2015

C++ Notes

Tassawer (T.H)

[C++ BRIEF INTRODUCTION]

The do-while Loop

Form:	Example:

`do`
`  statement;`
`while (expression);`
	`do`
	`  cout << x++ << endl;`
	`while (x < 100);`

`do`
`{`
`  statement;`
`  statement;`
`} while (expression);`
	`do`
	`{`
	`  cout << x << endl;`
	`  x++;`
	`} while (x < 100);`

Forms of the if Statement

Simple if	Example
`if (expression)` `  statement;`	`if (x < y)` `  x++;`

if/else	Example

`if (expression)`
`  statement;`
`else`
`  statement;`
	`if (x < y)`
	`  x++;`
	`else`
	`  x--;`

if/else if	Example

`if (expression)`
`  statement;`
`else if (expression)`
`  statement;`
`else`
`  statement;`
	`if (x < y)`
	`  x++;`
	`else if (x < z)`
	`  x--;`
	`else`
	`  y++;`

To conditionally-execute more than one statement, enclose the statements in braces:

Form	Example

`if (expression)`
`{`
`  statement;`
`  statement;`
`}`
	`if (x < y)`
	`{`
	`  x++;`
	`  cout << x;`
	`}`

C++ Data Types

Data Type	Description
`char`	Character
`unsigned char`	Unsigned Character
`int`	Integer
`short int`	Short integer
`short`	Same as `short int`
`unsigned short int`	Unsigned short integer
`unsigned short`	Same as `unsigned short int`
`unsigned int`	Unsigned integer
`unsigned`	Same as `unsigned int`
`long int`	Long integer
`long`	Same as `long int`
`unsigned long int`	Unsigned long integer
`unsigned long`	Same as `unsigned long int`
`float`	Single precision floating point
`double`	double precision floating point
`long double`	Long double precision floating point

The while Loop

Form:	Example:
`while (expression)` `  statement;`	`while (x < 100)` `  cout << x++ << endl;`

`while (expression)`
`{`
`  statement;`
`  statement;`
`}`
	`while (x < 100)`
	`{`
	`  cout << x << endl;`
	`  x++;`
	`}`

Commonly Used Operators

Assignment Operators

=	Assignment
+=	Combined addition/assignment
-=	Combined subtraction/assignment
*=	Combined multiplication/assignment
/=	Combined division/assignment
%=	Combined modulus/assignment

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators

&&	AND
||	OR
!	NOT

Increment/Decrement

++	Increment
--	Decrement

Conditional Operator ?:

Form:

expression ? expression : expression

Example:

```
x = a < b ? a : b;
```

The statement above works like:

```
if (a < b)
   x = a;
else
   x = b;
```


Tassawer Hussain

Mcsf13m016

PUCIT

Language	Description	CH # 01
BASIC	Beginners All-purpose Symbolic Instruction Code. A general programming language originally designed to be simple enough for beginners to learn.	
FORTRAN	Formula Translator. A language designed for programming complex mathematical algorithms.	
COBOL	Common Business-Oriented Language. A language designed for business applications.	
Pascal	A structured, general-purpose language designed primarily for teaching programming.	
C	A structured, general-purpose language developed at Bell Laboratories. C offers both high-level and low-level features.	
C++	Based on the C language, C++ offers object-oriented features not found in C. Also invented at Bell Laboratories.	
C#	Pronounced "C sharp." A language invented by Microsoft for developing applications based on the Microsoft .NET platform.	
Java	An object-oriented language invented at Sun Microsystems. Java may be used to develop programs that run over the Internet, in a Web browser.	
JavaScript	JavaScript can be used to write small programs that run in Web pages. Despite its name, JavaScript is not related to Java.	
Python	Python is a general purpose language created in the early 1990s. It has become popular in both business and academic applications.	
Ruby	Ruby is a general purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on Web servers.	
Visual Basic	A Microsoft programming language and software development environment that allows programmers to quickly create Windows-based applications.	

What Is a Program Made of?

Language Element	Description
Key Words	Words that have a special meaning. Key words may only be used for their intended purpose. Key words are also known as reserved words.
Programmer-Defined Identifiers	Words or names defined by the programmer. They are symbolic names that refer to variables or programming routines.
Operators	Operators perform operations on one or more operands. An operand is usually a piece of data, like a number.
Punctuation	Punctuation characters that mark the beginning or ending of a statement, or 7separate items in a list.
Syntax	Rules that must be followed when constructing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

Special Characters — CH # 02

Character	Name	Description
//	Double slash	Marks the beginning of a comment.
#	Pound sign	Marks the beginning of a preprocessor directive.
< >	Opening and closing brackets	Encloses a filename when used with the #include directive.
( )	Opening and closing parentheses	Used in naming a function, as in int main()
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
" "	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
;	Semicolon	Marks the end of a complete programming statement.

Common Escape Sequences

Escape Sequence	Name	Description
\n	Newline	Causes the cursor to go to the next line for subsequent printing.
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop.
\a	Alarm	Causes the computer to beep.
\b	Backspace	Causes the cursor to back up, or move left one position.
\r	Return	Causes the cursor to go to the beginning of the current line, not the next line.
\\	Backslash	Causes a backslash to be printed.
\'	Single quote	Causes a single quotation mark to be printed.
\"	Double quote	Causes a double quotation mark to be printed.

The C++ Key Words

and	continue	goto	public	try
and_eq	default	if	register	typedef
asm	delete	inline	reinterpret_cast	typeid
auto	do	int	return	typename
bitand	double	long	short	union
bitor	dynamic_cast	mutable	signed	unsigned
bool	else	namespace	sizeof	using
break	enum	new	static	virtual
case	explicit	not	static_cast	void
catch	export	not_eq	struct	volatile
char	extern	operator	switch	wchar_t
class	false	or	template	while
compl	float	or_eq	this	xor
const	for	private	throw	xor_eq
const_cast	friend	protected	true	

Legal Identifiers

some specific rules that must be followed with all identifiers.

- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means ItemsOrdered is not the same as itemsordered.

Integer Data Types

Integer variables can only hold whole numbers.

Data Type	Size	Range
short	2 bytes	–32,768 to +32,767
unsigned short	2 bytes	0 to +65,535
int	4 bytes	–2,147,483,648 to +2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
long	4 bytes	–2,147,483,648 to +2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

```
floors = 15;        rooms = 300;        suites = 30;
```

Each of these lines contains an integer literal. In C++, integer literals are normally stored in memory just as an int. On a system that uses 2 byte integers and 4 byte longs, the literal 50000 is too large to be stored as an int, so it is stored as a long.

if you are in a situation where you have an integer literal, but you need it to be stored in memory as a long integer? (Rest assured, this is a situation that does arise.) C++ allows you to force an integer literal to be stored as a long integer by placing the letter L at the end of the number. Here is an example:

32L

On a computer that uses 2-byte integers and 4-byte long integers, this literal will use 4 bytes. This is called a long integer literal.

By default, C++ assumes that all integer literals are expressed in decimal. You express hexadecimal numbers by placing 0x in front of them. (This is zero-x, not oh-x.) Here is how the hexadecimal number F4 would be expressed in C++: 0xF4

Octal numbers must be preceded by a 0 (zero, not oh). For example, the octal 31 would be written as 031

The char Data Type

As its name suggests, it is primarily for storing characters, but strictly speaking, it is an integer data type. The reason an integer data type is used to store characters is because characters are internally represented by numbers.

Strings that are stored in memory, with the null terminator appended to their end, are called *C-strings*.

Null terminator is represented by the \0 character.

A' is stored as | A |

"A" is stored as | A | \0 |

```
char   letter;
letter = 'A';     // This will work.
letter = "A";     // This will not work!
```

Floating-Point Data Types

Floating-point data types are used to define variables that can hold real numbers.

Data Type	Key Word	Description
Single precision	float	4 bytes. Numbers between ±3.4E-38 and ±3.4E38
Double precision	double	8 bytes. Numbers between ±1.7E-308 and ±1.7E308
Long double precision	long double*	8 bytes. Numbers between ±1.7E-308 and ±1.7E308

*Some compilers use 10 bytes for long doubles. This allows a range of ±3.4E-4932 to ±1.1E4832

You will notice there are no unsigned floating point data types. On all machines, variables of the float, double, and long double data types can store positive or negative numbers.

Floating-point literals are normally stored in memory as doubles. Just in case you need to force a literal to be stored as a float, you can append the letter F or f to the end of it. For example, 1.2F, 45.907f

If you want to force a value to be stored as a long double, append an L or l to it, as in the following examples:

1034.56L 89.2l

The bool Data Type

Boolean variables are set to either true **or** false**.**

True is represented in memory by the number 1, and false is represented by 0.

Determining the Size of a Data Type

Syntax: sizeof(datatype)

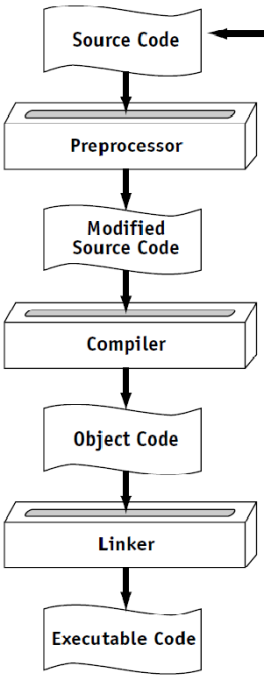
```
cout << "The size of an integer is " << sizeof(int);
```

An Older Style Program

```
#include <iostream.h>
int main()
{
        cout << "Programming is great fun!";
        return 0;        }
```

Let's review some important points regarding characters and strings:

- Printable characters are internally represented by numeric codes. Most computers use ASCII codes for this purpose.
- Characters normally occupy a single byte of memory.
- Strings are consecutive sequences of characters that occupy consecutive bytes of memory.
- C-strings always have a null terminator at the end. This marks the end of the string.
- Character literals are enclosed in single quotation marks.
- String literals are enclosed in double quotation marks.
- Escape sequences are stored internally as a single character.



Source code is entered with a text editor by the programmer.

```
#include <iostream>
using namespace std;

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

Precedence of Arithmetic Operators

(unary negation) -
* / %
+ -

(Highest to Lowest)

Fundamental Arithmetic Operators

Operator	Meaning	Type	Example
+	Addition	Binary	`total = cost + tax;`
–	Subtraction	Binary	`cost = total - tax;`
*	Multiplicatio	Binary	`tax = cost * rate;`
/	Division	Binary	`salePrice = original / 2;`
%	Modulus	Binary	`remainder = value % 3;`

Associativity of Arithmetic Operators

Operator	Associativity
(unary negation) -	Right to left
* / %	Left to right
+ -	Left to right

Power Function:

The program must include the cmath header file.

#include<cmath>

pow(base, exp);

Think of pow as a "black box" that you plug two numbers into, and that then sends a third number out. The number that comes out has the value of the first number raised to the power of the second number,

Data Type Ranking

```
long double
double
float
unsigned long
long
unsigned int
int
```

Type Coercion:

When C++ is working with an operator, it strives to convert the operands to the same type. This automatic conversion is known as *type coercion.* When a value is converted to a higher data type, it is said to be *promoted.* To *demote* a value means to convert it to a lower data type.

Rule 1: chars, shorts, and unsigned shorts are automatically promoted to int.

Rule 2: When an operator works with two values of different data types, the lower ranking value is promoted to the type of the higher-ranking value.

Rule 3: When the final value of an expression is assigned to a variable, it will be converted to the data type of that variable.

OverFlow & UnderFlow:

When a variable is assigned a value that is too large or too small in range for that variable's data type, the variable overflows or underflows.

e.g.

```
short     testVar  = 32767;
cout<< testVar << endl;

testVar = testVar + 1;
cout<< testVar << endl;

testVar = testVar - 1;
cout<< testVar << endl;
```

Output

```
32767
-32768
32767
```

Typically, when an integer overflows, its contents wrap around to that data type's lowest possible value.

Type Casting: Type casting allows you to perform manual data type conversion.

static_cast<*DataType*>(*Value*)

Syntax:

```
double number = 3.7;
int val;
val = static_cast <int> (number);
```

C++ also supports two older methods of creating type cast expressions: the C-style form and the prestandard C++ form. The C-style cast is the name of a data type enclosed in parentheses, preceding the value that is to be converted.

```
val = (int)number;                    perMonth = (double)books / months;
```

The prestandard C++ form of the type cast expression appears as a data type name followed by a value inside a set of parentheses. Example: The type cast in this statement returns a copy of the value in number, converted to an int.

```
val = int(number);                    perMonth = double(books) / months;
```

Named Constants:

A *named constant* is like a variable, but its content is read-only, and cannot be changed while the program is running. Here is a definition of a named constant:

```
const double INTEREST_RATE = 0.129;
```

The name of the variable is written in all uppercase characters. Its value will remain constant throughout the program's execution. An initialization value must be given when defining a variable with the const qualifier, or an error will result when the program is compiled. The key word const is a qualifier that tells the compiler to make the variable read-only.

It is important to realize the difference between const variables and constants created with the #define directive. Be careful not to put a semicolon at the end of a #define directive.

```
#define PI 3.14159
area = PI * pow(radius, 2.0);
```

```
#include <iostream>
using namespace std;
#define GREETING1 "This program calculates the number "
#define GREETING2 "of candy pieces sold."
#define QUESTION "How many jars of candy have you sold? "
#define RESULTS "The number of pieces sold: "
#define YOUR_COMMISSION "Candy pieces you get for commission: "
#define COMMISSION_RATE .20
Int main()
{
Const int PER_JAR = 1860;
int jars, pieces;
double commission;
cout << GREETING1;
cout << GREETING2 << endl;
cout << QUESTION;
cin >> jars;
pieces = jars * PER_JAR;
cout << RESULTS << pieces << endl;
commission = pieces * COMMISSION_RATE;
cout << YOUR_COMMISSION << commission << endl;
return 0;
}
```

Multiple Assignment and Combined Assignment :

Multiple assignment means to assign the same value to several variables with one statement.

```
a = b = c = d = 12;
```

The assignment operator works from right to left. 12 is first assigned to d, then to c, then to b, then to a.

Operator	Example	Equivalent to
+=	x += 5;	x = x + 5;
-=	y -= 2;	y = y - 2;
*=	z *= 10;	z = z * 10;
/=	a /= b;	a = a / b;
%=	c %= 3;	c = c % 3;

Combined assignment operators, also known as *compound operators,* and *arithmetic assignment operators.*

When constructing such statements, you must realize the precedence of the combined assignment operators is lower than that of the regular math operators.

```
result *= a + 5;     =>> result = result * (a + 5);
```

Formatting Output: #include <iomanip> // Required for setw

The cout object provides ways to format data as it is being displayed. This affects the way data appears on the screen. The way a value is printed is called its *formatting.*

Cout offers a way of specifying the minimum number of spaces to use for each number. A stream manipulator, **setw**, can be used to establish print fields of a specified width.

```
value = 23;                cout << setw(5) << value;
```

The no. inside the parentheses after the word setw specifies the *field width* for the value immediately following it.

```
value = 23;
cout << "(" << setw(5) << value << ")";
```

This will cause the following output: (23)

The no. appears on the right side of the field with blank spaces "padding" it in front, it is said to be *right-justified.*

- The field width of a floating-point number includes a position for the decimal point.
- The field width of a string includes all characters in the string, including spaces.
- The values printed in the field are right-justified by default.

The setprecision Manipulator

Floating-point values may be rounded to a number of *significant digits,* or *precision,* which is the total number of digits that appear before and after the decimal point.

```
double quotient, number1 = 132.364, number2 = 26.91;
quotient = number1 / number2;
cout << quotient << endl;                                  4.91877
cout << setprecision(5) << quotient << endl;               4.9188
cout << setprecision(4) << quotient << endl;               4.919
cout << setprecision(3) << quotient << endl;               4.92
cout << setprecision(2) << quotient << endl;               4.9
cout << setprecision(1) << quotient << endl;               5
```

*By default, the system in the illustration displays floating-point values with 6 significant digits.

The setprecision manipulator can sometimes surprise you in an undesirable way. When the precision of a number is set to a lower value, numbers tend to be printed in scientific notation.

The fixed Manipulator

Stream manipulator, fixed, forces cout to print the digits in *fixed-point notation,* or decimal.

When the fixed and setprecision manipulators are used together, the value specified by the setprecision manipulator will be the number of digits to appear after the decimal point, not the number of significant digits.

```
double x = 123.4567;
cout << setprecision(2) << fixed << x << endl;          output: 123.46
```

The showpoint Manipulator

By default, floating-point numbers are not displayed with trailing zeroes, and floatingpoint numbers that do not have a fractional part are not displayed with a decimal point.

```
double x = 123.4, y = 456.0;                            output:
cout << setprecision(6) << x << endl;                       123.4
cout << y << endl;                                          456
```

If we want the numbers padded with trailing zeroes, we must use the showpoint manipulator as shown in the following code.

```
double x = 123.4, y = 456.0;                            output:
cout << setprecision(6) << showpoint << x << endl;          123.400
cout << y << endl;                                          456.000
```

The *left* and *right* Manipulators

Normally output is right justified.

```
double x = 146.789,    y = 24.2,    z = 1.783;          output:
cout << setw(10) << x << endl;                              146.789
cout << setw(10) << y << endl;                                24.2
cout << setw(10) << z << endl;                               1.783
```

We can cause the values to be left-justified by using the left manipulator,

```
double x = 146.789,    y = 24.2,    z = 1.783;          output:
cout << setw(10) << x << endl;                              146.789
cout << setw(10) << y << endl;                               24.2
cout << setw(10) << z << endl;                               1.783
```

Data Type Ranking

```
long double
double
float
unsigned long
long
unsigned int
int
```

Stream Manipulator	Description
`setw(n)`	Establishes a print field of *n* spaces.
`fixed`	Displays floating-point numbers in fixed point notation.
`showpoint`	Causes a decimal point and trailing zeroes to be displayed, even if there is no fractional part.
`setprecision(n)`	Sets the precision of floating-point numbers.
`left`	Causes subsequent output to be left justified.
`right`	Causes subsequent output to be right justified.

Formatted Input:

The cin object provides ways of controlling string and character input.

If the user types more characters than the array will hold, cin will store the string in the array anyway, overwriting whatever is in memory next to the array. This type of error is known as a *buffer overrun*, Here is a statement defining an array of 10 characters and a cin statement reading no more characters than the array will hold:

```
char word[10];
cin >> setw(10) >> word;
```

There are two important points to remember about the way cin handles field widths:

- The field width only pertains to the very next item entered by the user.
- cin stops reading input when it encounters a *whitespace* character that include the **[Enter]** key, space, and tab.

Reading a "Line" of Input

cin provides a member function to read a string containing spaces. The function is called ***getline***, and its purpose is to read an entire "line" of text, until the **[Enter]** key is pressed.

```
cin.getline(sentence, 20);
```

The getline function takes two arguments separated by a comma. The first argument isthe name of the array that the string is to be stored in. The second argument is the size of the array. cin will read up to one character less than this number, leaving room for the null terminator.

Reading a Character

Sometimes we want to read only a single character of input. For example, a menu driven program. The simplest way to read a single character is with the >> operator.

Using cin.get

A limiting characteristic of the >> operator with char variables is that it requires a character to be entered and it ignores all leading whitespace characters. (The **[Enter]** key must still be pressed after the character has been typed.) Programs that ask the user to "press the enter key to continue" cannot use the >> operator to read only the pressing of the **[Enter]** key. In those situations another of cin's member functions, **get**, becomes useful. The get function reads a single character including any whitespace character. If the user types the character A and presses Enter, the cin.get function will store the character 'A' in the variable ch. If the user simply presses the Enter, the cin.get function will store the newline character ('\n') in the variable ch.

```
Char  ch;
cout  <<  "This program has paused. Press Enter to continue.";
cin.get(ch);
cout << "Thank you!" << endl;
return 0;
```

The only difference between the get function and the >> operator is that get reads the first character typed, even if it is a space, tab, or the **[Enter]** key.

Mixing cin >> and cin.get

```
char   ch;    // Define a character variable.
int   number;        // Define an integer variable.
Cout  <<   "Enter a number: ";
cin   >>    number; // Read an integer.
Cout   <<   "Enter a character: ";
cin.get(ch);          // Read a character.
cout   <<   "Thank You!\n";
```

These statements may allow the user to enter a number, but not a character. It will appear that the cin.get statement is skipped. This happens because both cin >> and cin.get read the user's keystrokes from the keyboard buffer. After the user enters a number, in response to the first prompt, he or she presses the Enter key. Pressing the Enter key causes a newline character ('\n') to be stored in the keyboard buffer.

Using cin.ignore

To solve the problem previously described, the cin.ignore member function can be used. cin.ignore tells the cin object to skip characters in the keyboard buffer.

cin.ignore(*n*, *c*);

The arguments shown in the parentheses are optional. If they are used, *n* is an integer and *c* is a character. They tell cin to skip *n* number of characters, or until the character *c* is encountered. For example, the following statement causes cin to skip the next 20 characters, or until a newline is encountered, whichever comes first:

```
cin.ignore(20, '\n');
```

If no arguments are used, cin will only skip the very next character.

```
cin.ignore();
```

Member Function

A member function is a procedure, written in C++ code, that is part of an object. A member function causes the object it is a member of to perform an action.

Objects are programming elements containing both data and procedures that operate on the data. The packaging together of data and the data's related procedures within an object is known as ***encapsulation***. In C++, the procedures that are part of an object are known as ***member functions***. Calling an object's member function causes the object to perform some operation.

In this chapter you have used the following member functions of the cin object:

- getline
- get
- ignore

More Mathematical Library Functions

The C++ runtime library provides several functions for performing complex mathematical operations.

Table shows several of these, each of which requires the **cmath** header file.

Function	Example	Description
`abs`	`y = abs(x);`	Returns the absolute value of the argument. The argument and the return value are integers.
`cos`	`y = cos(x);`	Returns the cosine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s.
`exp`	`y = exp(x);`	Computes the exponential function of the argument, which is `x`. The return type and the argument are `double`s.
`fmod`	`y = fmod(x, z);`	Returns, as a `double`, the remainder of the first argument divided by the second argument. Works like the modulus operator, but the arguments are `double`s. (The modulus operator only works with integers.) Take care not to pass zero as the second argument. Doing so would cause division by zero.
`log`	`y = log(x);`	Returns the natural logarithm of the argument. The return type and the argument are `double`s.
`log10`	`y = log10(x);`	Returns the base-10 logarithm of the argument. The return type and the argument are `double`s.
`sin`	`y = sin(x);`	Returns the sine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s.
`sqrt`	`y = sqrt(x);`	Returns the square root of the argument. The return type and argument are `double`s.
`tan`	`y = tan(x);`	Returns the tangent of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s.

Random Numbers

#include <cstdlib> Header File

Some programming techniques require the use of randomly generated numbers. The C++ library has a function, **rand(),** for this purpose. The number returned by the function is an int. Here is an example of its usage: y = rand();

After this statement executes, the variable y will contain a random number. In actuality, the numbers produced by **rand()** are pseudorandom. The function uses an algorithm that produces the same sequence of numbers each time the program is repeated on the same system. For example, suppose the following statements are executed.

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

The three numbers displayed will appear to be random, but each time the program runs, the same three values will be generated. In order to randomize the results of **rand()**, the **srand()** function must be used. **srand()** accepts an unsigned int argument, which acts as a seed value for the algorithm. By specifying different seed values, rand() will generate different sequences of random numbers.

A common practice for getting unique seed values is to call the time function, which is part of the standard library. The time function returns the number of seconds that have elapsed since **midnight, January 1, 1970**. The **time function requires the ctime header** file, and you pass 0 as an argument to the function.

```
#include   <iostream>                  //  This program demonstrates random numbers.
#include   <cstdlib>      // For rand and srand
#include   <ctime>        // For the time function
Using   namespace   std;
Int   main()
{       // Get the system time.
      unsigned   seed  =  time(0);
      // Seed the random number generator.
     srand(seed);
     // Display three random numbers.
     Cout  <<  rand()  <<  endl;
     Cout  <<  rand()  <<  endl;
     Cout  <<  rand()  <<  endl;
     return  0;       }
```

Program Output

```
23861
20884
21941
```

Introduction to File Input and Output

Data is read from files in a sequential manner.

There are always three steps that must be taken when a file is used by a program:

1. The file must be *opened*. If the file does not yet exist, opening it means creating it.
2. Data is then saved to the file, read from the file, or both.
3. When the program is finished using the file, the file must be *closed*.

Setting Up a Program for File Input/Output

#include <fstream>

1. Include Header File.
2. Create File Stream Object
   a. Ofstream fout; // for output to the file
   b. Ifstream fin; // for input from the file
3. Open the File.
   a. fout.open("File name with Extension / Path")
4. Read / Write into the file
   a. fin >> variable name / Literals;
   b. fout << variable name / Literals;
5. Close the File
   a. fout.close();

File Stream Data Type	Description
`ofstream`	Output file stream. This data type can be used to create files and write data to them. With the `ofstream` data type, data may only be copied from variables to the file, but not vice versa.
`ifstream`	Input file stream. This data type can be used to open existing files and read data from them into memory. With the `ifstream` data type, data may only be copied from the file into variables, not but vice versa.
`fstream`	File stream. This data type can be used to create files, write data to them, and read data from them. With the `fstream` data type, data may be copied from variables into a file, or from a file into variables.

R. Operators	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Relational Operators CH # 04

Relational operators allow you to compare numeric and char values and determine whether one is greater than, less than, equal to, or not equal to another. Numeric data is compared in C++ by using relational operators. All the relational operators have left-to-right associativity. Relational expressions are also known as *Boolean expressions*, which means their value can only be *true* or *false*. In C++, relational expressions represent true states with the number 1 and false states with the number 0. Relational expressions have a higher precedence than the assignment operator.

The if-else Statement

```
if (expression)
        statement or block
else
        statement or block
```

The if statement can cause other statements to execute only under certain conditions. If we inadvertently put a semicolon after the if part, the compiler will assume we are placing a null statement there. The *null statement* is an empty statement that does nothing. **Comparing Floating-Point Numbers** rounding errors sometimes occur. This is because some fractional numbers cannot be exactly represented using binary. So, you should be careful when using the equality operator (==) to compare floating point numbers.To test more than one condition, an if statement can be nested inside another if statement.

The if/else if Statement

The if/else if statement tests a series of conditions. It is often simpler to test a series of conditions with the if/else if statement than with a set of nested if/else statements. The last else clause, which does not have an if statement following it, is referred to as the ***trailing else***. The trailing else is optional, but in many situations you will use it to catch errors.

```
if (expression_1)
{
    statement
    statement          } If expression_1 is true these statements are executed, and the rest of the structure is ignored.
    etc.
}
else if (expression_2)
{
    statement
    statement          } Otherwise, if expression_2 is true these statements are executed, and the rest of the structure is ignored.
    etc.
}
Insert as many else if clauses as necessary

else
{
    statement
    statement          } These statements are executed if none of the expressions above are true.
    etc.
}
```

Flags

A flag is a Boolean or integer variable that signals when a condition exists. A *flag* is typically a bool variable that signals when some condition exists in the program. When the flag variable is set to false, it indicates that the condition does not yet exist. When the flag variable is set to true, it means the condition does exist.

```
bool highScore = false;
if (average > 95)
        highScore = true;
if (highScore)
        cout << "Congratulations! That's a high score!";
```

Integer Flags Integer variables may also be used as flags. This is because in C++ the value 0 is considered false, and any nonzero value is considered true.

```
int highScore = 0;          // 0 means false.
if (average > 95)
        highScore = 1;      // A nonzero value means true.
if (highScore)
        cout << "Congratulations! That's a high score!";.
```

Menus

You can use nested if/else statements or the if/else if statement to create menu-driven programs. A *menu-driven* program allows the user to determine the course of action by selecting it from a list of actions.

Logical Operators

Logical operators connect two or more relational expressions into one or reverse the logic of an expression.

If the sub-expression on the left side of an && operator is false, the expression on the right side will not be checked. Since the entire expression is false if only one of the subexpressions is false, it would waste CPU time to check the remaining expression. This is called ***short circuit evaluation.*** The logical operators have left-to-right associativity.

Operator	Meaning	Effect
&&	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
||	OR	Connects two expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which.
!	NOT	The ! operator reverses the "truth" of an expression. It makes a true expression false, and a false expression true.

Precedence
!
& &
| |

Logical operators are effective for determining whether a number is in or out of a range. When determining whether a number is inside a numeric range, it's best to use the && operator. For example,

```
if ( x >= 20 && x <= 40 )
        cout << x << " is in the acceptable range.\n ";
```

When determining whether a number is outside a range, the || operator is best to use.

```
If ( x < 20 || x > 40 )
        Cout << x << " is outside the acceptable range.\n ";
```

Input validation is the process of inspecting data given to a program by the user and determining if it is valid.

Scope of a variable

The scope of a variable is limited to the block in which it is defined. It is a common practice to define all of a function's variables at the top of the function. Sometimes, especially in longer programs, it's a good idea to define variables near the part of the program where they are used. Variables defined inside a set of braces have *local scope* or *block scope*.

Variables with the Same Name

When a block is nested inside another block, a variable defined in the inner block may have the same name as a variable defined in the outer block. As long as the variable in the inner block is visible, however, the variable in the outer block will be hidden

Comparing Strings

#include <iostream> Headerfile

You must use the strcmp library function to compare C-strings. In C++, C-string comparisons are done with the library function **strcmp**. The function compares the contents of string1 with the contents of string2 and returns one of the following values:

- If the two strings are identical, strcmp returns 0.
- If string1 < string2, strcmp returns a negative number.
- If string1 > string2, strcmp returns a positive number.

The function strcmp is case-sensitive when it compares the two strings.

```
Strcmp ( string1 , string2 );
```

```
if ( strcmp ( firstString, secondString ) == 0 )
if ( ! strcmp ( firstString, secondString ) )
```

```
if ( strcmp ( string1, string2 ) == 0 )
        statement; // The strings are the same
else
        statement; // The strings are NOT the same
```

```
cin.width ( SIZE ) ;            // Restrict input for code safety.
Cin >> variableName;
```

The Conditional Operator

You can use the conditional operator to create short expressions that work like if/else statements. The conditional operator is powerful and unique. It provides a shorthand method of expressing a simple if/else statement. The operator consists of the question-mark (?) and the colon(:).

The part of the conditional expression that comes before the question mark is the expression to be tested. It's like the expression in the parentheses of an if statement. If the expression is true, the part of the statement between the ? and the : is executed. Otherwise, the part after the : is executed.

```
expression ? expression : expression;
```

```
x < 0 ? y = 10 : z = 20;    if (x < 0)
                                y = 10;
                            else
                                z = 20;
```

```
a = x > 100 ? 0 : 1;        if (x > 100)
                                a = 0;
                            else
                                a = 1;
```

```
Cout << "Your grade is: " << ( score < 60 ? "Fail." : "Pass.");

if (score < 60)
    cout << "Your grade is: Fail.";
else
    cout << "Your grade is: Pass.";
```

The parentheses are placed around the conditional expression because the<< operator has higher precedence than the ? : operator.

```
switch (IntegerExpression)
{
   case ConstantExpression:
        // place one or more
        // statements here

   case ConstantExpression:
        // place one or more
        // statements here

   // case statements may be repeated as many
   // times as necessary

   default:
        // place one or more
        // statements here
}
```

The switch Statement

The switch statement lets the value of a variable or expression determine where the program will branch. The expression following the word case must be an integer literal or constant. It cannot be a variable, and it cannot be an expression such as x < 22 or n == 50.

```
char choice;      cout << "Enter A, B, or C: ";    cin >> choice;
switch ( choice ) {
case 'A': cout << "You entered A.\n";   break;
case 'B': cout << "You entered B.\n";   break;
case 'C': cout << "You entered C.\n";   break;
default:  cout << "You did not enter A, B, or C! \n";   }
```

Testing for File Open Errors

When opening a file you can test the file stream object to determine if an error occurred.

```
ifstream   fin;
fin.open ( " customers.txt " );
if  ( ! fin)
        cout  << " Error opening file. \n" ;
```

Another way to detect a failed attempt to open a file is with the fail member function, as shown in the following code:

```
ifstream   fin ;
fin.open ( " customers.txt " ) ;
if ( fin.fail() )
        cout  <<  " Error opening file. \n " ;
```

The fail member function returns true when an attempted file operation is unsuccessful.

```
ofstream   fout ;
fout.open ( " customer.txt " ) ;
if  ( fout.fail() )
{
        cout  <<  "  The customer.txt file could not be opened. \n " ;
        cout  <<  "  Perhaps the disk is full or you do not have \n " ;
        cout  <<  "  sufficient privileges. Contact your system  \n " ;
        cout  <<  "  manager for assistance.  \n " ;
}
```

The Increment and Decrement Operators CH # 05

++ and – -- (unary operators) are operators that add and subtract 1 from their operands. To *increment* a value means to increase it by one, and to ***decrement*** a value means to decrease it by one.

```
num  =  num  +  1 ;
num  +=  1 ;
num++ ;  (Postfix mode)
++num ;  (Prefix mode )
```

```
num  =  num  -  1 ;
num  -=  1 ;
num -- ;    (Postfix mode)
--num ;     (Prefix mode )
```

Introduction to Loops:

A loop is part of a program that repeats. A *loop* is a control structure that causes a statement or group of statements to repeat. C++ has three looping control structures: the while loop, the do-while loop, and the for loop.

```
while (expression)
{
statement;
// Place as many statements
// here as necessary.
}
```

```
int number = 1;
while (number <= 5)
{
cout << "Hello\n";
number++;
}
```

The while Loop has two important parts: (1) an expression that is tested for a true or false value, (**Header**) and (2) a statement or block that is repeated as long as the expression is true (**Body**). Each repetition of a loop is known as an ***iteration***. The while loop is known as a ***pretest*** loop, which means it tests its expression before each iteration. ***Input validation*** is the process of inspecting data given to a program by the user and determining if it is valid. **The while loop is especially useful for validating input**. The read operation that takes place just before the loop is called a ***priming read***.

A **counter** is a variable that is regularly incremented or decremented each time a loop iterates.

The do-while Loop

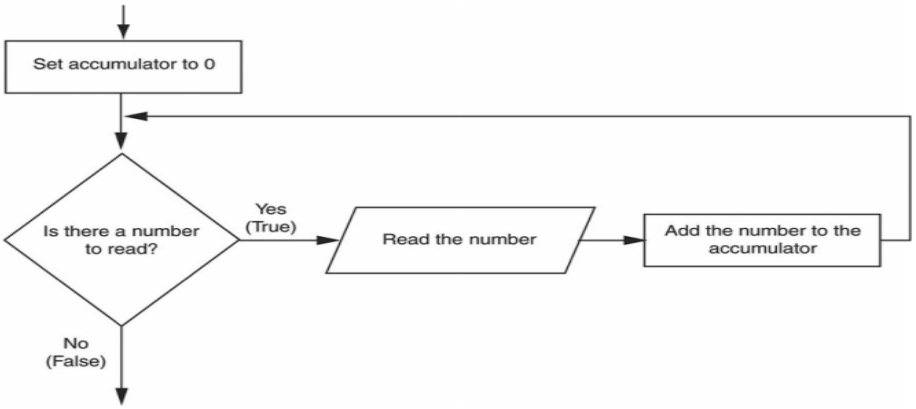
The do-while loop is a posttest loop, which means its expression is tested after each iteration. When program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user if he or she wants to repeat the process. This type of loop is known as a *user-controlled loop*, because it allows the user to decide the number of iterations. **Menu Driven Loop**

```
do
{
statement;
statement;
// Place as many statements here
// as necessary.
} while (expression);
```

The for Loop

The for loop is ideal for performing a **known number of iterations**. In general, there are **two** categories of loops: conditional loops and count-controlled loops. A ***conditional loop*** executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate. Sometimes you know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a ***count-controlled loop***.

```
for (initialization; test; update)
{
statement;
statement;
// Place as many statements here
// as necessary.
}
```



☺

Keeping a Running Total

A *running total* is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator*.

Sentinels

A *sentinel* is a special value that marks the end of a list of values. A *sentinel* is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.

```
#include <iostream>
using namespace std;
int main()
{
   int game = 1,    // Game counter
       points,      // To hold a number of points
       total = 0;   // Accumulator
   cout << "Enter the number of points your team has earned\n";
   cout << "so far in the season, then enter -1 when finished.\n\n";
   cout << "Enter the points for game " << game << ": ";
   cin >> points;
   while (points != -1)
   {
      total += points;
      game++;
      cout << "Enter the points for game " << game << ": ";
      cin >> points;
   }
   cout << "\nThe total points are " << total << endl;
   return 0;
}
```

Using a Loop to Read Data from a File

When reading a value from a file with the stream extraction operator (>>), the operator returns a true or false value indicating whether the value was successfully read. This return value can be used to detect when the end of a file has been reached. If the operator returns true, then a value was successfully read. If the operator returns false, it means that no value was read from the file.

```
if (inputFile >> number)
    cout << "The data read from the file is " << number << endl; // Data was successfully read from the file.
else
    cout << "Could not read an item from the file.\n"; // No data was read from the file.
```

Notice that the statement that reads an item from the file is also used as the conditional expression in the if statement:

```
if (inputFile >> number)
```

This statement does two things:

1. It uses the expression inputFile >> number to read an item from the file and stores the item in the number variable. The >> operator returns true if the item was successfully read, or false otherwise.
2. It tests the value returned by the stream extraction operator.

You can use the stream extraction operator's return value in a loop to determine when the end of the file has been reached. Here is an example:

```
while (inputFile >> number)
    cout << number << endl;
```

```
// This program displays all of the numbers in a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream inputFile;  // File stream object
   int number;          // To hold a value from the file

   inputFile.open("numbers.txt");     // Open the file.
   if (!inputFile)                    // Test for errors.
      cout << "Error opening file.\n";
   else
   {
      while (inputFile >> number)     // Read a number
      {
         cout << number << endl;      // Display the number.
      }
      inputFile.close();              // Close the file.
   }
   return 0;
}
```

```
// This program displays five numbers in a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream inputFile;  // File stream object
   int number;          // To hold a value from the file
   int count = 1;       // Loop counter, initialized with 1

   inputFile.open("numbers.txt");    // Open the file.
   if (!inputFile)                   // Test for errors.
      cout << "Error opening file.\n";
   else
   {
      while (count <= 5)
      {
         inputFile >> number;        // Read a number.
         cout << number << endl;     // Display the number.
         count++;                    // Increment the counter.
      }
      inputFile.close();             // Close the file.
   }
   return 0;
}
```

Nested Loop A loop that is inside another loop is called a *nested loop*. A few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

Cout.fill (' 0 ')
The fill member function of cout changes the fill character, which is a space by default. In the program segment above, the fill function causes a zero to be printed in front of all single digit numbers.

Breaking Out of a Loop

The ***break*** statement causes a loop to terminate early. In a nested loop, the break statement only interrupts the loop it is placed in.

The *continue* Statement

The continue statement causes a loop to stop its current iteration and begin the next one. When continue is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration.

```
// This program calculates the charges for DVD rentals.
// Every third DVD is free.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int dvdCount = 1;     // DVD counter
   int numDVDs;          // Number of DVDs rented
   double total = 0.0;   // Accumulator
   char current;         // Current release, Y or N

   // Get the number of DVDs.
   cout << "How many DVDs are being rented? ";
   cin >> numDVDs;

   // Determine the charges.
   do
   {
      if ((dvdCount % 3) == 0)
      {
         cout << "DVD #" << dvdCount << " is free!\n";
         continue;  // Immediately start the next iteration
      }
      cout << "Is DVD #" << dvdCount;
      cout << " a current release? (Y/N) ";
      cin >> current;
      if (current == 'Y' || current == 'y')
         total += 3.50;
      else
         total += 2.50;
   } while (dvdCount++ < numDVDs);

   // Display the total.
   cout << fixed << showpoint << setprecision(2);
   cout << "The total is $" << total << endl;
   return 0;
}
```

```
cout << fixed << right;
cout.fill('0');
for (int hours = 0; hours < 24; hours++)
{
   for (int minutes = 0; minutes < 60; minutes++)
   {
      for (int seconds = 0; seconds < 60; seconds++)
      {
         cout << setw(2) << hours << ":";
         cout << setw(2) << minutes << ":";
         cout << setw(2) << seconds << endl;
      }
   }
}
```

The output of the previous program segment follows:

```
00:00:00
00:00:01
00:00:02
   .   (The program will count through each second of 24 hours.)
   .
   .
23:59:59
```

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double value;
   char choice;
   cout << "Enter a number: ";
   cin >> value;
   cout << "This program will raise " << value;
   cout << " to the powers of 0 through 10.\n";
   for (int count = 0; count <= 10; count++)
   {
      cout << value << " raised to the power of ";
      cout << count << " is " << pow(value, count);
      cout << "\nEnter Q to quit or any other key ";
      cout << "to continue. ";
      cin >> choice;
      if (choice == 'Q' || choice == 'q')
         break;
   }
   return 0;
}
```

Modular Programming

CH # 06

A program may be broken up into manageable functions. A function is a collection of statements that performs a specific task. These small functions can then be executed in the desired order to solve the problem. This approach is sometimes called *divide and conquer* This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

Defining and Calling Functions

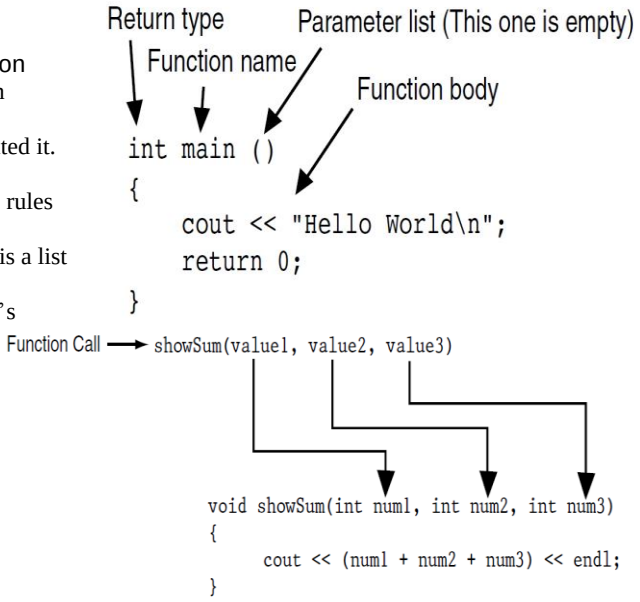
A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function. All function definitions have the following parts:

Return type: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.

Name: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.

Parameter list: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.

Body: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.



Void Functions

Calling a Function

A function is executed when it is *called.* Function main is called automatically when a program starts, but all other functions must be executed by *function call* statements.

Function Header ----------> void displayMessage()
Function Call ----------> displayMessage();
Function Prototypevoid ---------> void displayMessage();

Function Prototypes

A function prototype eliminates the need to place a function definition before all calls to the function. Function prototypes are also known as *function declarations*.

Sending Data into a Function

When a function is called, the program may send values into the function. The values that are passed into a function are called arguments, and the variables that receive those values are called parameters. There are several variations of these terms in use. Some call the arguments *actual parameters* and call the parameters *formal parameters*. Others use the terms *actual argument* and *formal argument*. If you pass an argument whose type is not the same as the parameter's type, the argument will be promoted or demoted automatically.

```
void  showSum ( int num1, num2, num3 )   // Error!
void  showSum ( int num1, int num2, int num3 )   // Correct
```

When passing a variable as an argument, simply write the variable name inside the parentheses of the function call. Do not write the data type of the argument variable in the function call. For example, the following funtion call will cause an error:

displayValue(int x); // Error!

The function call should appear as

displayValue(x); // Correct

Passing Data by Value

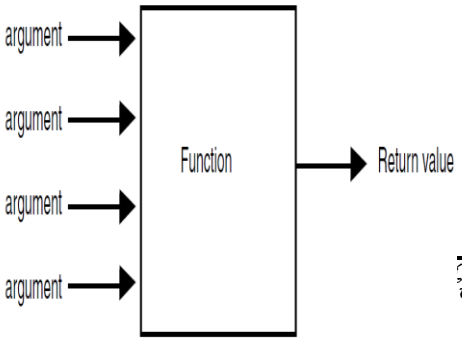
When only a copy of an argument is passed to a function, it is said to be *passed by value*. This is because the function receives a copy of the argument's value, and does not have access to the original argument. Functions are ideal for use in menu-driven programs. When the user selects an item from a menu, the program can call the appropriate function. A ***modular program*** is broken up into functions that perform specific tasks.

The return Statement

The return statement causes a function to end immediately. It's possible, however, to force a function to return before the last statement has been executed. When the **return statement is encountered**, the function immediately terminates and control of the program returns to the statement that called the function.

```
void divide(double arg1, double arg2) {
if (arg2 == 0.0)
{
        cout << "Sorry, I cannot divide by zero.\n";
        return;
}
cout << "The quotient is " << (arg1 / arg2) << endl;
}
```

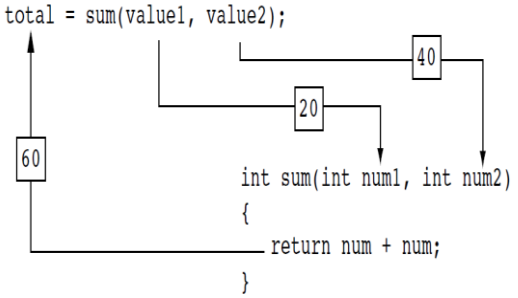
Value Returning Function



A function may send a value back to the part of the program that called the function. Functions that return a value are appropriately known as *value-returning functions*.

Syntax

Return**T**ype **F**unc**N**ame (**L**ist **O**f **P**arameters)
{ **B**ody
}



```
bool  isValid ( int number )
{
        bool status;
        if (number >= 1 && number <= 100)
                status = true;
        else
                status = false;
        return status;
}
```

Returning a Boolean Value

Functions may return true or false values.

Local Variables

A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope. A function's local variables exist only while the function is executing. This is known as the ***lifetime* of a local variable**.

Global Variables and Global Constants

A global variable is any variable defined outside all the functions in a program. The scope of a global variable is the portion of the program from the variable definition to the end. Unless you explicitly initialize numeric global variables, they are automatically initialized to zero. Global character variables are initialized to NULL.

Local and Global Variables with the Same Name

You cannot have two local variables with the same name in the same function. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same function the same name. However, you can have a local variable or a parameter variable with the same name as a global variable, or a global constant. When you do, the name of the local or parameter variable *shadows* the name of the global variable or global constant.

Static Local Variables

If a function is called more than once in a program, the values stored in the function's local variables do not persist between function calls. This is because the local variables are destroyed when the function terminates and are then re-created when the function starts again. Static local variables are not destroyed when a function returns. They exist for the lifetime of the program, even though their scope is only the function in which they are defined.

Syntax: ------→ **Static DataType VariableName** ;

Like global variables, all static local variables are initialized to zero by default. If you do provide an initialization value for a static local variable, the initialization only occurs once.

Default Arguments

Default arguments are passed to parameters automatically if no argument is provided in the function call. The default arguments are usually listed in the function prototype. Here is an example:

```
Void  showArea ( double = 20.0,  double = 10.0 ) ;
void  showArea  ( double  length  =  20.0,  double  width  =  10.0 ) ;
```

If a function does not have a prototype, default arguments may be specified in the function header.

Here is a summary of the important points about default arguments:

1. The value of a default argument must be a literal value or a named constant.
2. When an argument is left out of a function call (because it has a default value), all the arguments that come after it must be left out too.
3. When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be declared last.

```
void  calcPay ( int  empNum,  double  payRate,  double  hours  =  40.0 ) ;
calcPay ( 769,  15.75 ) ;          // Use default arg for 40 hours
calcPay ( 142,  12.00,  20 ) ;     // Specify number of hours
```

Using Reference Variables as Parameters

When used as parameters, reference variables allow a function to access the parameter's original argument. Changes to the parameter are also made to the argument. C++ provides a special type of variable called a *reference variable* that, when used as a function parameter, allows access to the original argument. Reference variables are defined like regular variables, except you place an ampersand (&) in front of the name. The ampersand must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call.

Only variables may be passed by reference. If you attempt to pass a non variable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result.

```
void  doubleNum ( int  &refVar )
{
        refVar  *=  2;
}
```

The variable refVar is called "a reference to an int."

Overloading Functions

Two or more functions may have the same name, as long as their parameter lists are different. Use a different set of parameters or parameters of different data types. In C++, each function has a signature. The ***function signature*** is the name of the function and the data types of the function's parameters in the proper order. Note that the function's return value is not part of the signature.

⇨ Square (int) => Square (double)

When an overloaded function is called, C++ uses the function signature to distinguish it from other functions with the same name.

The exit() Function

```
#include  <cstdlib> // For exit      => exit ( 0 )
```

The exit() function causes a program to terminate, regardless of which function or control mechanism is executing. The exit code zero is passed, which commonly indicates a successful exit. If you are unsure which code to use with the exit

```
void showFees (double memberRate, int months)
{
cout << "The showFees function was called with "
<< "the following arguments:\n"
<< "memberRate: " << memberRate << endl
<< "months: " << months << endl;
}
```

function, there are two named constants, EXIT_FAILURE and EXIT_SUCCESS, defined in cstdlib for you to use. The constant EXIT_FAILURE is defined as the termination code that commonly represents an unsuccessful exit under the current operating system. exit(EXIT_FAILURE);
The constant EXIT_SUCCESS is defined as the termination code that commonly represents a successful exit under the current operating system. exit(EXIT_SUCCESS)
Generally, the exit code is important only if you know it will be tested outside the program. If it is not used, just pass zero, or EXIT_SUCCESS.

Stubs and Drivers

Stubs and ***drivers*** are very helpful tools for testing and debugging programs that use functions. They allow you to test the individual functions in a program, in isolation from the parts of the program that call the functions. A *stub* is a dummy function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, and nothing more. Primarily, the stub allows you to determine whether your program is calling a function when you expect it to, and to confirm that valid values are being passed to the function. A driver is a program that tests a function by simply calling it. If the function accepts arguments, the driver passes test data. If the function returns a value, the driver displays the return value on the screen.

Arrays Hold Multiple Values — CH # 07

An array allows you to store and work with multiple values of the same data type. The values are stored together in consecutive memory locations. The name of this array is days. The number inside the brackets is the array's *size declarator.* It indicates the number of *elements,* or values, the array can hold. An array's size declarator must be a constant integer expression with a value greater than zero. The size of an array can be calculated by multiplying the size of an individual element by the number of elements in the array. dataType ArayName [size]; => Int days [6];

Accessing Array Elements

The individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements. Each element is assigned a number known as a *subscript*. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0. Subscript numbering in C++ always starts at zero. The subscript of the last element in an array is one less than the total number of elements in the array. If an array is defined globally, all of its elements are initialized to zero by default. Local arrays, however, have no default initialization value.

Inputting and Outputting Array Contents

```
const int ARRAY_SIZE = 5;
int numbers [ ARRAY_SIZE ] ;
for ( int coun t = 0; count < ARRAY_SIZE; count++ )
        numbers[count] = 99;
```

1. Reading Data from a File into an Array
2. Writing the Contents of an Array to a File

Watch for Off-by-One Errors

```
// This code has an off-by-one error.
Const int SIZE = 100 ;
int numbers [ SIZE ];
for ( int count = 1; count <= SIZE; count++ )
        numbers[count] = 0;
```

No Bounds Checking in C++

C++ gives you the freedom to store data past an array's boundaries.

Array Initialization

Arrays may be initialized when they are defined. The series of values inside the braces and separated with commas is called an *initialization list.* An array's initialization list cannot have more values than the array has elements.

```
const int MONTHS = 12;
int days [ MONTHS ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Partial Array Initialization

If an array is partially initialized, the uninitialized elements will be set to zero. This is true even if the array is defined locally. If we leave an element uninitialized, we must leave all the elements that follow it uninitialized as well.

```
int numbers [ 7 ] = { 1, 2, 4, 8 };
int array [ 6 ] = {2, 4,  , 8,  , 12 };          // NOT Legal!
```

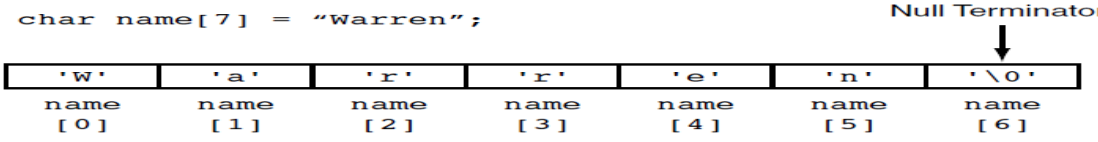
Implicit Array Sizing

It's possible to define an array without specifying its size, as long as you provide an initialization list. C++ automatically makes the array large enough to hold all the initialization values. double ratings [ ] = { 1.0, 1.5, 2.0, 2.5, 3.0 };

Initializing with Strings

When initializing a character array with a string, simply enclose the string in quotation marks, Char name [7] = "Warren";
The null terminator is not automatically included when an array is initialized with individual characters. It must be included in the initialization list, as shown below: char name [7] = { 'W', 'a', 'r', 'r', 'e', 'n', '\0'};
In Chapter 2 you were shown that to display a string stored in a character array, you simply use the stream insertion operator to send the name of the array (without the brackets) to the cout object.

```
char name[7] = "Warren";
```



NOTE: Recall from Chapter 2 that '\0' represents the null terminator. '\0' is an escape sequence that is stored in memory as a single character. Its ASCII code is 0.

NOTE: When using increment and decrement operators, be careful not to confuse the subscript with the array element. For example, the following statement decrements the variable `count`, but does nothing to the value in `amount[count]`:

```
amount[count--];
```

To decrement the value stored in `amount[count]`, use the following statement:

```
amount[count]--;
```

Processing Array Contents

Individual array elements are processed like any other type of variable
Array elements may also be used in relational expressions.

If (cost [20] < cost [0])

And the following statement sets up a while loop to iterate as long as value[place] does not equal 0: While (value [place] != 0)

Thou Shall Not Assign

Anytime the name of an array is used without brackets and a subscript, *it is seen as the array's beginning memory address*.

Printing the Contents of an Array

This explains why the following statement cannot be used to display the contents of array: cout << array << endl; //Wrong!

```
for (int count = 0; count < SIZE; count++)
    cout << array[count] << endl;
```

```
Const int SIZE = 4;
Int oldValues [ SIZE ] = {10, 100, 200, 300};
Int newValues [ SIZE ];

newValues = oldValues;   // Wrong!

For ( int count = 0;  count < SIZE;  count++ )
   newValues [count] = oldValues [count];
```

```
char name [ ] = "Ruth";
cout << name << endl ;
```

WARNING! Do not pass the name of a char array to cout if the char array does not contain a null-terminated C-string. If you do, cout will display all the characters in memory, starting at the array's address, until it encounters a null terminator.

Comparing Arrays

When you use the == operator with array names, the operator compares the beginning memory addresses of the arrays, not the contents of the arrays. The two array names in this code will obviously have different memory addresses.

By using the same subscript, you can build relationships between data stored in two or more arrays.

```
Const int SIZE = 5;
Int Array1 [ SIZE ] = { 5, 10, 15, 20, 25 };
int Array2 [ SIZE ] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0; // Loop counter variable
while (arraysEqual && count < SIZE)
{ if ( Array1 [ count ] != Array2 [ count ] )
        arraysEqual = false;
count++; }
if (arraysEqual)
cout << "The arrays are equal.\n";
else
cout << "The arrays are not equal.\n";
```

Arrays as Function Arguments To pass an array as an argument to a function, pass the name of the array. Array parameters work very much like reference variables. They give the function direct access to the original array.

```
#include < iostream >
using namespace std;
void showValues ( int  [ ], int);  // Function prototype
int main( )
{ const int Size = 8;
int numbers[Size] = {5, 10, 15, 20, 25, 30, 35, 40};
showValues(numbers, ARRAY_SIZE);
return 0;          }
void showValues(int nums[], int size)
{  for (int index = 0; index < size; index++)
          cout << nums[index] << " ";
cout << endl;      }
```

Summing the Values in a Numeric Array

```
const int Size = 24;
Int units [ Size ] ;
Int total = 0;   // Initialize accumulator
for ( int count = 0;  count < Size;  count++ )
      total += units [ count ];
```

Find the Highest in a Numeric Array

```
const int SIZE = 50;
int numbers [ SIZE ];
int count;
int highest;
highest = numbers[0];
for ( count = 1;  count < SIZE;  count++)
{
     If ( numbers [ count ] > highest )
          highest = numbers [ count ];
}
```

Find Lowest Values in a Numeric Array

```
int count;
int lowest;
lowest = numbers [ 0 ];
for (count = 1; count < SIZE; count++)
{
    if ( numbers [ count ] < lowest )
          lowest = numbers [ count ] ;
}
```

Average of the Values in a Numeric Array

```
const int Size = 24;
Int units [ Size ] ;
Int total = 0;   // Initialize accumulator
for ( int count = 0;  count < Size;  count++ )
      total += units [ count ];
average = total / Size;
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main()   {
const int Size = 100; // Array size
int numbers [ Size ];
int i = 0;
ifstream inputFile;
inputFile.open( "numbers.txt" );
while (i < Size  &&  inputFile >>
numbers[ I ] )
       count++;
inputFile.close();
cout << "The numbers are: ";
for (int i = 0; i < count; i++)
cout << numbers[i] << " ";
cout << endl;
return 0;
}
```

Two-Dimensional Arrays

A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data. To define a two-dimensional array, two size declarators are required. The first one is for the number of rows and the second one is for the number of columns. => double scores[3][4];

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
int hours[3][2] = {8, 5, 7, 9, 6, 3};      Initialize 2D Array
int table[3][2] = {{1}, {3, 4}, {5}};
```

	Column 0	Column 1	Column 2	Column 3
Row 0	scores[0] [0]	scores[0] [1]	scores[0] [2]	scores[0] [3]
Row 1	scores[1] [0]	scores[1] [1]	scores[1] [2]	scores[1] [3]
Row 2	scores[2] [0]	scores[2] [1]	scores[2] [2]	scores[2] [3]

Passing 2D Arrays to Functions

```
const int Col = 4;
const int  Rows = 3;
void showArr(int  [ ][Col], int );
int main()            {
int table[Rows][Col] = {{1, 2, 3, 4},
{5, 6, 7, 8}, {9, 10, 11, 12} };
cout << "The contents of table are:\n";
showArr(table, Rows);
return 0;             }
void showArr(int arr[ ][Col], int  rows)
{ for (int x = 0; x < rows; x++ )  {
    for (int y = 0; y < Col; y++)
      cout << setw(4) << arr[x][y] << " ";
    cout << endl;  }
```

Summing All the Elements of a 2D Array

```
const  int  ROWS  =  5; // Number of rows
Const int  COLS  =  5; // Number of columns
int total = 0; // Accumulator
int Arr [ROWS] [COLS] = {{2, 7, 9, 6, 4},
{6, 1, 8, 9, 4}, {4, 3, 7, 2, 9},
{9, 9, 0, 3, 1}, {6, 2, 7, 4, 1} };
// Sum the array elements.
for (int i = 0; row < ROWS; i++)
   for (int j = 0; col < COLS; j++)
       total += numbers[ row ] [ col ];
// Display the sum.
cout << "The total is " << total << endl;
```

Summing the Rows of a 2D Array

```
const int STUDENTS = 3;
const int SCORES = 5;
double total,  average;
double scores[STUDENTS][SCORES] =
{{88, 97, 79, 86, 94},  {86, 91, 78, 79, 84},
{82, 73, 77, 82, 89}};
for (int i = 0; i < STUDENTS; i++) {
      total = 0;
      for (int j = 0; j < SCORES; j++)
            total += scores[ I ] [ j ];
average = total / SCORES;
cout << "Score average for student "
<< ( i + 1) << " is " << average <<endl;
}
```

Summing the Columns of a 2D Array

```
const int STUDENTS = 3;
const int SCORES = 5;
double total,  average;
double scores[STUDENTS][SCORES] =
{ {88, 97, 79, 86, 94}, {86, 91, 78, 79, 84},
{82, 73, 77, 82, 89} };
for (int j = 0; j < SCORES; j++)  {
   total = 0;
   for (int i = 0; i < STUDENTS; i++)
       total += scores[ i ] [ j ];
   average = total / STUDENTS;
   cout << "Class average for test " <<
   (j + 1) << " is " << average << endl;  }
```

Arrays of Strings

A two-dimensional array of characters can be used as an array of strings.

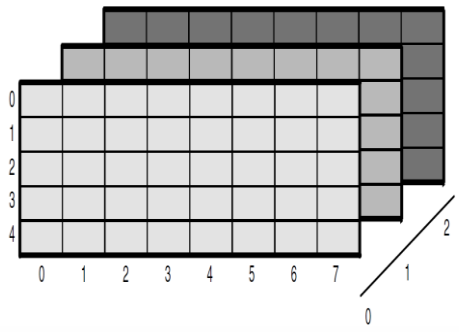
```
char scientists[4][9] = {"Galileo",
                         "Kepler",
                         "Newton",
                         "Einstein" };
```

G	a	l	i	l	e	o	\0	
K	e	p	l	e	r	\0		
N	e	w	t	o	n	\0		
E	i	n	s	t	e	i	n	\0

```
for (int count = 0; count < 4; count++)
   cout << scientists[count] << endl;
```

Arrays with Three or More Dimensions

```
double  seats [ 3 ] [ 5 ] [ 8 ];
```



The **Standard Template Library** offers a vector data type, which in many ways, is superior to standard arrays. The data types that are defined in the STL are commonly called *containers*. They are called containers because they store and organize data. 2 types of containers in the STL: sequence containers and associative containers. A *sequence container* organizes data in a sequential fashion, similar to an array. *Associative containers* organize data with keys, which allow rapid, random access to elements stored in the container.

A vector is like an array in the following ways:

- A vector holds a sequence of values, or elements.
- A vector stores its elements in contiguous memory locations.
- You can use the array subscript operator [ ] to read the individual elements in the vector.

However, a vector offers several advantages over arrays.

- You do not have to declare the number of elements that the vector will have.
- If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.
- vectors can report the number of elements they contain.

Defining a vector

To use vectors in your program, you must include the vector header file and using namespace std;

```
#include <vector>                    <= Header File
```

Syntax:

```
vector  < dataType >  VectorName ;
vector  < dataType >  vectorName ( size ) ;
vector  < dataType >  vectorName ( size , initializingValue ) ;
```

You may also initialize a vector with the values in another vector.

```
Vector < int > set2 ( set1 ) ;
```

After this statement executes, set2 will be a copy of set1.

To store a value in an element that already exists in a vector, you may use the array subscript operator [ ].

Member Function	Description
`at(element)`	Returns the value of the element located at *element* in the `vector`. *Example:* `x = vect.at(5);` This statement assigns the value of the fifth element of `vect` to `x`.
`capacity()`	Returns the maximum number of elements that may be stored in the `vector` without additional memory being allocated. (This is not the same value as returned by the `size` member function). *Example:* `x = vect.capacity();` This statement assigns the capacity of `vect` to `x`.
`clear()`	Clears a `vector` of all its elements. Example: `vect.clear();` This statement removes all the elements from `vect`.
`empty()`	Returns true if the `vector` is empty. Otherwise, it returns false. *Example:* `if (vect.empty()) cout << "The vector is empty.";` This statement displays the message if `vect` is empty.
`pop_back()`	Removes the last element from the `vector`. *Example:* `vect.pop_back();` This statement removes the last element of `vect`, thus reducing its size by 1.
`push_back(value)`	Stores a value in the last element of the `vector`. If the `vector` is full or empty, a new element is created. *Example:* `vect.push_back(7);` This statement stores 7 in the last element of `vect`.
`reverse()`	Reverses the order of the elements in the `vector`. (The last element becomes the first element, and the first element becomes the last element.) *Example:* `vect.reverse();` This statement reverses the order of the element in `vect`.
`resize(elements, value)`	Resizes a `vector` by *elements* elements. Each of the new elements is initialized with the value in *value*. *Example:* `vect.resize(5, 1);` This statement increases the size of `vect` by five elements. The five new elements are initialized to the value 1.
`swap(vector2)`	Swaps the contents of the `vector` with the contents of *vector2*. *Examble:* `vect1.swap(vect2);` This statement swaps the contents of `vect1` and `vect2`

Determining the Size of a vector

The size member function is especially useful when you are writing functions that accept vectors as arguments.

```
Void  showValues ( vector <int>); // prototype
showValues( values ); // Function Call
void showValues(vector<int> vect)  {
    for (int i = 0; i < vect.size ( ); i++)
      cout << vect[count] << endl; }
```

Introduction to Search Algorithms **CH # 08**

A search algorithm is a method of locating a specific item in a larger collection of data. The linear Search and the binary search.

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. A maximum of 20 comparisons will be made on an array of 1,000,000 elements ($2^{20} = 1{,}048{,}576$).

Introduction to Sorting Algorithms

Sorting algorithms are used to arrange data into some order.

The *bubble sort* and the *selection sort*.

The Linear Search

The *linear search* is a very simple algorithm. Sometimes called a *sequential search*, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered.

```
int searchList ( int list [ ], int numElems, int value)
{
    int index = 0; // Used as a subscript to search array
    int position = -1; // To record position of search value
    bool found = false; // Flag to indicate if the value was found
    while (index < numElems && !found)
    {
        if (list[index] == value) // If the value is found
        {
                found = true; // Set the flag
                position = index; // Record the value's subscript
        }
        index++; // Go to the next element
     }
     return position; // Return the position, or -1
}
```

The Selection Sort

The bubble sort is inefficient for large arrays because items only move by one element at a time. The selection sort, however, usually performs fewer exchanges because it moves items immediately to their final position in the array.

```
void selectionSort ( int array[ ], int size)
{
    int startScan, minIndex, minValue;
    for (startScan = 0; startScan < (size - 1); startScan++)
    {
            minIndex = startScan;
            minValue = array[startScan];
            for(int index = startScan + 1; index < size; index++)
            {
                if (array[index] < minValue)
                {
                        minValue = array[index];
                        minIndex = index;
                }
            }
            array[minIndex] = array[startScan];
            array[startScan] = minValue;
     }
}
```

The Binary Search

The *binary search* is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be sorted in order.

```
int binarySearch ( int array[ ], int numElems, int value)
{
    int first = 0, // First array element
    last = numElems - 1, // Last array element
    middle, // Midpoint of search
    position = -1; // Position of search value
    bool found = false; // Flag
    while (!found && first <= last)
    {
        middle = (first + last) / 2; // Calculate midpoint
        if (array[middle] == value) // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value) // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1; // If value is in upper half
    }
    return position;
}
```

The Bubble Sort

```
void sortArray(int array [ ], int size)
{
    bool swap;
    int temp;
    do
    {
        swap = false;
        for (int i = 0; i < (size - 1); i++)
        {
            if (array [ i ] > array [ i + 1 ] )
            {
                temp = array[ i ];
                array [ i ] = array [ i + 1];
                array [ i + 1] = temp;
                swap = true;
            }
        }
    } while (swap);
}
```

Getting the Address of a Variable Ch # 09

The address operator (&) returns the memory address of a variable. Each byte of memory has a unique *address*. When the address operator (&) is placed in front of a variable name, it returns the address of that variable. The address of the variable x is displayed in hexadecimal.

Pointer Variables which are often just called *pointers*, are designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables. We are passing the name of the array, numbers, and its size as arguments to the showValues function. In the function, the values parameter receives the address of the numbers array. It works like a pointer because it "points" to the numbers array.

Creating and Using Pointer Variables

dataType asterisk pointerName — int * ptr;
pointerName = &VariableNAme — ptr = &x;

The asterisk in front of the variable name indicates that ptr is a pointer variable. The int data type indicates that ptr can be used to hold the address of an integer variable. The definition statement above would read "ptr is a pointer to an int."

NOTE: So far you've seen three different uses of the asterisk in C++:

- As the multiplication operator, in statements such as
  distance = speed * time;
- In the definition of a pointer variable, such as
  int *ptr;
- As the indirection operator, in statements such as
  *ptr = 100;

The Relationship Between Arrays and Pointers

Array names can be used as constant pointers, and pointers can be used as array names. It's important to know, however, that pointers do not work like regular variables when used in mathematical statements. In C++, when you add a value to a pointer, you are actually adding that value *times the size of the data type being referenced by the pointer*.

When working with arrays, remember the following rule
Array [index] is equivalent to * (*array* + index)

Notice that the address operator is not needed when an array's address is assigned to a pointer. Because the name of an array is already an address, use of the & operator would be incorrect.

The only difference between array names and pointer variables is that you cannot change the address an array name points to. Array names are *pointer constants*.

Pointer Arithmetic

Some mathematical operations may be performed on pointers. We cannot multiply or divide a pointer. The following operations are allowable:

- The ++ and -- operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the + and - operators, or the += and -= operators.
- A pointer may be subtracted from another pointer.

Initializing Pointers

Pointers may be initialized with the address of an existing object.

```
int myValue;
int *pint = &myValue;

int ages[20];
int *pint = ages;

float myFloat;
int *pint = &myFloat; // Illegal!
```

Comparing Pointers

CONCEPT: If one address comes before another address in memory, the first address is considered "less than" the second. C++'s relational operators may be used to compare pointer values.

Pointers may be compared by using any of C++'s relational operators:
> < == != >= <=

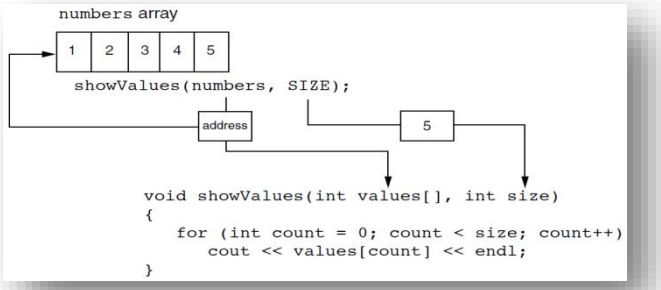
Pointers as Function Parameters

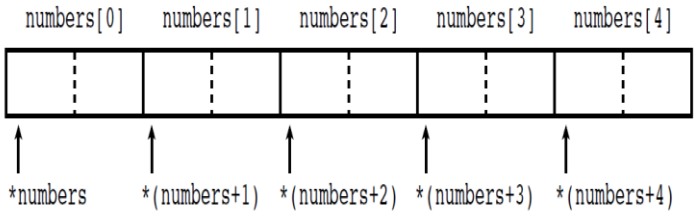
A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.

Pointers to Constants

Sometimes it is necessary to pass the address of a const item into a pointer. When this is the case, the pointer must be defined as a pointer to a const item. In passing the address of a constant into a pointer variable, the variable must be defined as a pointer to a constant. When you are writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to const. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and nonconstant arguments.

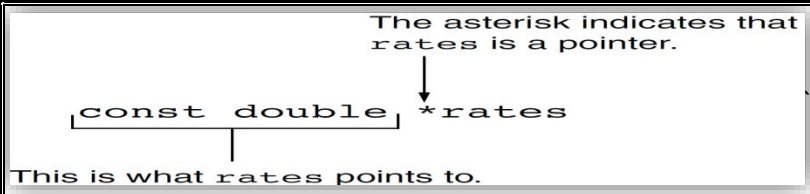
```
int main()
{
   int x = 25;
   cout << "The address of x is " << &x << endl;
   cout << "The size of x is " << sizeof(x) << " bytes\n";
   cout << "The value in x is " << x << endl;
   return 0;
}
```






```
int main()
{ const int SIZE = 8;
  int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
  int *numPtr;   // Pointer
  int count;     // Counter variable for loops
 // Use the pointer to display the array contents.
 cout << "The numbers in set are:\n";
 for (count = 0; count < SIZE; count++)
 {  cout << *numPtr << " ";
    numPtr++;     }
// Display the array contents in reverse order.
cout << "\nThe numbers in set backward are:\n";
for (count = 0; count < SIZE; count++)
{   numPtr--;
    cout << *numPtr << " ";  }
return 0;  }
```

```
// Function prototypes
void getNumber(int *);
void doubleValue(int *);
int main()
{  int number;
   // Call getNumber and pass the address of number.
   getNumber(&number);
   // Call doubleValue and pass the address of number.
   doubleValue(&number);
   // Display the value in number.
   cout << "That value doubled is " << number << endl;
   return 0;
}
void getNumber(int *input)
{  cout << "Enter an integer number: ";
   cin >> *input; }
void doubleValue(int *val)
{   *val *= 2;  }
```




```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                                12.85, 14.97,
                                10.35, 18.89 };
void displayPayRates(const double *rates, int size)
{   // Set numeric output formatting.
    cout << setprecision(2) << fixed << showpoint;
    // Display all the pay rates.
    for (int count = 0; count < size; count++)
    {  cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
    }
}
```

Constant Pointers

Here is the difference between a pointer to const and a const pointer:

• A pointer to const points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.

• With a const pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

```
int value = 22;
int * const ptr = &value;
```

Notice in the definition of ptr the word const appears after the asterisk. This means that ptr is a const pointer. Although the parameter is const pointer, we can call the function multiple times with different arguments.

```
#include <iostream>
using namespace std;
void displayValues(const int *, int);
int main()
{  const int SIZE = 6;
   const int array1[SIZE] = { 1, 2, 3, 4, 5, 6 };
   int array2[SIZE] = { 2, 4, 6, 8, 10, 12 };
   displayValues(array1, SIZE);
   displayValues(array2, SIZE);
   return 0;
}
void displayValues(const int *numbers, int size)
{   for (int count = 0; count < size; count++)
    {  cout << *(numbers + count) << " ";  }
    cout << endl;
}
```

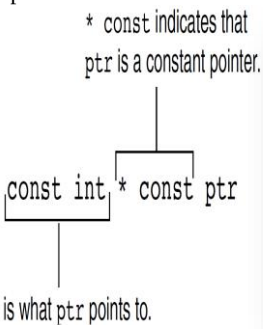
Constant Pointers to Constants

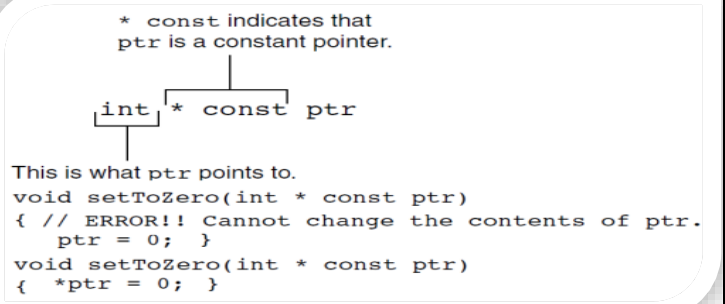
You can also have constant pointers to constants. For example, look at the following code:

```
int value = 22;
const int * const ptr = &value;
```

In the code, ptr is initialized with the address of value. Because ptr is a const pointer, we cannot write code that makes ptr point to anything else. Because ptr is a pointer to const, we cannot use it to change the contents of value. The following code shows one more example of a const pointer to a const.

```
void displayValues(const int * const numbers, int size)
{ // Display all the values.
  for (int count = 0; count < size; count++)
  { cout << *(numbers + count) << " "; }
  cout << endl;
}
```






```
void setToZero(int * const ptr)
{ // ERROR!! Cannot change the contents of ptr.
   ptr = 0;  }
void setToZero(int * const ptr)
{  *ptr = 0;  }
```

Dynamic Memory Allocation

Variables may be created and destroyed while a program is running. You allow the program to create its own variables "on the fly." This is called *dynamic memory allocation*, and is only possible through the use of pointers. The program can only access the newly allocated memory through its address, so a pointer is required to use those bytes. When memory cannot be dynamically allocated, C++ throws an exception and terminates the program. *Throwing an exception* means the program signals that an error has occurred. Under older compilers, the new operator returns the address 0, or NULL when it fails to allocate the requested amount of memory. (NULL is a named constant, defined in the iostream file, that stands for address 0.) The delete operator is used to free memory that was allocated with new.

NOTE: A pointer that contains the address 0 is called a *null pointer.*

```
int *iptr;          iptr = new int;          delete iptr;
```

```
int *iptr;
iptr  =  new int[100];
if (iptr == NULL)
{   cout << "Error allocating memory!\n";
    return;  }
else
    for (int count = 0; count < 100; count++)
        iptr[count] = 1;
delete [] iptr;
```

Value 0 is assigned to the sales pointer. It is a good practice to store 0 in a pointer variable after using delete on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents errors from occurring if delete is accidentally called on the pointer again. The delete operator is designed to have no effect when used on a null pointer.

```
int main()
{  double *sales,      // To dynamically allocate an array
          total = 0.0, // Accumulator
          average;     // To hold average sales
      int numDays,   // To hold the number of days of sales
          count;     // Counter variable
   // Get the number of days of sales.
   cout << "How many days of sales figures do you wish ";
   cout << "to process? ";
   cin >> numDays;
   // Dynamically allocate an array large enough to hold
   // that many days of sales amounts.
   sales = new double[numDays];
   // Get the sales figures for each day.
   cout << "Enter the sales figures below.\n";
   for (count = 0; count < numDays; count++)
   {  cout << "Day " << (count + 1) << ": ";
      cin >> sales[count];  }
   // Calculate the total sales
   for (count = 0; count < numDays; count++)
   {  total += sales[count];  }
   // Calculate the average sales per day
   average = total / numDays;
   // Display the results
   cout << fixed << showpoint << setprecision(2);
   cout << "\n\nTotal Sales: $" << total << endl;
   cout << "Average Sales: $" << average << endl;
   // Free dynamically allocated memory
   delete [] sales;
   sales = 0;       // Make sales point to null.
   return 0;
}
```

Returning Pointers from Functions

Functions can return pointers, but you must be sure the item the pointer references still exists. You should only return a pointer from a function if it is

- A pointer to an item that was passed into the function as an argument
- A pointer to a dynamically allocated chunk of memory

This function uses the new operator to allocate a section of memory. This memory will remain allocated until the delete operator is used or the program ends, so it's safe to return a pointer to it.

```
char *getName(char *name)
{
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

```
char *getName()
{ char *name;
  name = new char[81];
  cout << "Enter your name: ";
  cin.getline(name, 81);
  return name; }
```

```
// This program shows the donations made to the United Cause
// by the employees of CK Graphics, Inc. It displays
// the donations in order from lowest to highest
// and in the original order they were received.
#include <iostream>
using namespace std;
// Function prototypes
void arrSelectSort(int *[], int);
void showArray(int [], int);
void showArrPtr(int *[], int);
int main()
{  const int NUM_DONATIONS = 15;  // Number of donations
   // An array containing the donation amounts.
   int donations[NUM_DONATIONS] = {5,  100, 5,  25, 10,
                                   5,  25,  5,  5,  100,
                                   10, 15,  10, 5,  10 };
   // An array of pointers to int.
   int *arrPtr[NUM_DONATIONS];
   // Each element of arrPtr is a pointer to int. Make each
   // element point to an element in the donations array.
   for (int count = 0; count < NUM_DONATIONS; count++)
      arrPtr[count] = &donations[count];

   // Sort the elements of the array of pointers.
   arrSelectSort(arrPtr, NUM_DONATIONS);

   // Display the donations using the array of pointers. This
   // will display them in sorted order.
   cout << "The donations, sorted in ascending order, are: \n";
   showArrPtr(arrPtr, NUM_DONATIONS);

   // Display the donations in their original order.
   cout << "The donations, in their original order, are: \n";
   showArray(donations, NUM_DONATIONS);
   return 0;
}
```

```
void arrSelectSort(int *arr[], int size)
{  int startScan, minIndex;
   int *minElem;
   for (startScan = 0; startScan < (size - 1); startScan++)
   {  minIndex = startScan;
      minElem = arr[startScan];
      for(int index = startScan + 1; index < size; index++)
      {  if (*(arr[index]) < *minElem)
         {  minElem = arr[index];
            minIndex = index; }
      }
      arr[minIndex] = arr[startScan];
      arr[startScan] = minElem;
   }
}

void showArray(int arr[], int size)
{  for (int count = 0; count < size; count++)
      cout << arr[count] << " ";
   cout << endl;  }

void showArrPtr(int *arr[], int size)
{  for (int count = 0; count < size; count++)
      cout << *(arr[count]) << " ";
   cout << endl;  }
```

Program Output

```
The donations, sorted in ascending order, are:
5 5 5 5 5 5 10 10 10 10 15 25 25 100 100
The donations, in their original order, are:
5 100 5 25 10 5 25 5 5 100 10 15 10 5 10
```

Table 10-2

Function	Description
toupper	Returns the uppercase equivalent of its argument.
tolower	Returns the lowercase equivalent of its argument.

Character Testing Ch # 10

The C++ library provides several functions for testing characters. To use these functions you must include the **cctype** header file. These functions actually return an int value. The return value is nonzero to indicate true, or zero to indicate false.

```
char letter = 'a';
if (isupper(letter))
    cout << "Letter is uppercase.\n";
else
    cout << "Letter is lowercase.\n";
```

Character Case Conversion

The C++ library offers functions for converting a character to upper or lowercase. Each of the functions in Table accepts a single character argument. If the argument is already an uppercase letter, toupper returns it unchanged. Any nonletter argument passed to toupper is returned as it is.

Function	Description
`toupper`	Returns the uppercase equivalent of its argument.
`tolower`	Returns the lowercase equivalent of its argument.

```
cout << toupper('Z');      // Displays Z
cout << toupper('*');      // Displays *
cout << toupper ('&');     // Displays &
cout << toupper('%');      // Displays %
```

Internal Storage of C-Strings

In C++, a C-string is a sequence of characters stored in consecutive memory locations, terminated by a null character. *String* is a generic term that describes any consecutive sequence of characters. A word, a sentence, a person's name, and the title of a song are all strings. A *string literal* or *string constant* is the literal representation of a string in a program. In C++, string literals are enclosed in double quotation marks, such as: "What is your name?"

The term *C-string* describes a string whose characters are stored in consecutive memory locations and are followed by a null character, or null terminator.

String Literals

A string literal or string constant is enclosed in a set of double quotation marks (" ").All of a program's string literals are stored in memory as C-strings, with the null terminator automatically appended.

Library Functions for Working with C-Strings

The C++ library has numerous functions for handling C-strings. These functions perform various tests and manipulations, and require that the cstring header file be included.

The strlen Function

```
char name[50] = "Thomas Edison";
int length;
length = strlen(name);
length = strlen("Thomas Edison");
```

The strcat Function

The strcat function accepts two pointers to C-strings as its arguments. The function *concatenates*, or appends one string to another

```
char string1[13] = "Hello ";
char string2[7] = "World!";
strcat(string1, string2);

if ( sizeof ( string1 )  >=  ( strlen ( string1 )  +
 strlen ( string2 )  +  1 ) )
          strcat ( string1,  string2 );
else
          cout << "String1 is not large enough for both
          strings.\n";
```

Character Function	Description
`isalpha`	Returns true (a nonzero number) if the argument is a letter of the alphabet. Returns 0 if the argument is not a letter.
`isalnum`	Returns true (a nonzero number) if the argument is a letter of the alphabet or a digit. Otherwise it returns 0.
`isdigit`	Returns true (a nonzero number) if the argument is a digit from 0 through 9. Otherwise it returns 0.
`islower`	Returns true (a nonzero number) if the argument is a lowercase letter. Otherwise, it returns 0.
`isprint`	Returns true (a nonzero number) if the argument is a printable character (including a space). Returns 0 otherwise.
`ispunct`	Returns true (a nonzero number) if the argument is a printable character other than a digit, letter, or space. Returns 0 otherwise.
`isupper`	Returns true (a nonzero number) if the argument is an uppercase letter. Otherwise, it returns 0.
`isspace`	Returns true (a nonzero number) if the argument is a whitespace character. Whitespace characters are any of the following: space `' '`, vertical tab `'\v'`, newline `'\n'`, tab `'\t'`. Otherwise, it returns 0.

The strcpy Function

```
char string1[10] = "Hello", string2[10] = "World!";
cout << string1 << endl;              // Hello
cout << string2 << endl;              // World!
strcpy(string1, string2);
cout << string1 << endl;              // World!
cout << string2 << endl;              // World!
```

The strncat and strncpy Functions

Because the strcat and strcpy functions can potentially overwrite the bounds of an array, they make it possible to write unsafe code. The strncat functions works like strcat, except it takes a third argument specifying the maximum number of characters from the second string to append to the first.

```
Int   maxChars;
char  string1 [ 17 ]  =  "Welcome ";
char  string2 [ 18 ]  =  "to North Carolina";
cout  <<  string1 << endl;
cout  <<  string2 << endl;
maxChars  =  sizeof ( string1 )  -  ( strlen ( string1 )  +  1 );
strncat ( string1 ,  string2 ,  maxChars );
cout <<  string1 << endl;
```

The strncpy function allows you to copy a specified number of characters from a string to a destination. Calling strncpy is similar to calling strcpy, except you pass a third argument specifying the maximum number of characters from the second string to copy to the first.

```
Strncpy ( string1,  string2,  5 );
```

The strstr Function

The strstr function searches for a string inside of a string.

The function's first argument is the string to be searched, and the second argument is the string to look for. If the function finds the second string inside the first, it returns the address of the occurrence of the second string within the first string. Otherwise it returns the address 0, or the NULL address.

```
char arr[] = "Four score and seven years ago";
char *strPtr;
cout << arr << endl;
strPtr = strstr(arr, "seven"); // search for "seven"
cout << strPtr << endl;
```

String/Numeric Conversion Func

The C++ library provides functions for converting a string representation of a number to a numeric data type and vice versa. These functions require the cstdlib header file to be included.

There is a great difference between a number that is stored as a string and one stored as a numeric value. The string "26792" isn't actually a number, but a series of ASCII codes representing the individual digits of the number. It uses six bytes of memory (including the null terminator). Because it isn't an actual number, it's not possible to perform mathematical operations with it, unless it is first converted to a numeric value.

Function	Description
`atoi`	Accepts a C-string as an argument. The function converts the C-string to an integer and returns that value. *Example Usage:* `num = atoi("4569");`
`atol`	Accepts a C-string as an argument. The function converts the C-string to a `long` integer and returns that value. *Example Usage:* `lnum = atol("500000");`
`atof`	Accepts a C-string as an argument. The function converts the C-string to a `double` and returns that value. *Example Usage:* `fnum = atof("3.14159");`
`itoa`	Converts an integer to a C-string.* The first argument, `value`, is the integer. The result will be stored at the location pointed to by the second argument, `string`. The third argument, `base`, is an integer. It specifies the numbering system that the converted integer should be expressed in (8 = octal, 10 = decimal, 16 = hexadecimal, etc.). *Example Usage:* `itoa(value, string, base);`

*The `itoa` function is not supported by all compilers.

Function	Description
`strlen`	Accepts a C-string or a pointer to a C-string as an argument. Returns the length of the C-string (not including the null terminator.) *Example Usage:* `len = strlen(name);`
`strcat`	Accepts two C-strings or pointers to two C-strings as arguments. The function appends the contents of the second string to the first C-string. (The first string is altered, the second string is left unchanged.) *Example Usage:* `strcat(string1, string2);`
`strcpy`	Accepts two C-strings or pointers to two C-strings as arguments. The function copies the second C-string to the first C-string. The second C-string is left unchanged. *Example Usage:* `strcpy(string1, string2);`
`strncat`	Accepts two C-strings or pointers to two C-strings, and an integer argument. The third argument, an integer, indicates the maximum number of characters to copy from the second C-string to the first C-string. *Example Usage:* `strncat(string1, string2, n);`
`strncpy`	Accepts two C-strings or pointers to two C-strings, and an integer argument. The third argument, an integer, indicates the maximum number of characters to copy from the second C-string to the first C-string. If `n` is less than the length of `string2`, the null terminator is not automatically appended to `string1`. If `n` is greater than the length of `string2`, `string1` is padded with '\0' characters. *Example Usage:* `strncpy(string1, string2, n);`
`strcmp`	Accepts two C-strings or pointers to two C-strings arguments. If `string1` and `string2` are the same, this function returns 0. If `string2` is alphabetically greater than `string1`, it returns a negative number. If `string2` is alphabetically less than `string1`, it returns a positive number. *Example Usage:* `if (strcmp(string1, string2))`
`strstr`	Accepts two C-strings or pointers to two C-strings as arguments. Searches for the first occurrence of `string2` in `string1`. If an occurrence of `string2` is found, the function returns a pointer to it. Otherwise, it returns a NULL pointer (address 0). *Example Usage:* `cout << strstr(string1, string2);`

The C++ string Class

Standard C++ provides a special data type for storing and working with strings. C++ provides two ways of storing and working with strings. One method is to store them as C-strings in character array variables. Another way is to store them in string class objects. **What Is the string Class?**

The string class is an abstract data type. This means it is not a built-in, primitive data type like int or char. The first step in using the string class is to #include the string header file.

```
#include  <string>
string   movieTitle;
getline ( cin, movietitle ) ;
```

The code for the dollarFormat function follows.

```
void  dollarFormat ( string  &currency )
{
      int dp;
      dp = currency.find('.'); // Find decimal point
      if  ( dp  >  3 )  / / Insert commas
      {
          for (int x = dp - 3; x > 0; x -= 3)
          currency.insert(x, ",");
      }
      currency.insert(0, "$"); // Insert dollar sign
}
```

Definition	Description
`string address;`	Defines an empty `string` object named `address`.
`string name("William Smith");`	Defines a `string` object named `name`, initialized with "William Smith."
`string person1(person2);`	Defines a `string` object named `person1`, which is a copy of `person2`. `person2` may be either a `string` object or character array.
`string set1(set2, 5);`	Defines a `string` object named `set1`, which is initialized to the first five characters in the character array `set2`.
`string lineFull('z', 10);`	Defines a `string` object named `lineFull` initialized with 10 `'z'` characters.
`string firstName(fullName, 0, 7);`	Defines a `string` object named `firstName`, initialized with a substring of the `string` `fullName`. The substring is seven characters long, beginning at position 0.

Supported Operator	Description
`>>`	Extracts characters from a stream and inserts them into the `string`. Characters are copied until a whitespace or the end of the string is encountered.
`<<`	Inserts the `string` into a stream.
`=`	Assigns the `string` on the right to the `string` object on the left.
`+=`	Appends a copy of the `string` on the right to the `string` object on the left.
`+`	Returns a `string` that is the concatenation of the two `string` operands.
`[]`	Implements array-subscript notation, as in `name[x]`. A reference to the character in the `x` position is returned.
Relational Operators	Each of the relational operators is implemented: `< > <= >= == !=`

Member Function Example	Description
`theString.append(n, 'z');`	Appends `n` copies of 'z' to `theString`.
`theString.append(str);`	Appends `str` to `theString`. `str` can be a `string` object or character array.
`theString.append(str, n);`	The first `n` characters of the character array `str` are appended to `theString`.
`theString.append(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are appended to `theString`. If `theString` is too small, the function will copy as many characters as possible.
`theString.assign(n, 'z');`	Assigns `n` copies of `'z'` to `theString`.
`theString.assign(str);`	Assigns `str` to `theString`. `str` can be a `string` object or character array.
`theString.assign(str, n);`	The first `n` characters of the character array `str` are assigned to `theString`.
`theString.assign(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are assigned to `theString`. If `theString` is too small, the function will copy as many characters as possible.
`theString.at(x);`	Returns the character at position `x` in the `string`.
`theString.begin();`	Returns an iterator pointing to the first character in the `string`.
`theString.c_str();`	Returns a character array containing a null terminated string, as stored in `theString`.
`theString.capacity();`	Returns the size of the storage allocated for the `string`.
`theString.clear();`	Clears the `string` by deleting all the characters stored in it.

Member Function Example	Description
`theString.compare(str);`	Performs a comparison like the `strcmp` function (see Chapter 4), with the same return values. `str` can be a `string` object or a character array.
`theString.compare(x, n, str);`	Compares `theString` and `str`, starting at position `x`, and continuing for `n` characters. The return value is like `strcmp`. `str` can be a `string` object or character array.
`theString.copy(str, x, n);`	Copies the character array `str` to `theString`, beginning at position `x`, for `n` characters. If `theString` is too small, the function will copy as many characters as possible.
`theString.empty();`	Returns true if `theString` is empty.
`theString.end();`	Returns an iterator pointing to the last character of the string in `theString`.
`theString.erase(x, n);`	Erases `n` characters from `theString`, beginning at position `x`.
`theString.find(str, x);`	Returns the first position at or beyond position `x` where the string `str` is found in `theString`. `str` may be either a `string` object or a character array.
`theString.find('z', x);`	Returns the first position at or beyond position `x` where 'z' is found in `theString`.
`theString.insert(x, n, 'z');`	Inserts 'z' `n` times into `theString` at position `x`.
`theString.insert(x, str);`	Inserts a copy of `str` into `theString`, beginning at position `x`. `str` may be either a `string` object or a character array.
`theString.length();`	Returns the length of the string in `theString`.
`theString.replace(x, n, str);`	Replaces the `n` characters in `theString` beginning at position `x` with the characters in string object `str`.
`theString.resize(n, 'z');`	Changes the size of the allocation in `theString` to `n`. If `n` is less than the current size of the string, the string is truncated to `n` characters. If `n` is greater, the string is expanded and 'z' is appended at the end enough times to fill the new spaces.
`theString.size();`	Returns the length of the string in `theString`.
`theString.substr(x, n);`	Returns a copy of a substring. The substring is `n` characters long and begins at position `x` of `theString`.
`theString.swap(str);`	Swaps the contents of `theString` with `str`.

Abstract Data Types — CH # 11, Structured Data

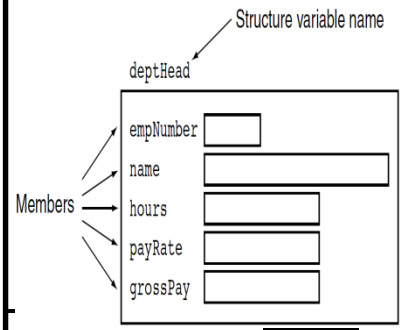
Abstract data types (ADTs) are data types created by the programmer. ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them. An ***abstraction*** is a general model of something. It is a definition that includes only the general characteristics of an object. For example, the term "dog" is an abstraction. It defines a general type of animal. A real-life dog is not abstract. It is concrete. Abstract Data Types An abstract data type (ADT) is a data type created by the programmer and is composed of one or more primitive data types.

Combining Data into Structures

C++ allows you to group several variables together into a single item known as a structure. So far you've written programs that keep data in individual variables. If you need to group items together, C++ allows you to create arrays. The limitation of arrays, however, is that all the elements must be of the same data type. Sometimes a relationship exists between items of different types. To create a relationship between variables, C++ gives you the ability to package them together into a *structure*. The *tag* is the name of the structure. As you will see later, it's used like a data type name. Remember that structure variables are actually made up of other variables known as members. An instance of a structure is a variable that exists in memory. In review, there are typically two steps to implementing structures in a program:

- Create the structure declaration. This establishes the tag (or name) of the structure and a list of items that are members.
- Define variables (or instances) of the structure and use them in the program to hold data.

```
struct tag
{
   variable declaration;
   // ... more declarations
   //     may follow...
};
```



Accessing Structure Members

The *dot operator* (.) allows you to access structure members in a program. The dot operator connects the name of the member variable with the name of the structure variable it belongs to.

Comparing Structure Variables

You cannot perform comparison operations directly on structure variables. if (circle1 == circle2) // Error!

In order to compare two structures, you must compare the individual members, as shown in the following code.

```
if (circle1.radius == circle2.radius &&
circle1.diameter == circle2.diameter &&
circle1.area == circle2.area)
```

Strings as Structure Members

When a character array is a structure member, you can use the same string manipulation techniques with it as you would with any other character array. For example, assume product.description is a character array. The following statement copies into it the string "19-inch television":

```
Strcpy ( product.description, "19-inch television");
```

Also, assume that product.partNum is a 15-element character array. The following statement reads into it a line of input:

```
cin.getline(product.partNum, 15);

// Assemble the full name.
Strcpy ( personName.full, personName.first );
Strcat  ( personName.full, " ") ;
Strcat  ( personName.full, personName.middle);
Strcat  ( personName.full, " ");
Strcat  ( personName.full, personName.last);
```

Initializing a Structure

The members of a structure variable may be initialized with starting values when the structure variable is defined.

```
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;     };
```

A variable may then be defined with an initialization list, as shown in the following:

```
CityInfo location = {"Asheville", "NC", 50000, 28};
```

You do not have to provide initializers for all the members of a structure variable.

```
CityInfo location = {"Tampa"};
CityInfo location = {"Atlanta", "GA"};
CityInfo location = {"Knoxville", "TN", , 90}; // Illegal!
```

It's important to note that you cannot initialize a structure member in the declaration of the structure. For instance, the following declaration is illegal:

```
struct  CityInfo    // Illegal structure declaration
{
    char cityName[30] = "Asheville"; // Error!
    char state[3] = "NC"; // Error!
    long population = 50000; // Error!
    int distance = 28; // Error!          };
```

Arrays of Structures

Arrays of structures can simplify some programming tasks. It's possible for a structure variable to be a member of another structure variable.

```
#include <iostream>
#include <iomanip>
using namespace std;

const int SIZE = 25; // Array size

struct PayRoll
{
   int empNumber;     // Employee number
   char name[SIZE];   // Employee's name
   double hours;      // Hours worked
   double payRate;    // Hourly pay rate
   double grossPay;   // Gross pay
};

int main()
{
   PayRoll employee; // employee is a PayRoll structure.

   // Get the employee's number.
   cout << "Enter the employee's number: ";
   cin >> employee.empNumber;

   // Get the employee's name.
   cout << "Enter the employee's name: ";
   cin.ignore();  // To skip the remaining '\n' character
   cin.getline(employee.name, SIZE);

   // Get the hours worked by the employee.
   cout << "How many hours did the employee work? ";
   cin >> employee.hours;

   // Get the employee's hourly pay rate.
   cout << "What is the employee's hourly pay rate? ";
   cin >> employee.payRate;

   // Calculate the employee's gross pay.
   employee.grossPay = employee.hours * employee.payRate;

   // Display the employee data.
   cout << "Here is the employee's payroll data:\n";
   cout << "Name: " << employee.name << endl;
   cout << "Number: " << employee.empNumber << endl;
   cout << "Hours worked: " << employee.hours << endl;
   cout << "Hourly pay rate: " << employee.payRate << endl;
   cout << fixed << showpoint << setprecision(2);
   cout << "Gross pay: $" << employee.grossPay << endl;
   return 0;
}
```

Structures as Function Arguments

Structure variables may be passed as arguments to functions. Structures, like all variables, are normally passed by value into a function. If a function is to access the members of the original argument, a reference variable may be used as the parameter.

```
struct Rectangle  {
double length;
double width;
double area; };

double multiply(double x, double y)
{     return x * y;     }

box.area = multiply(box.length, box.width);void
showRect(Rectangle r)  {
cout << r.length << endl;
cout << r.width << endl;
cout << r.area << endl;    }
showRect(box);
```

Constant Reference Parameters

Disadvantage of passing an object by reference is that the function has access to the original argument. It can potentially alter the argument's value. This can be prevented, however, by passing the argument as a constant reference.

```
Void  showItem ( const  InventoryItem  &p )
{  // …….  }
```

Returning a Structure from a Function

A function may return a structure.

```
struct Circle    {
    double radius;
    double diameter;
    double area;    };
Circle getCircleData()   {
    Circle temp; // Temporary Circle structure
     temp.radius = 10.0; // Store the radius
     temp.diameter = 20.0; // Store the diameter
     temp.area = 314.159; // Store the area
     return temp; // Return the temporary structure    }

myCircle = getCircleData();
```

Pointers to Structures

You may take the address of a structure variable and create variables that are pointers to structures.

```
Circle myCircle = { 10.0, 20.0, 314.159 };
Circle *cirPtr;              cirPtr = &myCircle;
```

One might think the following statement would access the radius member of the structure pointed to by cirPtr, but it doesn't: `*cirPtr.radius = 10;`

The dot operator has higher precedence than the indirection operator, so the indirection operator tries to dereference cirPtr.radius, not cirPtr. To dereference the cirPtr pointer, a set of parentheses must be used. `(*cirPtr).radius = 10`

C++ has a special operator for dereferencing structure pointers. It's called the *structure pointer operator*, and it consists of a hyphen (-) followed by the greater-than symbol (>).; `cirPtr->radius = 10;`

Expression	Description
`s->m`	`s` is a structure pointer and `m` is a member. This expression accesses the `m` member of the structure pointed to by `s`.
`*a.p`	`a` is a structure variable and `p`, a pointer, is a member. This expression dereferences the value pointed to by `p`.
`(*s).m`	`s` is a structure pointer and `m` is a member. The * operator dereferences `s`, causing the expression to access the `m` member of the structure pointed to by `s`. This expression is the same as `s->m`.
`*s->p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. (The -> operator dereferences `s` and the * operator dereferences `p`.)
`*(*s).p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. `(*s)` dereferences `s` and the outermost * operator dereferences `p`. The expression `*s->p` is equivalent.

Unions is like a structure, except all the members occupy the same memory area.

```
union   PaySource
{
    short hours;
    float sales;    };
```

The difference is that all the members of a union use the same memory area, so only one member can be used at a time. A union might be used in an application where the program needs to work with two or more values (of different data types), but only needs to use one of the values at a time. Unions conserve memory by storing all their members in the same memory location. The entire variable will only take up as much memory as the largest member (in this case, a float). Everything else you already know about structures applies to unions. e.g arrays of unions, passed as an argument to a function or returned from a function. Pointers to unions and the members of the union referenced by the pointer can be accessed with the -> operator.

Anonymous Unions

```
union
{
member declaration;
...
};
```

The members of an anonymous union have names, but the union itself has no name. An anonymous union declaration actually creates the member variables in memory, so there is no need to separately define a union variable. Anonymous unions are simple to use because the members may be accessed without the dot operator. Notice the anonymous union is declared inside function main. If an anonymous union is declared globally, it must be declared static. This means the word static must appear before the word union.

An enumerated data type is a programmer-defined data type. It consists of values known as enumerators, which represent integer constants. Using the enum key word you can create your own data type and specify the values that belong to that type. Such a type is known as an *enumerated data type*. Here is an example of an enumerated data type declaration:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

An enumerated type declaration begins with the key word enum, followed by the name of the type, followed by a list of identifiers inside braces, and is terminated with a semicolon. The example declaration creates an enumerated data type named Day. The identifiers MONDAY, TUESDAY, WEDNESDAY, THURSDAY, and FRIDAY, which are listed inside the braces, are known as *enumerators*. They represent the values that belong to the Day data type. Note that the enumerators are not enclosed in quotation marks, therefore they are not strings. Enumerators must be legal C++ identifiers. Once you have created an enumerated data type in your program, you can define variables of that type. For example, the following statement defines workDay as a variable of the Day type: `Day workDay;`

Because workDay is a variable of the Day data type, we may assign any of the enumerators MONDAY, TUESDAY, WEDNESDAY, THURSDAY, or FRIDAY to it. For example, the following statement assigns the value WEDNESDAY to the workDay variable. `Day workDay = WEDNESDAY;`

So just what are these enumerators MONDAY, TUESDAY, WEDNESDAY, THURSDAY, and FRIDAY? You can think of them as integer named constants. Internally, the compiler assigns integer values to the enumerators, beginning with 0. The enumerator MONDAY is stored in memory as the number 0, TUESDAY is stored as 1, and so forth.

enum typeName {one or more enumerator } ;

Even though the enumerators of an enumerated data type are stored in memory as integers, cannot directly assign an integer value to an enum variable

```
workDay = 3 ;     // Error!
```

However, if circumstances require that you store an integer value in an enum variable, you can do so by casting the integer. Here is an example:

```
workday = static_cast <Day> (3) ;
```

Although you cannot directly assign an integer value to an enum variable, you can directly assign an enumerator to an integer variable.
Enumerator values can be compared using the relational operators.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
int x = THURSDAY;   cout << x << endl;
```

Enumerator values can be compared using the relational operators.

```
Day workDay = FRIDAY;
int x = workDay;
cout << x << endl;
```

File Operations — Ch 12, Advanced File Operation

A file is a collection of data that is usually stored on a computer's disk. Data can be saved to files and then later reused. The stream extraction operator (>>) may be used with an ifstream object to read data from a file, and that the stream insertion operator (<<) may be used with an ofstream object to write data to a file.

fstream File Stream. This data type can be used to create files, write data to them, and read data from them. fstream dataFile;

An fstream object's open function requires two arguments, however. The first argument is a string containing the name of the file. The second argument is a file access flag that indicates the mode in which you wish to open the file.

```
dataFile.open("info.txt", ios::out);
```

The second argument is the file access flag ios::out. This tells C++ to open the file in output mode.

```
dataFile.open("info.txt", ios::in);
dataFile.open("info.txt", ios::in | ios::out);
```

This statement opens the file info.txt in both input and output modes. This means data may be written to and read from the file. When used by itself, the ios::out flag causes the file's contents to be deleted if the file already exists. When used with the ios::in flag, however, the file's existing contents are preserved. If the file does not already exist, it will be created. The following statement opens the file in such a way that data will only be written to its end: dataFile .open (" info.tx t", ios::out | ios::app);

The characters are added to the file sequentially, in the order they are written by the program. The very last character is an *end-of-file marker*. It is a character that marks the end of the file and is automatically written when the file is closed.

ofstream The file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is truncated).

ifstream The file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from its beginning. If the file does not exist, the open function fails.

Testing: If the file does not exist, the open operation will fail.

File Output Formatting

File output may be formatted in the same way that screen output is formatted.

File stream objects may Be passed by reference To functions.

Testing

```
fstream fin;
fin.open("values.txt", ios::in);
if ( fin.fail() ) {
    // The file does not exist, so create it
    fin.open("values.txt", ios::out);
    // Continue to process the file... }
else // The file already exists.
{   fin.close();
   cout << "The file already exists.\n"; }
```

```
bool openFileIn(fstream &file, char *name)
{   bool status;
   file.open(name, ios::in);
   if (file.fail())
     status = false;
   else
     status = true;
   return status;  }
```

The name and extension are separated by a period, known as a "dot."

File Name Extension	File Contents
myprog.bas	BASIC program
menu.bat	Windows batch file
install.doc	Microsoft Word document
crunch.exe	Executable file
bob.html	HTML (Hypertext Markup Language) file
3dmodel.java	Java program or applet
vacation.jpg	JPEG image file
invent.obj	Object file
instructions.pdf	Adobe Portable Document Format file
prog1.prj	Borland C++ project file
ansi.sys	System device driver
readme.txt	Text file

File Access Flag	Meaning
ios::app	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
ios::ate	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
ios::binary	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
ios::in	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
ios::out	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
ios::trunc	If the file already exists, its contents will be deleted (truncated). This is the default mode used by ios::out.

More Detailed Error Testing

All stream objects have error state bits that indicate the condition of the stream. All stream objects contain a set of bits that act as flags. These flags indicate the current state of the stream.

Member Functions for Reading and Writing Files

File stream objects have member functions for more specialized file reading and writing. If whitespace characters are part of the data in a file, a problem arises when the file is read by the >> operator. Because the operator considers whitespace characters as delimiters, it does not read them.

The getline Member Function

The function reads a "line" of data, including whitespace characters. Here is an example of the function call:

dataFile.getline (str , 81, '\n') ;

The three arguments in this statement are explained as follows:

Str This is the name of a character array, or a pointer to a section of memory. The data read from the file will be stored here.

81 This number is one greater than the maximum number of characters to be read. In this example, a maximum of 80 characters will be read.

'\n' This is a delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading before it has read the maximum number of characters. (This argument is optional. If it's left out, '\n' is the default.)

The get Member Function

It reads a single character from the file. Here is an example of its.

inFile.get (ch); ch is a char variable. A character will be read from the file and stored in ch.

The put Member Function

The put member function writes a single character to the file.

outFile.put(ch); the variable ch is assumed to be a char variable. Its contents will be written to the file associated with the file stream object outFile.

It's possible to have more than one file open at once in a program. In C++, you open multiple files by defining multiple file stream objects. ifstream file1, file2, file3;

The file is opened and read. Each character is converted to uppercase and written to a second file called out.txt. This type of program can be considered a *filter*. Filters read the input of one file, changing the data in some fashion, and write it out to a second file. The second file is a modified version of the first file.

```
#include <iostream>
#include <fstream>
using namespace std;
const int MAX_LINE_SIZE = 81;
bool openFileIn(fstream &, char *);
void showContents(fstream &);

int main()
{  fstream dataFile;
   if (!openFileIn(dataFile,"demofile.txt"))
   { cout << "File open error!" << endl;
     return 0;   // Exit the program on error. }
   cout << "File opened successfully.\n";
   cout << "Now reading data from the file.\n\n"
   showContents(dataFile);
   dataFile.close();
   cout << "\nDone.\n";
   return 0;
}

bool openFileIn(fstream &file, char *name)
{  file.open(name, ios::in);
   if (file.fail())
      return false;
   else
      return true; }

void showContents(fstream &file)
{   char line[MAX_LINE_SIZE];
    while (file >> line)
       cout << line << endl; }
```

Bit	Description
ios::eofbit	Set when the end of an input stream is encountered.
ios::failbit	Set when an attempted operation has failed.
ios::hardfail	Set when an unrecoverable error has occurred.
ios::badbit	Set when an invalid operation has been attempted.
ios::goodbit	Set when all the flags above are not set. Indicates the stream is in good condition.

Function	Description
eof()	Returns true (nonzero) if the eofbit flag is set, otherwise returns false.
fail()	Returns true (nonzero) if the failbit or hardfail flags are set, otherwise returns false.
bad()	Returns true (nonzero) if the badbit flag is set, otherwise returns false.
good()	Returns true (nonzero) if the goodbit flag is set, otherwise returns false.
clear()	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

Binary File

By default, files are opened in text mode. **Binary files contain data that is not necessarily stored as ASCII text.** Data can be stored in a file in its pure, binary format. The first step is to open the file in binary mode. This is accomplished by using the ios::binary flag. file.open("stuff.dat", ios::out | ios::binary); Notice the ios::out and ios::binary flags are joined in the statement with the | operator. This causes the file to be opened in both output and binary modes.

The write and read Member Functions

The file stream object's write member function is used to write binary data to a file. *fileObject*.write(*address*, *size*); Let's look at the parts of this function call format.

- *fileObject* is the name of a file stream object.
- *address* is the starting address of the section of memory that is to be written to the file. This argument is expected to be the address of a char (or a pointer to a char).
- *size* is the number of bytes of memory to write. This argument must be an integer value.

Example:
```
char letter = 'A';
file.write(&letter, sizeof(letter));
```

The read member function is used to read binary data from a file into memory. *fileObject*.read(*address*, *size*); Here are the parts of this function call format:

- *fileObject* is the name of a file stream object.
- *address* is the starting address of the section of memory where the data being read from the file is to be stored. This is expected to be the address of a char (or a pointer to a char).
- *size* is the number of bytes of memory to read from the file. This argument must be an integer value.

Example:
```
char letter;
file.read(&letter, sizeof(letter));
```

The following code shows another example. This code reads enough data from a binary file to fill an entire char array.
```
char data[4];
file.read(data, sizeof(data));
```

Writing Data Other Than char to Binary Files

Because the write and read member functions expect their first argument to be a pointer to a char, you must use a type cast when writing and reading items that are of other data types. To convert a pointer from one type to another you should use the reinterpret_cast type cast.

reinterpret_cast<*dataType*>(*value*)

where *dataType* is the data type that you are converting to, and *value* is the value that you are converting.

1.
```
int x = 65;
char *ptr;
ptr = reinterpret_cast<char *>(&x);
```

2.
```
int x = 27;
file.write(reinterpret_cast<char *>(&x), sizeof(x));
```

3.
```
const int SIZE = 10;
int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
file.write(reinterpret_cast<char *>(numbers),
sizeof(numbers));
```

4.
```
const int SIZE = 10;
int numbers[SIZE];
file.read(reinterpret_cast<char *>(numbers),
sizeof(numbers));
```

Creating Records with Structures

Structures may be used to store fixed-length records to a file. A field is an individual piece of data pertaining to a single item. A record is made up of fields and is a complete set of data about a single item. Once the members (or fields) of person are filled with data, the entire variable may be written to a file using the write func:

```
const int NAME_SIZE = 51,
ADDR_SIZE = 51,
PHONE_SIZE = 14;
struct Info {
char name[NAME_SIZE];
int age;
char address1[ADDR_SIZE];
char address2[ADDR_SIZE];
char phone[PHONE_SIZE]; };
Info person;
```

```
file.write(reinterpret_cast<char *>(&person), sizeof(person));
```

Because structures can contain a mixture of data types, you should always use the ios::binary mode when opening a file to store them.

Random-Access Files

Means non-sequentially accessing data in a file.

When a file is opened, the position where reading and/or writing will occur is at the file's beginning (unless the ios::app mode is used, which causes data to be written to the end of the file). In random file access, a program may immediately jump to any byte in the file without first reading the preceding bytes.

The seekp and seekg Member Functions

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are seekp and seekg. The seekp function is used with files opened for output and seekg is used with files opened for input. "p" stands for "put" and "g" stands for "get."

```
file.seekp(20L, ios::beg);
```

The first argument is a long integer representing an offset into the file. This is the number of the byte you wish to move to. In this example, 20L is used. (Remember, the L suffix forces the compiler to treat the number as a long integer.) This statement moves the file's write position to byte number 20. (All numbering starts at 0, so byte number 20 is actually the twenty-first byte.)

The second argument is called the mode, and it designates where to calculate the offset *from*. The flag ios::beg means the offset is calculated from the beginning of the file. Alternatively, the offset can be calculated from the end of the file or the current position in the file

Mode Flag	Description
ios::beg	The offset is calculated from the beginning of the file.
ios::end	The offset is calculated from the end of the file.
ios::cur	The offset is calculated from the current position.

WARNING! If a program has read to the end of a file, you must call the file stream object's clear member function before calling seekg or seekp. This clears the file stream object's eof flag. Otherwise, the seekg or seekp function will not work.

Statement	How It Affects the Read/Write Position
`file.seekp(32L, ios::beg);`	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
`file.seekp(-10L, ios::end);`	Sets the write position to the 10th byte from the end of the file.
`file.seekp(120L, ios::cur);`	Sets the write position to the 121st byte (byte 120) from the current position.
`file.seekg(2L, ios::beg);`	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
`file.seekg(-100L, ios::end);`	Sets the read position to the 100th byte from the end of the file.
`file.seekg(40L, ios::cur);`	Sets the read position to the 41st byte (byte 40) from the current position.
`file.seekg(0L, ios::end);`	Sets the read position to the end of the file.

The tellp and tellg Member Functions

File stream objects have two more member functions that may be used for random file access: tellp and tellg. Their purpose is to return, as a long integer, the current byte number of a file's read and write position. As you can guess, tellp returns the write position and tellg returns the read position. Assuming pos is a long integer, here is an example of the functions' usage:

```
pos = outFile.tellp();
pos = inFile.tellg();
```

One application of these functions is to determine the number of bytes that a file contains.

The following example demonstrates how to do this using the tellg function.

```
file.seekg(0L, ios::end);
numBytes = file.tellg();
cout << "The file has " << numBytes << "
bytes.\n";
```

First the seekg member function is used to move the read position to the last byte in the file. Then the tellg function is used to get the current byte number of the read position.

Rewinding a Sequential-Access File with seekg

Each time the file is reopened, its read position is located at the beginning of the file. The read position is the byte in the file that will be read with the next read operation.

Another approach is to "rewind" the file. This means moving the read position to the beginning of the file without closing and reopening it. This is accomplished with the file stream object's seekg member function to move the read position back to the beginning of the file. The following example code demonstrates.

```
dataFile.open("file.txt", ios::in); // Open the file.
// Read and process the file's contents.
dataFile.clear(); // Clear the eof flag.
dataFile.seekg(0L, ios::beg); // Rewind the read
position.
// Read and process the file's contents again.
dataFile.close(); // Close the file.
```

Notice that prior to calling the seekg member function, the clear member function is called. As previously mentioned this clears the file object's eof flag and is necessary only if the program has read to the end of the file. This approach eliminates the need to close and reopen the file each time the file's contents are processed.

Opening a File for Both Input and Output

You may perform input and output on an fstream file without closing it and reopening it. Sometimes you'll need to perform both input and output on a file without closing and reopening it. For example, consider a program that allows you to search for a record in a file and then make changes to it.

Such operations are possible with fstream objects. The ios::in and ios::out file access flags may be joined with the | operator, as shown in the following statement.

```
fstream file("data.dat", ios::in | ios::out)
```

Same operation may be accomplished with the open member func:

```
file.open("data.dat", ios::in | ios::out);
```

You may also specify the ios::binary flag if binary data is to be written to the file.

```
file.open("data.dat", ios::in | ios::out | ios::binary);
```

When an fstream file is opened with both the ios::in and ios::out flags, the file's current contents are preserved and the read/write position is initially placed at the beginning of the file. If the file does not exist, it is created.

Procedural and Object-Oriented Programming

Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered around the object. Objects are created from abstract data types that encapsulate data and functions together.

An *object* is a software entity that contains both data and procedures. The data that are contained in an object are known as the object's *attributes*. The procedures that an object performs are called *member functions*. The object is, conceptually, a self-contained unit consisting of attributes (data) and procedures (functions).

OOP addresses the problems that can result from the separation of code and data through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code that is outside the object. Only the object's member functions may directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access its member functions.

A *class* is code that specifies the attributes and member functions that a particular type of object may have. Think of a class as a "blueprint" that objects may be created from. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint.
So, a class is not an object, but it is a description of an object. Each object that is created from a class is called an *instance* of the class.

Accessors and Mutators

A member function that gets a value from a class's member variable but does not change it is known as an *accessor*. A member function that stores a value in member variable or changes the value of member variable in some other way is known as a *mutator*. Some programmers refer to mutators as *setter functions* because they set the value of an attribute, and accessors as *getter functions* because they get the value of an attribute.

Using const with Accessors

When you mark a member function as const, the const key word must appear in both the declaration and the function header. In essence, when you mark a member function as const, you are telling the compiler that the calling object is a constant. The compiler will generate an error if you inadvertently write code in the function that changes the calling object's data.

Defining Instance of a Class

Class objects must be defined after the class is declared.

```
ClassName   objectName;
Rectangle   box;
```

Defining a class object is called the *instantiation* of a class. In this statement, box is an *instance* of the Rectangle class.

Accessing an Object's Members

```
box.setWidth(12.7)
```

Just as you use the dot operator to access a structure's members, you use the dot operator to call a class's member functions. An object's member variables are not automatically initialized to 0.

Avoiding Stale Data

The area is not stored in a member variable because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data are changed, it is said that the item has become *stale*.

Pointers to Objects

```
Rectangle *rectPtr;
rectPtr = new Rectangle;

rectPtr->setWidth(10.0);
rectPtr->setLength(15.0);

delete rectPtr;
rectPtr = 0;
```

Member Variables

```
double width;
double length;
```

Member Functions

```
void setWidth(double w)
{ ... function code ...}

void setLength(double len)
{ ... function code ...}

double getWidth()
{ ... function code ...}

double getLength()
{ ... function code ...}

double getArea()
{ ... function code ...}
```

Introduction to Classes

In C++, the class is the construct primarily used to create objects. Unlike structures, the members of a class are *private* by default. In C++, a class's private members are hidden, and can be accessed only by functions that are members of the same class. A class's *public* members may be accessed by code outside the class.

Access Specifiers

C++ provides the key words private and public which you may use in class declarations. These key words are known as *access specifiers* because they specify how class members may be accessed. Notice that the access specifiers are followed by a colon (:)

```
class Rectangle
{
  private:
  double width;
  double length;

  public:
  void setWidth(double);
  void setLength(double);
  double getWidth() const;
 double getLength() const;
  double getArea() const;
};
```

Public Mem Functions

To allow access to a class's private member variables, you create public member functions that work with the private member variables. These public functions provide an interface for code outside the class to use Rectangle objects. When the key word const appears after the parentheses in a member function declaration, it specifies that the function will not change any data stored in the calling object.

There is no rule requiring you to declare private members before public members.

In each function definition, the following precedes the name of each function: Rectangle::
The two colons are called the *scope resolution operator*. When Rectangle:: appears before the name of a function in a function header, it identifies the function as a member of the Rectangle class.

```
ReturnType  ClassName::functionName ( ParameterList )
```

In the general format, *ReturnType* is the function's return type. *ClassName* is the name of the class that the function is a member of. *functionName* is the name of the member function. *ParameterList* is an optional list of parameter variable declarations.

Separating Class Specification from Implementation

Usually class declarations are stored in their own header files. Member function definitions are stored in their own .cpp files.

Program components are stored in the following fashion:

1. Class declarations are stored in their own header files.
2. The member function definitions for a class are stored in a separate .cpp file called the *class implementation* file.
3. Any program that uses the class should #include the class's header file.

The #ifndef directive is called an *include guard.*

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
  // Member declarations
  // appear here.
};
#endif
```

This directive tells the preprocessor to see if a constant named RECTANGLE_H has *not* been previously created with a #define directive.

If the RECTANGLE_H constant has *not* been defined, these lines are included in the program. Otherwise, these lines are not included in the program.

The name of the header file is enclosed in double-quote characters (" ") instead of angled brackets (< >). When you are including a C++ system header file, such as iostream, you enclose the name of the file in angled brackets. This indicates that the file is located in the compiler's *include file directory*. The include file directory is the directory or folder where all of the standard C++ header files are located. When you are including a header file that you have written, such as a class specification file, you enclose the name of the file in double-quote marks. This indicates that the file is located in the current project directory.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
  // Member declarations
  // appear here.
};
#endif
```

The first included line defines the RECTANGLE_H constant. If this file is included again, the include guard will skip its contents.

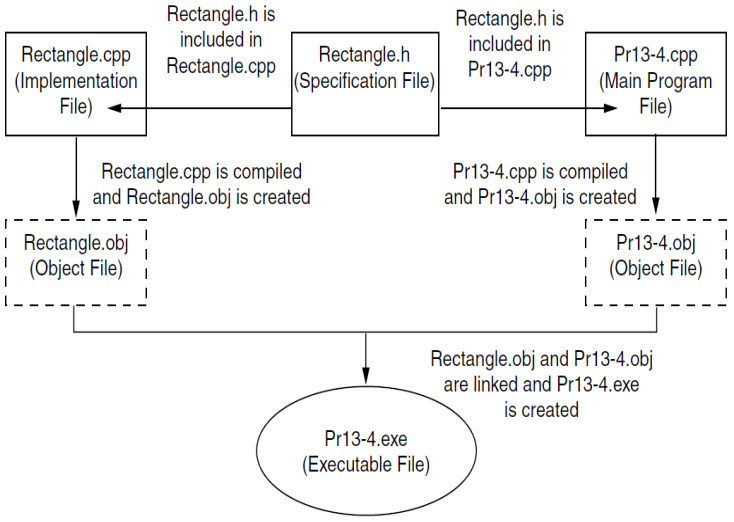
How to Create an Executeable File

To create an executable program, the following steps must be taken:

• The implementation file, Rectangle.cpp, must be compiled. Rectangle.cpp is not a complete program, so you cannot create an executable file from it alone. Instead, you compile Rectangle.cpp to an object file which contains the compiled code for the Rectangle class. This file would typically be named Rectangle.obj.

• The main program file, Pr13-4.cpp, must be compiled. This file is not a complete program either, because it does not contain any of the implementation code for the Rectangle class. So, you compile this file to an object file such as Pr13-4.obj.

• The object files, Pr13-4.obj and Rectangle.obj, are linked together to create an executable file, which would be named something like Pr13-4.exe.



Inline Member Functions When the body of a member function is written inside a class declaration, it is declared inline.

ClassName :: *ClassName* (*ParameterList*)

Constructors

A constructor is a member function that is automatically called when a class object is created. A constructor is a member function that has the same name as the class. It is automatically called when the object is created in memory, or instantiated. Notice that the constructor's function header looks different than that of a regular member function. There is no return type—not even void.

This is because constructors are not executed by explicit function calls and cannot return a value. A constructor's purpose is to initialize an object's attributes. Because the constructor executes as soon as the object is created, it can initialize the object's data members to valid values before those members are used by other code.

A *default constructor* is a constructor that takes no arguments. Like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded.

A constructor can have parameters, and can accept arguments when an object is created.

1. **Using Default Arguments with Constructors**
2. **A class can have more than one constructor.**

Destructors

A destructor is a member function that is automatically called when an object is destroyed.

Destructors are member functions with the same name as the class, preceded by a tilde character (~).For example, a common use of destructors is to free memory that was dynamically allocated by the class object.

Following points should be mentioned:

• Like constructors, destructors have no return type.

• Destructors cannot accept arguments, so they never have a parameter list.

Remember, a constructor whose parameters all have a default argument is considered a default constructor. It would be an error to create a constructor that accepts no parameters along with another constructor that has default arguments for all its parameters.

Classes may also only have one destructor. Because destructors take no arguments, the compiler has no way to distinguish different destructors.

A private member function may only be called from a function that is a member of the same class.

Arrays of Objects

You may define and work with arrays of class objects. InventoryItem inventory[40];
This statement defines an array of 40 InventoryItem objects. The name of the array is inventory, and the default constructor is called for each object in the array.

```
InventoryItem inventory[3] = {"Hammer", "Wrench", "Pliers"};
```

WARNING! If the class does not have a default constructor you must provide an initializer for each object in the array.

In summary, if you use an initializer list for class object arrays, there are three things to remember:

- If there is no default constructor you must furnish an initializer for each object in the array.
- If there are fewer initializers in the list than objects in the array, the default constructor will be called for all the remaining objects.
- If a constructor requires more than one argument, the initializer takes the form of a constructor function call.

Accessing Members of Objects in an Array

Objects in an array are accessed with subscripts, just like any other data type in an array.

```
inventory[2].setUnits(30);
```

The Unified Modeling Language (UML)

The Unified Modeling Language provides a standard method for graphically depicting an object-oriented system.

In a UML diagram you may optionally place a - character before a member name to indicate that it is private, or a + character to indicate that it is public.

NOTE: In UML notation the variable name is listed first, then the data type. This is the opposite of C++ syntax, which requires the data type to appear first. The return type of a member function can be listed in the same manner: After the function's name, place a colon followed by the return type.

Rectangle
- width : double - length : double
+ setWidth(w : double) : void + setLength(len : double) : void + getWidth() : double + getLength() : double + getArea() : double

InventoryItem
- description : char* - cost : double - units : int - createDescription(size : int, value : char*) : void
+ InventoryItem() : + InventoryItem(desc : char*) : + InventoryItem(desc : char*, c : double, u : int) : + ~InventoryItem() : + setDescription(d : char*) : void + setCost(c : double) : void + setUnits(u : int) : void + getDescription() : char* + getCost() : double + getUnits() : int

Finding the Classes

Typically, your goal is to identify the different types of real-world objects that are present in the problem, and then create classes for those types of objects within your application.
One simple and popular technique involves the following steps.

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Identifying a Class's Responsibilities

A class's *responsibilities* are

- **the things that the class is responsible for knowing**
- **the actions that the class is responsible for doing**

When you have identified the things that a class is responsible for knowing, then you have identified the class's attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its member functions.

More About Classes — Chapter # 14

Instance and Static Members

Each instance of a class has its own copies of the class's instance variables. If a member variable is declared static, however, all instances of that class have access to that variable. If a member function is declared static, it may be called without any instances of the class being defined. In object-oriented programming, member variables such as the Rectangle class's width and length members are known as *instance variables*. They are called instance variables because each instance of the class has its own copies of the variables.

Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a *static member variables* and *static member functions*. Likewise, static member functions do not operate on instance variables. Instead, they can operate only on static member variables.

Static Member Variables

When a member variable is declared with the key word static, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class.

```
int Tree::objectCount = 0;    //initialization
dataType  className::staticMember =  Val
```

This external definition statement causes the variable to be created in memory, and is required. C++ automatically stores 0 in all uninitialized static member variables.

Static Member Functions

```
static  ReturnType  FunctionName  (ParameterTypeList);
```

The following two points are important for understanding their usefulness:

• Even though static member variables are declared in a class, they are actually defined outside the class declaration. The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.

• A class's static member functions can be called before any instances of the class are created. This means that a class's static member functions can access the class's static member variables *before* any instances of the class are defined in memory.

Calls to static member functions do not use the regular notation of connecting the function name to an object name with the dot operator. Instead, static member functions are called by connecting the function name to the class name with the scope resolution operator.

NOTE: If an instance of a class with a static member function exists, the static member function can be called with the class object name and the dot operator, just like any other member function.

Friends of Classes

A friend is a function or class that is not a member of a class, but has access to the private members of the class. A *friend* function is a function that is not part of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

friend *ReturnType FunctionName* (*ParameterTypeList*)

Memberwise Assignment

The = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.

```
Rectangle box1(10.0, 10.0);
Rectangle box2 (20.0, 20.0);
box2 = box1;
```

Memberwise assignment also occurs when one object is initialized with another object's values. Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created.

Copy Constructors

A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object's data. Most of the time, the default memberwise assignment behavior in C++ is perfectly acceptable. There are instances, however, where memberwise assignment cannot be used.

```
PersonInfo person2 = person1;
```

In the statement above, person2's constructor isn't called. Instead, memberwise assignment takes place, copying each of person1's member variables into person2. This means that a separate section of memory is not allocated for person2's name member. It simply gets a copy of the address stored in person1's name member. Both pointers will point to the same address.

In this situation, either object can manipulate the string, causing the changes to show up in the other object. Likewise, one object can be destroyed, causing its destructor to be called, which frees the allocated memory. The remaining object's name pointer would still reference this section of memory, although it should no longer be used.

The solution to this problem is to create a *copy constructor* for the object. A copy constructor is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself.

When the = operator is used to initialize a PersonInfo object with the contents of another PersonInfo object, the copy constructor is called.

NOTE: C++ requires that a copy constructor's parameter be a reference object.

Using const Parameters in Copy Constructors

Because copy constructors are required to use reference parameters, they have access to their argument's data.

```
PersonInfo ( const  PersonInfo  &obj )
{ name = new char [ strlen ( obj.name ) + 1];
strcpy ( name, obj.name);
age = obj.age; }
```

The const key word ensures that the function cannot change the contents of the parameter. This will prevent you from inadvertently writing code that corrupts data.

The Default Copy Constructor

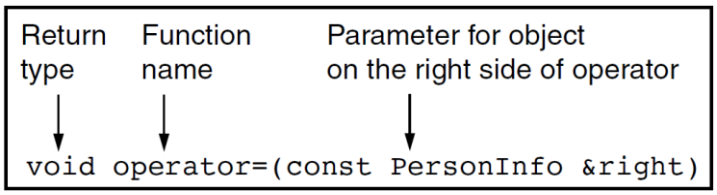
Although you may not realize it, you have seen the action of a copy constructor before. If a class doesn't have a copy constructor, C++ creates a *default copy constructor* for it. The default copy constructor performs the memberwise assignment.

Operator Overloading

C++ allows you to redefine how standard operators work when used with class objects.
Operator overloading permits you to redefine an existing operator's behavior when used with a class object.

Overloading the = Operator

Although copy constructors solve the initialization problems inherent with objects containing pointer members, they do not work with simple assignment statements. Copy constructors are just that—constructors. They are only invoked when an object is created.



The PersonInfo example declares right as a const reference for the following reasons:

- It was declared as a reference for efficiency purposes. This prevents the compiler from making a copy of the object being passed into the function.
- It was declared constant so the function will not accidentally change the contents of the argument.

In learning the mechanics of operator overloading, it is helpful to know that the following two statements do the same thing:

```
person2 = person1; // Call operator= function
person2.operator=(person1); // Call operator= function
```

NOTE: C++ allows operator functions to be called with regular function call notation, or by using the operator symbol.

The = Operator's Return Value

There is only one problem with the overloaded = operator, it has a void return type. C++'s built-in = operator allows multiple assignment statements such as:

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

In this statement, the expression b = c causes c to be assigned to b and then returns the value of c. The return value is then stored in a. If a class object's overloaded = operator is to function this way, it too must have a valid return type.

For example, the PersonInfo class's operator= function could be written as:

```
const PersonInfo operator=(const PersonInfo &right)
{    delete [] name;
     name = new char[strlen(right.name) + 1];
     strcpy(name, right.name);
     age = right.age;
     return *this;
 }
```

The data type of the operator function specifies that a const PersonInfo object is returned. Look at the last statement in the function: return *this;
This statement returns the value of a dereferenced pointer: this.

The this Pointer

The this pointer is a special built-in pointer that is available to a class's member functions. It always points to the instance of the class making the function call.
NOTE: The this pointer is passed as a hidden argument to all nonstatic member functions.

Some General Issues of Operator Overloading

1. You can change an operator's entire meaning if that's what you wish to do.

```
void operator=(const weird &right)
{ cout << right.value << endl; }
```

2. Another operator overloading issue is that you cannot change the number of operands taken by an operator. The = symbol must always be a binary operator. Likewise, ++ and -- must always be unary operators.

3. The last issue is that although you may overload most of the C++ operators, you cannot overload all of them.

+	-	*	/	%	^
>	+=	-=	*=	/=	%=
<<=	==	!=	<=	>=	&&
&	|	~	!	=	<
^=	&=	|=	<<	>>	>>=
||	++	--	->*	,	->
[]	()	new	delete		

The only operators that cannot be overloaded are
? : . .* :: sizeof

Overloading Math Operators

Overloading the Prefix ++ Operator

Overloading the Postfix ++ Operator

Overloading Relational Operators >

<

==

Cout Operator <<

Cin Operator >>

NOTE: Some compilers require you to prototype the >> and << operator functions outside the class. For this reason,

```
class  FeetInches;   // Forward Declaration
// Function Prototypes for Overloaded Stream Operators
ostream  &operator  <<  ( ostream &,  const FeetInches &);
istream  &operator  >>  ( istream &,  FeetInches &);
```

Overloading the [ ] Operator

Object Conversion

Special operator functions may be written to convert a class object to any other type.

For example, assuming distance is a FeetInches object and d is a double, the following statement would conveniently convert distance's value into a floating-point number and store it in d.

```
int  i;
double  d;
d =  i;
 i =  d;
```

For example, the value 4 feet 6 inches will be converted to 4.5. This value is stored in the local variable temp. The temp variable is then returned.

```
d = distance;  // Conversion

FeetInches::operator double()
{
    double temp = feet;
    temp += (inches / 12.0);
    return temp;
}
```

NOTE: No return type is specified in the function header. Because the function is a FeetInches-to-double conversion function, it will always return a double. Also,because the function takes no arguments, there are no parameters.

Aggregation

Aggregation occurs when a class contains an instance of another class.

For example, suppose you need an object to represent a course that you are taking in college. However, a good design principle is to separate related items into their own classes. In this example, an Instructor class could be created to hold the instructor-related data and a TextBook class could be created to hold the textbook-related data. Instances of these classes could then be used as attributes in the Course class.

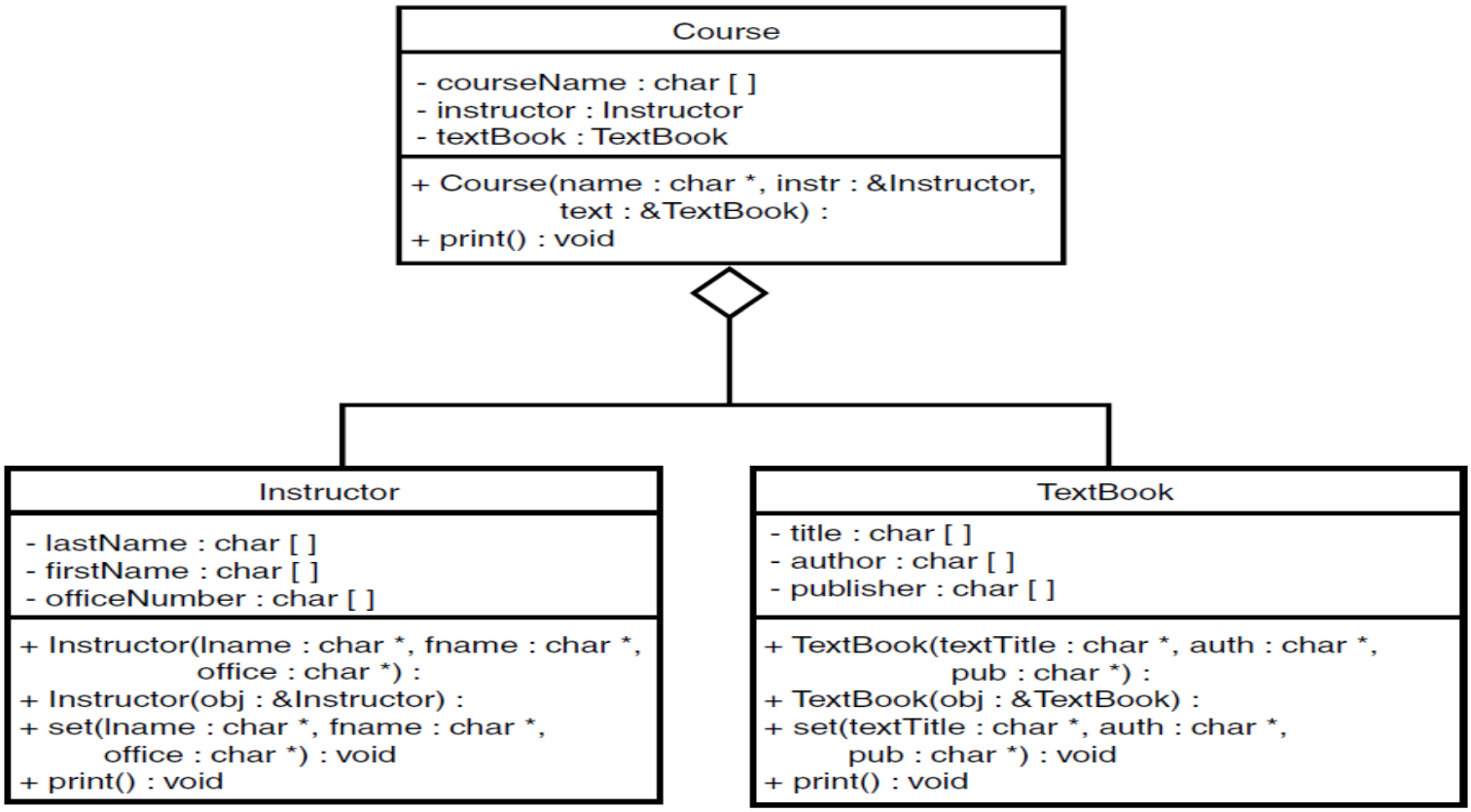
Notice that the Course class has an Instructor object and a TextBook object as member variables. Those objects are used as attributes of the Course object. Making an instance of one class an attribute of another class is called ***object aggregation***. The word *aggregate* means "a whole that is made of constituent parts."

In this example, the Course class is an aggregate class because an instance of it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a **"has a" relationship** between the classes. For example, the relationships that exist among the Course, Instructor, and TextBook classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

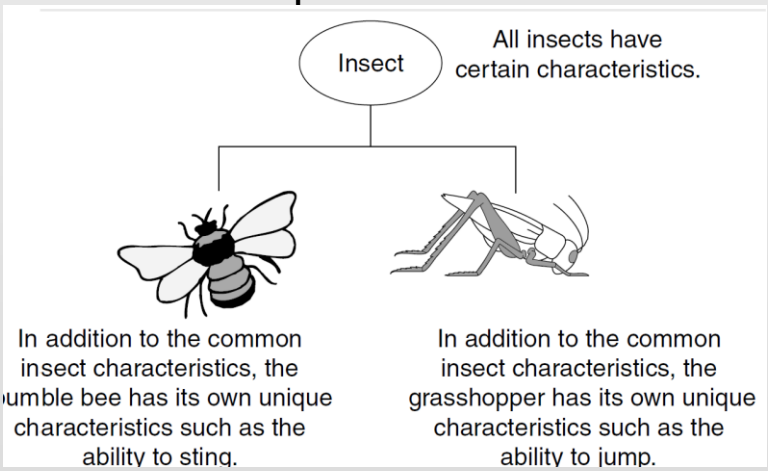
The "has a" relationship is sometimes called a *whole–part relationship* because one object is part of a greater whole.


Course
- courseName : char [ ]
- instructor : Instructor
- textBook : TextBook
+ Course(name : char *, instr : &Instructor,
text : &TextBook) :
+ print() : void
Instructor
- lastName : char [ ]
- firstName : char [ ]
- officeNumber : char [ ]
+ Instructor(lname : char *, fname : char *,
office : char *) :
+ Instructor(obj : &Instructor) :
+ set(lname : char *, fname : char *,
office : char *) : void
+ print() : void
TextBook
- title : char [ ]
- author : char [ ]
- publisher : char [ ]
+ TextBook(textTitle : char *, auth : char *,
pub : char *) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : char *, auth : char *,
pub : char *) : void
+ print() : void

What Is Inheritance?

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors & destructor) of the class it is based on.

Generalization and Specialization



Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an *"is a" relationship* between them. For example, a grasshopper *is an* insect.

When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an "is a" relationship between classes.

Inheritance involves a base class and a derived class. The *base class* is the general class and the *derived class* is the specialized class. The derived class is based on, or derived from, the base class. You can think of the base class as the parent and the derived class as the child.

```
class FinalExam : public GradedActivity
```

Class being declared (the derived class)

Base class

If we want to express the relationship between the two classes, we can say that a FinalExam *is a* GradedActivity.

The word public affects how the members of the base class are inherited by the derived class. When you create an object of a derived class, you can think of it as being built on top of an object of the base class. The members of the base class object become members of the derived class object. How the base class members appear in the derived class is determined by the base class access specification.

This means that the public members of the GradedActivity class will become public members of the FinalExam class. The private members of the GradedActivity class cannot be accessed directly by code in the FinalExam class.

Although the private members of the GradedActivity class are inherited, it's as though they are invisible to the code in the FinalExam class. They can only be accessed by the member functions of the GradedActivity class.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class.

Protected Members and Class Access

Protected members of a base class are like private members, but they may be accessed by derived classes. The base class access specification determines how private, public, and protected base class members are accessed when they are inherited by the derived classes.

Protected members of a base class are like private members, except they may be accessed by functions in a derived class. To the rest of the program, however, protected members are inaccessible.

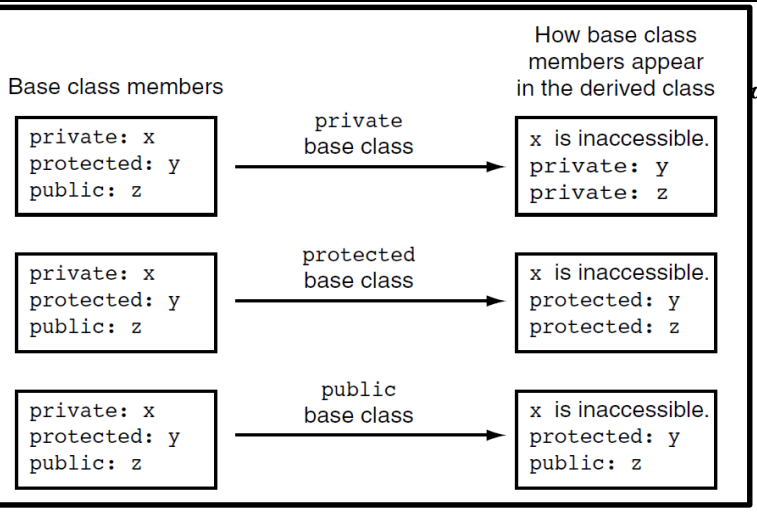
More About Base Class Access Specification

Base class access specification affects how inherited base class members are accessed. Be careful not to confuse base class access specification with member access specification. Member access specification determines how members that are *defined* within the class are accessed. Base class access specification determines how *inherited* members are accessed.

Base Class Access Specification	How Members of the Base Class Appear in the Derived Class
private	Private members of the base class are inaccessible to the derived class. Protected members of the base class become private members of the derived class. Public members of the base class become private members of the derived class.
protected	Private members of the base class are inaccessible to the derived class. Protected members of the base class become protected members of the derived class. Public members of the base class become protected members of the derived class.
public	Private members of the base class are inaccessible to the derived class. Protected members of the base class become protected members of the derived class. Public members of the base class become public members of the derived class.

NOTE: If the base class access specification is left out of a declaration, the default access specification is private. For example, in the following declaration, Grade is declared as a private base class:

```
class Test : Grade
```



Constructors and Destructors in Base and Derived Classes Pg # 887

The base class's constructor is called before the derived class's constructor. The destructors are called in reverse order, with the derived class's destructor being called first.

Passing Arguments to Base Class Constructors

What if the base class's constructor takes arguments? What if there is more than one constructor in the base class? The answer to these questions is to let the derived class constructor pass arguments to the base class constructor.

The general format of this type of constructor declaration is

```
ClassName::ClassName(ParameterList) : BaseClassName(ArgumentList)
```

Any literal value or variable that is in scope may be used as an argument to the derived class constructor. Usually, one or more of the arguments passed to the derived class constructor are, in turn, passed to the base class constructor. The values that may be used as base class constructor arguments are

- Derived class constructor parameters
- Literal values
- Global variables that are accessible to the file containing the derived class constructor definition
- Expressions involving any of these items

NOTE: If the base class has no default constructor, then the derived class must have a constructor that calls one of the base class constructors.

Redefining Base Class Functions

A base class member function may be redefined in a derived class.

Inheritance is commonly used to extend a class or give it additional capabilities. Sometimes it may be helpful to overload a base class function with a function of the same name in the derived class.

When a derived class's member function has the same name as a base class member function, it is said that the derived class function *redefines* the base class function. When an object of the derived class calls the function, it calls the derived class's version of the function.

There is a distinction between redefining a function and overloading a function. An overloaded function is one with the same name as one or more other functions, but with a different parameter list. The compiler uses the arguments passed to the function to tell which version to call. Overloading can take place with regular functions that are not members of a class. Overloading can also take place inside a class when two or more member functions *of the same class* have the same name. These member functions must have different parameter lists for the compiler to tell them apart in function calls.

Redefining happens when a derived class has a function with the same name as a base class function. The parameter lists of the two functions can be the same because the derived class function is always called by objects of the derived class type.

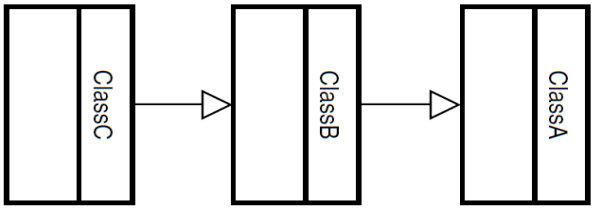
A derived class function may call a base class function of the same name using this notation, which takes this form:

BaseClassName::*functionName*(*ArgumentList*);

It is important to note that even though a derived class may redefine a function in the base class, objects that are defined of the base class type still call the base class version of the function.

Class Hierarchies

A base class can also be derived from another class.



Polymorphism & Virtual Member Functions

Polymorphism allows an object reference variable or an object pointer to reference objects of different types, and to call the correct member functions, depending upon the type of object being referenced.

```
void displayGrade(const GradedActivity &activity)
{       cout << setprecision(1) << fixed;
        cout << "The activity's numeric score is "
             << activity.getScore() << endl;
        cout << "The activity's letter grade is "
             << activity.getLetterGrade() << endl;
}
```

This function uses a const GradedActivity Reference variable as its parameter. When a GradedActivity object is passed as an argument to this function, the function calls the object's getScore and getLetterGrade member functions to display the numeric score and letter grade.

Recall that the GradedActivity class is also the base class for the FinalExam class. Because of the "is-a" relationship between a base class and a derived class, an object of the FinalExam class is not just a FinalExam object. It is also a GradedActivity object. (A final exam *is a* graded activity.) Because of this relationship, we can also pass a FinalExam object to the displayGrade function.

```
FinalExam test2(100, 25);
displayGrade(test2);
```

Because the parameter in the displayGrade function is a GradedActivity reference variable, it can reference any object that is derived from GradedActivity. A problem can occur with this type of code, however, when redefined member functions are involved.

This behavior happens because of the way C++ matches function calls with the correct function. This process is known as *binding*. C++ decides at compile time which version of the function to execute when it encounters the call to the function.

Even though we passed a PassFailActivity object to the displayGrade function, the activity parameter in the displayGrade function is a GradedActivity reference variable. Because it is of the GradedActivity type, the compiler binds the function call with the GradedActivity class's getLetterGrade function. When the program executes, it has already been determined by the compiler that the GradedActivity class's getLetterGrade function will be called. The process of matching a function call with a function at compile time is called *static binding*.

To remedy this, the getLetterGrade function can be made *virtual*. A *virtual function* is a member function that is dynamically bound to function calls. In *dynamic binding*, C++ determines which function to call at runtime, depending on the type of the object responsible for the call. If a GradedActivity object is responsible for the call, C++ will execute the GradedActivity::getLetterGrade function. If a PassFailActivity object is responsible for the call, C++ will execute the PassFailActivity:: getLetterGrade function. Virtual functions are declared by placing the key word virtual before the return type in the base class's function declaration, such as

```
virtual char getLetterGrade() const;
```

This declaration tells the compiler to expect getLetterGrade to be redefined in a derived class. The compiler does not bind calls to the function with the actual function. Instead, it allows the program to bind calls, at runtime, to the version of the function that belongs to the same class as the object responsible for the call.

NOTE: You place the virtual key word only in the function's declaration or prototype. If the function is defined outside the class, you do not place the virtual key word in the function header.

When a member function is declared virtual in a base class, any redefined versions of the function that appear in derived classes automatically become virtual. So, it is not necessary to declare the getLetterGrade function in the PassFailActivity class as virtual. It is still a good idea to declare the function virtual in the PassFailActivity class for documentation purposes.

Now that the getLetterGrade function is declared virtual, the program works properly. This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms. Program 15-11

Polymorphism Requires References or Pointers

When we call the function, we pass an object by reference. Polymorphic behavior is not possible when an object is passed by value, however.

Even though the getLetterGrade function is declared virtual, static binding still takes place because activity is not a reference variable or a pointer. Alternatively we could have used a GradedActivity pointer in the displayGrade function

Base Class Pointers

Pointers to a base class may be assigned the address of a derived class object. For example, look at the following code:

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
```

This statement dynamically allocates a PassFailExam object and assigns its address to exam, which is a GradedActivity pointer. We can then use the exam pointer to call member functions, as shown here:

```
cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Base Class Pointers and References Know Only About Base Class Members

Although a base class pointer can reference objects of any class that derives from the base class, there are limits to what the pointer can do with those objects. For example, look at the following code.

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
cout << exam->getScore() << endl; // This works.
cout << exam->getLetterGrade() << endl; // This works.
cout << exam->getPointsEach() << endl; // ERROR! Won't work!
```

In this code, exam is a GradedActivity pointer, and is assigned the address of a PassFailExam object. The GradedActivity class has only the setScore, getScore, and getLetterGrade member functions, so those are the only member functions that the exam variable knows how to execute. The last statement in this code is a call to the getPointsEach member function, which is defined in the PassFailExam class. Because the exam variable only knows about member functions in the GradedActivity class, it cannot execute this function.

The "Is-a" Relationship Does Not Work in Reverse

You cannot assign the address of a Base Calss object to a Drived Class pointer. Interestingly, the C++ compiler will let you make such an assignment if you use a type cast, as shown here:

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);.
FinalExam *fePointer = static_cast<FinalExam *>(gaPointer);
```

After this code executes, the derived class pointer fePointer will be pointing to a base class object. We can use the pointer to access members of the object, but only the members that exist.

Redefining vs. Overriding

Earlier in this chapter you learned how a derived class can redefine a base class member function. When a class redefines a virtual function, it is said that the class *overrides* the function. In C++, the difference between overriding and redefining base class functions is that overridden functions are dynamically bound, and redefined functions are statically bound. Only virtual functions can be overridden.

Virtual Destructors

When you write a class with a destructor, and that class could potentially become a base class, you should always declare the destructor virtual. This is because the compiler will perform static binding on the destructor if it is not declared virtual. This can lead to problems when a base class pointer or reference variable references a derived class object. If the derived class has its own destructor, it will not execute when the object is destroyed or goes out of scope. Only the base class destructor will execute.

A good programming practice to follow is that any class that has a virtual member function should also have a virtual destructor. If the class doesn't require a destructor, it should have a virtual destructor that performs no statements. Remember, when a base class function is declared virtual, all overridden versions of the function in derived classes automatically become virtual. Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

Abstract Base Classes & Pure Virtual Functions

An abstract base class cannot be instantiated, but other classes are derived from it. A pure virtual function is a virtual member function of a base class that must be overridden. When a class contains a pure virtual function as a member, that class becomes an abstract base class.

Sometimes it is helpful to begin a class hierarchy with an *abstract base class.* An abstract base class is not instantiated itself, but serves as a base class for other classes. The abstract base class represents the generic, or abstract, form of all the classes that are derived from it.

A class becomes an abstract base class when one or more of its member functions is a *pure virtual function.* A pure virtual function is a virtual member function declared in a manner similar to the following:

```
virtual void showInfo() = 0;
```

The = 0 notation indicates that showInfo is a pure virtual function. Pure virtual functions have no body, or definition, in the base class. They must be overridden in derived classes. Additionally, the presence of a pure virtual function in a class prevents a program from instantiating the class. The compiler will generate an error if you attempt to define an object of an abstract base class.

Remember the following points about abstract base classes and pure virtual functions:

- When a class contains a pure virtual function, it is an abstract base class.
- Pure virtual functions are declared with the = 0 notation.
- Abstract base classes cannot be instantiated.
- Pure virtual functions have no body, or definition, in the base class.
- A pure virtual function *must* be overridden at some point in a derived class in order for it to become nonabstract.

Multiple Inheritance

Multiple inheritance is when a derived class has two or more base classes.

Previously we discussed how a class may be derived from a second class that is itself derived from a third class. The series of classes establishes a chain of inheritance. In such a scheme, you might be tempted to think of the lowest class in the chain as having multiple base classes. A base class, however, should be thought of as the class that another class is directly derived from. Even though there may be several classes in a chain, each class (below the topmost class) only has one base class. Another way of combining classes is through multiple inheritance. *Multiple inheritance* is when a class has two or more base classes.

NOTE: It should be noted that multiple inheritance opens the opportunity for a derived class to have ambiguous members. That is, two base classes may have member variables or functions of the same name. In situations like these, the derived class should always redefine or override the member functions. Calls to the member functions of the appropriate base class can be performed within the derived class using the scope resolution operator(::). The derived class can also access the ambiguously named member variables of the correct base class using the scope resolution operator. If these steps aren't taken, the compiler will generate an error when it can't tell which member is being accessed.

```
class DerivedClassName : AccessSpecification BaseClassName,
                         AccessSpecification BaseClassName [, ...]
```

```
DerivedClassName(ParameterList) : BaseClassName(ArgumentList),
                                  BaseClassName( ArgumentList)[, ...]
```