-specify that namespace each time they are used. This method may be employed when the verbosity of method 1 as well as the wide exposure of method 3 are both undesirable.

   Example using *cout* and *cin*, which are both in the *std* namespace:
   ```
   using std::cin;
   using std::cout;
   cin >> someVariable;
   cout << "Hello world\n";
   ```

3. The "**using-directive**": **using namespace** *namespaceName*; // NEVER use in a header file!
   This technique is typically employed when one namespace will be used much more frequently than any other(s), but is usually not the best choice in programs of significant size. All entities not exploiting either of the two previous techniques will be assumed to be from this namespace.

   Example using *cout* and *cin*, which are both in the *std* namespace:
   ```
   using namespace std;
   cin >> someVariable;
   cout << "Hello world\n";
   ```