Chapters 1-6 (A Review)

Review Basic Concepts of Object Oriented Programming

Identifiers, Assignment, Operators, Expressions.
Formatting output. Debugging.
Controlling execution: Decisions. Scope.
Logical Operators
Loops
Functions

Chapter 1:

Introduction to Computers and Programming

Main Hardware Component Categories:

1. Central Processing Unit (CPU)
2. Main Memory
3. Secondary Memory / Storage
4. Input Devices
5. Output Devices

Programs and Programming Languages

- A program is a set of instructions that the computer follows to perform a task
- We start with an *algorithm*, which is a set of well-defined steps.

Example Algorithm for Calculating Gross Pay

1. Display a message on the screen asking "How many hours did you work?"
2. Wait for the user to enter the number of hours worked. Once the user enters a number, store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Wait for the user to enter an hourly pay rate. Once the user enters a number, store it in memory.
5. Multiply the number of hours by the amount paid per hour, and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in Step 5.

From a High-Level Program to an Executable File

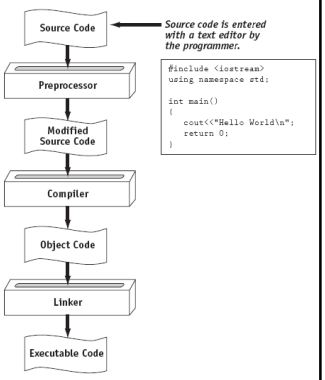
a) Create file containing the program with a text editor.

b) Run preprocessor to convert source file directives to source code program statements.

c) Run compiler to convert source program into machine instructions.

d) Run linker to connect hardware-specific code to machine instructions, producing an executable file.

- Steps b–d are often performed by a single command or button click.
- Errors detected at any step will prevent execution of following steps.



What is a Program Made of?

- Common elements in programming languages:
  - Key Words
    - Also known as reserved words
    - Have a special meaning in C++
    - Can not be used for any other purpose
    - Key words in the Program 1-1: `using`, `namespace`, `int`, `double`, and `return`
  - Programmer-Defined Identifiers
    - Names made up by the programmer
    - Not part of the C++ language
    - Used to represent various things: variables (memory locations), functions, etc.
    - In Program 1-1: `hours`, `rate`, and `pay`.
  - Operators
    - Used to perform operations on data
    - Many types of operators:
      - Arithmetic - ex: `+`, `-`, `*`, `/`
      - Assignment – ex: `=`
  - Punctuation
    - Characters that mark the end of a statement, or that separate items in a list
    - In Program 1-1: `,` and `;`
  - Syntax
    - The rules of grammar that must be followed when writing a program
    - Controls the use of key words, operators, programmer-defined symbols, and punctuation

Variables

- A variable is a named storage location in the computer's memory for holding a piece of data.
- To create a variable in a program you must write a variable definition (also called a variable declaration)
- A variable holds a specific type of data.
- The variable definition specifies the type of data a variable can hold, and the variable name.

Variables of the same type can be defined

- On separate lines:

```
int length;
int width;
unsigned int area;
```

- On the same line:

```
int length, width;
unsigned int area;
```

Variables of different types must be in different definitions

Input, Processing, and Output

Three steps that a program typically performs:

1) **Gather input data:**
   - from keyboard
   - from files on disk drives
2) **Process the input data**
3) **Display the results as output:**
   - send it to the screen
   - write to a file

The Programming Process

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.
4. Check the model for logical errors.
5. Type the code, save it, and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

Procedural and Object-Oriented Programming

- Procedural programming: focus is on the process. Procedures/functions are written to process data.
- Object-Oriented programming: focus is on objects, which contain data and the means to manipulate the data. Messages sent to objects to perform operations.

Chapter 2:

Introduction to C++

The Parts of a C++ Program

```
// sample C++ program
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, there!";
    return 0;
}
```

comment

preprocessor directive

which namespace to use

beginning of function named `main`

beginning of block for `main`

output statement

string literal

Send `0` to operating system

end of block for `main`

Special Characters

Character	Name	Meaning
//	Double slash	Beginning of a comment
#	Pound sign	Beginning of preprocessor directive
< >	Open/close brackets	Enclose filename in #include
( )	Open/close parentheses	Used when naming a function
{ }	Open/close brace	Encloses a group of statements
" "	Open/close quotation marks	Encloses string of characters
;	Semicolon	End of a programming statement

The cout Object

- Displays output on the computer screen
- You use the stream insertion operator << to send output to cout:

```
cout << "Programming is fun!";
```

- Can be used to send more than one item to cout:

```
cout << "Hello " << "there!";
```

Or:

```
cout << "Hello ";
cout << "there!";
```

The endl Manipulator

- You can use the **endl** manipulator to start a new line of output. This will produce two lines of output:

```
cout << "Programming is" << endl;
cout << "fun!";
```

The \n Escape Sequence

- You can also use the **\n** escape sequence to start a new line of output. This will produce two lines of output:

```
cout << "Programming is\n";
cout << "fun!";
```

\n is INSIDE the string.

The `#include` Directive

- Inserts the contents of another file into the program
- This is a preprocessor directive, not part of C++ language
- `#include` lines not seen by compiler
- Do not place a semicolon at end of `#include` line

Literals

- Literal: a value that is written into a program's code.

```
"hello, there" (string literal)
12  (integer literal)
```

Integer Literals

- Integer literals are stored in memory as `int`s by default
- To store an integer constant in a long memory location, put '`L`' at the end of the number: `1234L`
- To store an integer constant in a long long memory location, put '`LL`' at the end of the number: `324LL`
- Constants that begin with '`0`' (zero) are base 8: `075`
- Constants that begin with '`0x`' are base 16: `0x75A`

Identifiers

- An identifier is a programmer-defined name for some part of a program: variables, functions, etc.
- The first character of an identifier must be an alphabetic character or and underscore (_),
- After the first character you may use alphabetic characters, numbers, or underscore characters.
- Upper- and lowercase characters are distinct

IDENTIFIER	VALID?	REASON IF INVALID
`totalSales`	Yes	
`total_Sales`	Yes	
`total.Sales`	No	Cannot contain `.`
`4thQtrSales`	No	Cannot begin with digit
`totalSale$`	No	Cannot contain `$`

The `char` Data Type

- Used to hold characters or very small integer values
- Usually 1 byte of memory
- Numeric value of character from the character set is stored in memory:
- Character literals must be enclosed in single quote marks.

CODE:
```
char letter;
letter = 'C';
```

MEMORY:
letter

67

Character Strings

- A series of characters in consecutive memory locations:
  `"Hello"`
- Stored with the null terminator, `\0`, at the end:
- Comprised of the characters between the " "

H	e	l	l	o	\0

The C++ `string` Class

- Special data type supports working with strings

```
#include <string>
```

- Can define `string` variables in programs:

```
string firstName, lastName;
```

- Can receive values with assignment operator:

```
firstName = "George";
lastName = "Washington";
```

- Can be displayed via `cout`

```
cout << firstName << " " << lastName;
```

Floating-Point Data Types

- The floating-point data types are:
  `float`
  `double`
  `long double`
- They can hold real numbers such as:

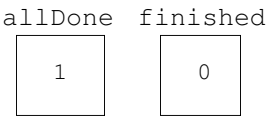
```
12.45      -3.8
```

- Stored in a form similar to scientific notation
- All floating-point numbers are signed

The `bool` Data Type

- Represents values that are `true` or `false`
- `bool` variables are stored as small integers
- `false` is represented by 0, `true` by 1:

```
bool allDone = true;
bool finished = false;
```



Determining the Size of a Data Type

- The `sizeof` operator gives the size of any data type or variable:

```
double amount;
cout << "A double is stored in "
     << sizeof(double) << "bytes\n";
cout << "Variable amount is stored in "
     << sizeof(amount)
    << "bytes\n";
```

Variable Assignments and Initialization

- An assignment statement uses the = operator to store a value in a variable.

```
item = 12;
```

-
```
// ERROR!
 12 = item;
```

- This statement assigns the value 12 to the `item` variable.
- To initialize a variable means to assign it a value when it is defined:

```
int length = 12;
```

- Can initialize some or all variables:

```
int length = 12, width = 5, area;
```

Declaring Variables With the `auto` Key Word

- C++ 11 introduces an alternative way to define variables, using the `auto` key word and an initialization value. Here is an example:

```
auto amount = 100;
```
int

- The `auto` key word tells the compiler to determine the variable's data type from the initialization value.

```
auto interestRate= 12.0;
```
double

```
auto stockCode = 'D';
```
char

```
auto customerNum = 459L;
```
long

Scope

- The scope of a variable: the part of the program in which the variable can be accessed
- A variable cannot be used before it is defined

Arithmetic Operators

- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators:
  - unary (1 operand) `-5`
  - binary (2 operands) `13 - 7`

SYMBOL	OPERATION	EXAMPLE	VALUE OF `ans`
+	addition	`ans = 7 + 3;`	10
-	subtraction	`ans = 7 - 3;`	4
*	multiplication	`ans = 7 * 3;`	21
/	division	`ans = 7 / 3;`	2
%	modulus	`ans = 7 % 3;`	1

  - ternary (3 operands) `exp1 ? exp2 : exp3`
    - `If expr1 is true then you execute expr2 else expr3`

A Closer Look at the / Operator

- / (division) operator performs integer division if both operands are integers

```
cout << 13 / 5;     // displays 2
cout << 91 / 7;     // displays 13
```

- If either operand is floating point, the result is floating point

```
cout << 13 / 5.0;  // displays 2.6
cout << 91.0 / 7;  // displays 13.0
```

A Closer Look at the % Operator

- % (modulus) operator computes the remainder resulting from integer division

```
cout << 13 % 5;    // displays 3
```

- % requires integers for both operands

```
cout << 13 % 5.0; // error
```

Comments

- Used to document parts of the program
- Intended for persons reading the source code of the program:
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Are ignored by the compiler
- Single Line begin with `//` through to the end of line:

```
int length = 12; // length in inches
```

- Multiple Line begin with `/*`, end with `*/ and` Can span multiple lines:

```
/* this is a multi-line
   comment
*/
```

Named Constants

- Named constant (constant variable): variable whose content cannot be changed during program execution
- Used for representing constant values with descriptive names:

```
const double TAX_RATE = 0.0675;
const int NUM_STATES = 50;
```

- Often named in uppercase letters

Programming Style

- The visual organization of the source code
- Includes the use of spaces, tabs, and blank lines
- Does not affect the syntax of the program
- Affects the readability of the source code

Chapter 3:

Expressions and Interactivity

The `cin` Object

- Standard input object
- Like `cout`, requires `iostream` file
- Used to read input from keyboard
- Information retrieved from `cin` with `>>`
- Input is stored in one or more variables
- **`cin`** converts data to the type that matches the variable:

```
int height;
cout << "How tall is the room? ";
cin >> height;
```

- Can be used to input more than one value:

```
cin >> height >> width;
```

- Multiple values from keyboard must be separated by spaces
- Order is important: first value entered goes to first variable, etc.

Mathematical Expressions

- Can create complex expressions using multiple mathematical operators
- An expression can be a literal, a variable, or a mathematical combination of constants and variables
- Can be used in assignment, `cout`, other statements:

```
area = 2 * PI * radius;
cout << "border is: " << 2*(l+w);
```

Order of Operations

In an expression with more than one operator, evaluate in this order:

- – (unary negation), in order, left to right
- * / %, in order, left to right
- + –, in order, left to right

In the expression `2 + 2 * 2 – 2`

Associativity of Operators

- – (unary negation) associates right to left
- *, /, %, +, – associate right to left
- parentheses ( ) can be used to override the order of operations:

```
 2 + 2  *  2 – 2  = 4
(2 + 2) *  2 – 2  = 6
 2 + 2  * (2 – 2) = 2
(2 + 2) * (2 – 2) = 0
```

Algebraic Expressions

- Multiplication requires an operator:
  $Area=lw$ is written as `Area = l * w;`
- There is no exponentiation operator:
  $Area=s^2$ is written as `Area = pow(s, 2);`
- Parentheses may be needed to maintain order of operations:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$ is written as `m = (y2-y1) /(x2-x1);`

When You Mix Apples with Oranges: Type Conversion

- Operations are performed between operands of the same type.
- If not of the same type, C++ will convert one to be the type of the other
- This can impact the results of calculations.

Type Coercion

- Type Coercion: automatic conversion of an operand to another data type
- Promotion: convert to a higher type
- Demotion: convert to a lower type
- `char`, `short`, `unsigned short` automatically promoted to `int`
- When operating on values of different data types, the lower one is promoted to the type of the higher one.
- When using the = operator, the type of expression on right will be converted to type of variable on left

Highest:
```
long double
double
float
unsigned long
long
unsigned int
int
```
Lowest:
Ranked by largest number they can hold

Overflow and Underflow

- Occurs when assigning a value that is too large (overflow) or too small (underflow) to be held in a variable
- Variable contains value that is 'wrapped around' set of possible values
- Different systems may display a warning/error message, stop the program, or continue execution using the incorrect value

Type Casting

- Used for manual data type conversion
- Useful for floating point division using ints:

```
double m;
m = static_cast<double>(y2-y1)/(x2-x1);
```

- Useful to see `int` value of a `char` variable:

```
char ch = 'C';
cout << ch << " is "
     << static_cast<int>(ch);
```

Multiple Assignment and Combined Assignment

- The = can be used to assign a value to multiple variables:

```
x = y = z = 5;
```

- Value of = is the value that is assigned
- Associates right to left:

```
x = (y = (z = 5));
```

- Look at the following statements:

```
sum = sum + 1;
sum += 1;
```

They both add 1 to the variable **sum**.

Formatting Output & Stream Manipulators

- Can control how output displays for numeric, string data:
  - Size
  - position
  - number of digits
- Requires `iomanip` header file
- Used to control how an output field is displayed
- Some affect just the next value displayed:
  - `setw(x)`: print in a field at least `x` spaces wide. Use more spaces if field is not wide enough
- Some affect values until changed again:
  - `fixed`: use decimal notation for floating-point values
  - `setprecision(x)`:
    - With `fixed`, print floating-point value using `x` digits after the decimal.
    - Without `fixed`, print floating-point value using x significant digits
  - `showpoint`: always print decimal for floating-point values

Working with Characters and `string` Objects

- Using `cin` with the >> operator to input strings can cause problems:
- It passes over and ignores any leading *whitespace characters (spaces, tabs, or line breaks)*
- To work around this problem, you can use a C++ function named `getline`.

Working with Characters and `string` Objects

- To read a single character:
  - Use `cin`:

```
char ch;
cout << "Strike any key to continue";
cin >> ch;
```

Problem: will skip over blanks, tabs, <CR>

  - Use `cin.get()`:

```
cin.get(ch);
```

Will read the next character entered, even whitespace

Working with Characters and `string` Objects

- Mixing `cin >>` and `cin.get()` in the same program can cause input errors that are hard to detect

- To skip over unneeded characters that are still in the keyboard buffer, use `cin.ignore()`:

```
cin.ignore(); // skip next char
cin.ignore(10, '\n'); // skip the next
                      // 10 char. or until a '\n'
```

`string` Member Functions and Operators

- To find the length of a string:

```
string state = "Texas";
int size = state.length();
```

- To concatenate (join) multiple strings:

```
greeting2 = greeting1 + name1;
greeting1 = greeting1 + name2;
```

Or using the **+=** combined assignment operator:

```
greeting1 += name2;
```

More Mathematical Library Functions

- Require `cmath` header file
- Take `double` as input, return a `double`
- Commonly used functions:

`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`sqrt`	Square root
`log`	Natural (e) log
`abs`	Absolute value (takes and returns an int)

Chapter 4:

Making Decisions

Relational Operators

- Used to compare numbers to determine relative order
- Operators:

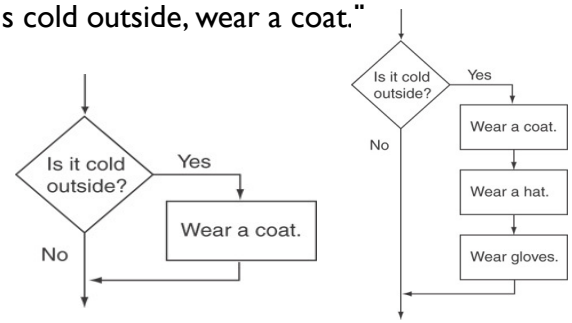
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Relational Expressions

- Boolean expressions – `true` or `false`
- Examples:
  `12 > 5` is `true`
  `7 <= 5` is `false`
  if `x` is 10, then
  `x == 10` is `true`,
  `x != 8` is `true`, and
  `x == 8` is `false`
- Can be assigned to a variable:
  ```
  result = x <= y;
  ```
- Assigns `0` for `false`, `1` for `true`
- Do not confuse `=` and `==`

The `if` Statement

- Allows statements to be conditionally executed or skipped over
- Models the way we mentally evaluate situations:
  - "If it is raining, take an umbrella."
  - "If it is cold outside, wear a coat."



The if Statement-What Happens

To evaluate:

```
if (expression)
     statement;
```

- If the *expression* is `true`, then *statement* is executed.
- If the *expression* is `false`, then *statement* is skipped.

- Do not place `;` after `(expression)`
- Place *statement*`;` on a separate line after `(expression)`, indented:

```
if (score > 90)
     grade = 'A';
```

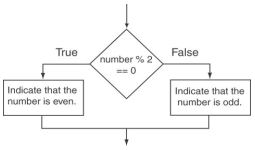
- Be careful testing `float`s and `double`s for equality
- `0` is `false`; any other value is `true`

The `if/else` statement

- Provides two possible paths of execution
- Performs one statement or block if the *expression* is true, otherwise performs another statement or block.
- General Format:

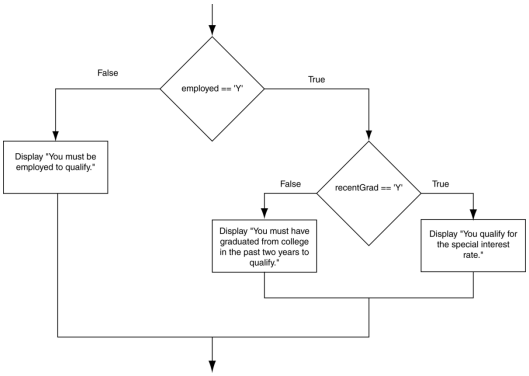
```
if (expression)
         statement1;  // or block
else
         statement2;  // or block
```

- If the *expression* is `true`, then *statement1* is executed and *statement2* is skipped.
- If the *expression* is *false*, then *statement1* is skipped and *statement2* is executed.



Nested `if` Statements

- An `if` statement that is nested inside another `if` statement
- Nested `if` statements can be used to test more than one condition



Flags

- Variable that signals a condition
- Usually implemented as a `bool` variable
- Can also be an integer
  - The value `0` is considered `false`
  - Any nonzero value is considered `true`
- As with other variables in functions, must be assigned an initial value before it is used

Logical Operators

- Used to create relational expressions from other relational expressions
- Operators, meaning, and explanation:

`&&`	AND	New relational expression is true if both expressions are true
`||`	OR	New relational expression is true if either expression is true
`!`	NOT	Reverses the value of an expression – true expression becomes false, and false becomes true

```
int x = 12, y = 5, z = -4;
```

`(x > y) && (y > z)`	`true`
`(x > y) && (z > y)`	`false`
`(x <= z) || (y == z)`	`false`
`(x <= z) || (y != z)`	`true`
`!(x >= z)`	`false`

Logical Operator-Notes

- `!` has highest precedence, followed by `&&`, then `||`
- If the value of an expression can be determined by evaluating just the sub-expression on left side of a logical operator, then the sub-expression on the right side will not be evaluated (*short circuit evaluation*)

Menus

- Menu-driven program: program execution controlled by user selecting from a list of actions
- Menu: list of choices on the screen
- Menus can be implemented using `if/else if` statements

- Used to test to see if a value falls **inside** a range:

```
if (grade >= 0 && grade <= 100)
   cout << "Valid grade";
```

- Can also test to see if value falls **outside** of range:

```
if (grade <= 0 || grade >= 100)
   cout << "Invalid grade";
```

- Cannot use mathematical notation:

```
if (0 <= grade <= 100) //doesn't work!
```

Validating User Input

- Input validation: inspecting input data to determine whether it is acceptable
- Bad output will be produced from bad input
- Can perform various tests:
  - Range
  - Reasonableness
  - Valid menu choice
  - Divide by zero

Comparing Characters

- Characters are compared using their ASCII values
- 'A' < 'B'
  - The ASCII value of 'A' (65) is less than the ASCII value of 'B'(66)
- '1' < '2'
  - The ASCII value of '1' (49) is less than the ASCI value of '2' (50)
- Lowercase letters have higher ASCII codes than uppercase letters, so 'a' > 'Z'

Comparing `string` Objects

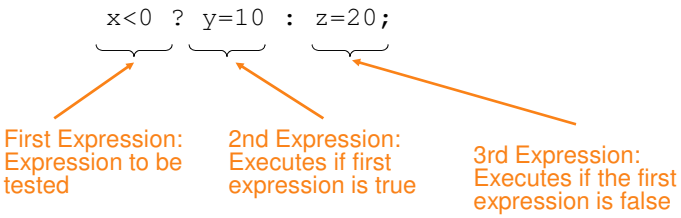
- Like characters, strings are compared using their ASCII values

```
string name1 = "Mary";
string name2 = "Mark";

name1 > name2   // true
name1 <= name2 // false
name1 != name2  // true

name1 < "Mary Jane" // true
```

The characters in each string must match before they are equal

The Conditional Operator

- Can use to create short `if/else` statements
- Format: `expr ? expr : expr;`



The value of a conditional expression is:

The value of the second expression if the first expression is true
The value of the third expression if the first expression is false

The `switch` Statement

- Used to select among statements from several alternatives
- In some cases, can be used instead of `if/else if` statements

```
switch (expression) //integer
{
  case exp1: statement1;
  case exp2: statement2;
  ...
  case expn: statementn;
  default:   statementn+1;
}
```

`switch` Statement Requirements

1) *`expression`* must be an integer variable or an expression that evaluates to an integer value
2) *`exp`**`1`*** through *`exp`**`n`*** must be constant integer expressions or literals, and must be unique in the `switch` statement
3) `default` is optional but recommended
4) How To:
   - *`expression`* is evaluated
   - The value of *`expression`* is compared against *`exp`**`1`*** through *`exp`**`n`***.
   - If *`expression`* matches value *`exp`**`i`***, the program branches to the statement following *`exp`**`i`*** and continues to the end of the `switch`
   - If no matching value is found, the program branches to the statement after `default`:

`break` Statement

- Used to exit a `switch` statement
- If it is left out, the program "falls through" the remaining statements in the `switch` statement

Using `switch` in Menu Systems

- `switch` statement is a natural choice for menu-driven program:
  - display the menu
  - then, get the user's menu selection
  - use user input as `expression` in `switch` statement
  - use menu choices as *`expr`* in `case` statements

More About Blocks and Scope

- Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Usually defined at beginning of function
- May be defined close to first use

Variables with the Same Name

- Variables defined inside { } have local or block scope
- When inside a block within another block, can define variables with the same name as in the outer block.
  - When in inner block, outer definition is not available
  - Not a good idea

Chapter 5:

Loops and Files

Deciding Which Loop to Use

- The `while` loop is a conditional pretest loop
  - Iterates as long as a certain condition exits
  - Validating input
  - Reading lists of data terminated by a sentinel
- The `do-while` loop is a conditional posttest loop
  - Always iterates at least once
  - Repeating a menu
- The `for` loop is a pretest loop
  - Built-in expressions for initializing, testing, and updating
  - Situations where the exact number of iterations is known

The Increment and Decrement Operators

- ++ is the increment operator.
  It adds one to a variable.
  `val++;` is the same as `val = val + 1;`
- ++ can be used before (prefix) or after (postfix) a variable:

```
++val;      val++;
```

- -- is the decrement operator.
  It subtracts one from a variable.
  `val--;` is the same as `val = val - 1;`
- -- can be also used before (prefix) or after (postfix) a variable:

```
--val;      val--;
```

Prefix vs. Postfix

- `++` and `--` operators can be used in complex statements and expressions
- In prefix mode (`++val, --val`) the operator increments or decrements, *then* returns the value of the variable
- In postfix mode (`val++, val--`) the operator returns the value of the variable, *then* increments or decrements
- Can be used in expressions:

```
result = num1++ + --num2;
```

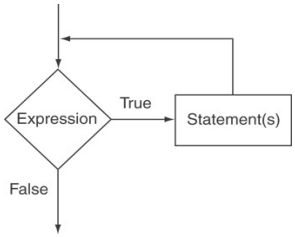
- Must be applied to something that has a location in memory.
- Can be used in relational expressions:

```
if (++num > limit)
```

pre- and post-operations will cause different comparisons

```
int num, val = 12;

          cout << val++; // displays 12,
                         // val is now 13;
          cout << ++val; // sets val to 14,
                         // then displays it
          num = --val;   // sets val to 13,
                         // stores 13 in num
          num = val--;   // stores 13 in num,
                         // sets val to 12
```

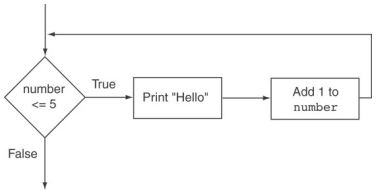
The `while` Loop



- Loop: a control structure that causes a statement or statements to repeat
- General format of the `while` loop:

```
while (expression)
      statement;
```

- *`statement`*`;` can also be a block of statements enclosed in `{}`
- *`expression`* is evaluated
  - if `true`, then *`statement`* is executed, and *`expression`* is evaluated again
  - if `false`, then the loop is finished and program statements following *`statement`* execute

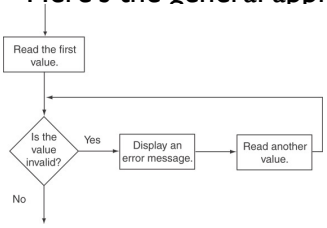


Watch Out for Infinite Loops

- The loop must contain code to make *expression* become `false`
- Otherwise, the loop will have no way of stopping
- Such a loop is called an *infinite loop*, because it will repeat an infinite number of times

Using the `while` Loop for Input Validation

- Input validation is the process of inspecting data that is given to the program as input and determining whether it is valid.
- The while loop can be used to create input routines that reject invalid data, and repeat until valid data is entered.
- Here's the general approach, in pseudocode:



Read an item of input.
While the input is invalid
    Display an error message.
    Read the input again.
End While

Counters

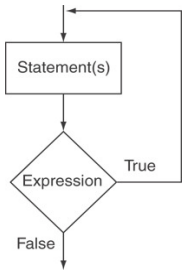
- Counter: a variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (also known as the *loop control variable*)
- Must be initialized before entering loop

The `do-while` Loop

- `do-while`: a posttest loop – execute the loop, then test the `expression` Loop always executes at least once
- General Format:

```
do
   statement;  // or block in { }
while (expression);
```

- Note that a semicolon is required after `(expression)`
- Execution continues as long as *`expression`* is `true`, stops repetition when *`expression`* becomes `false`
- Useful in menu-driven programs to bring user back to menu to make another choice



The `for` Loop

- Useful for counter-controlled loop
- General Format:

```
for(initialization; test; update)
     statement; // or block in { }
```

- No semicolon after the `update` expression or after the `)`

```
for(initialization; test; update)
          statement; // or block in { }
```

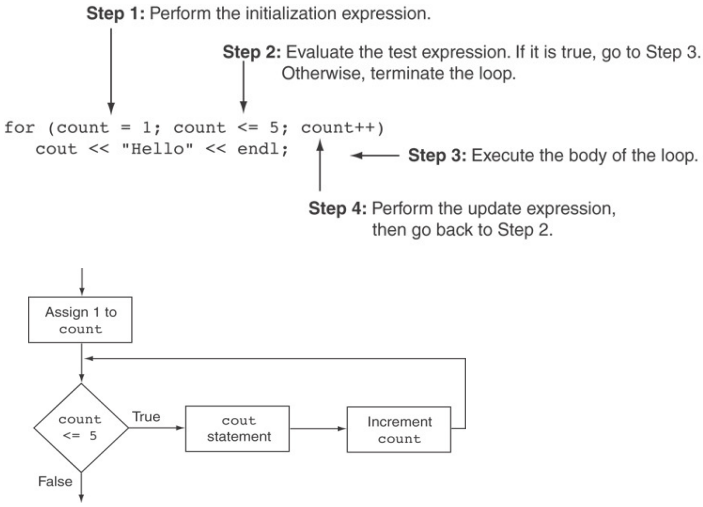
- Perform *initialization*
- Evaluate *test* expression
  - If `true`, execute *statement*
  - If `false`, terminate loop execution
- Execute *update*, then re-evaluate *test* expression

The `for` Loop is a Pretest Loop

- The for loop tests its test expression before each iteration, so it is a pretest loop.
- The following loop will never iterate:

```
for (count = 11; count <= 10; count++)
   cout << "Hello" << endl;
```

A For Loop Example



When to Use the `for` Loop

- In any situation that clearly requires
  - an initialization
  - a false condition to stop the loop
  - an update to occur at the end of each iteration

Sentinels

- sentinel: value in a list of values that indicates end of data
- Special value that cannot be confused with a valid value, e.g., `-999` for a test score
- Used to terminate input when user may not know how many values will be entered

Nested Loops

- A nested loop is a loop inside the body of another loop
- Inner (inside), outer (outside) loops:

```
for (row=1; row<=3; row++)  //outer
  for (col=1; col<=3; col++)//inner
      cout << row * col << endl;
```

- Inner loop goes through all repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops.

Using Files for Data Storage

- Can use files instead of keyboard, monitor screen for program input, output
- Allows data to be retained between program runs
- Steps:
  - *Open* the file
  - *Use* the file (read from, write to, or both)
  - *Close* the file

Files: What is Needed

- Use `fstream` header file for file access
- File stream types:
  - `ifstream` for input from a file
  - `ofstream` for output to a file
  - `fstream` for input from or output to a file
- Define file stream objects:

```
ifstream infile;
ofstream outfile;
```

Opening Files

- Create a link between file name (outside the program) and file stream object (inside the program)
- Use the `open` member function:

```
infile.open("inventory.dat");
outfile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing file will be erased first
- Input file must exist for `open` to work

Testing for File Open Errors

- Can test a file stream object to detect if an open operation failed:

```
infile.open("test.txt");
if (!infile)
{
   cout << "File open failure!";
}
```

- Can also use the `fail` member function

Using Files

- Can use output file object and << to send data to a file:

```
outfile << "Inventory report";
```

- Can use input file object and >> to copy data from file to variables:

```
infile >> partNum;
infile >> qtyInStock >>
qtyOnOrder;
```

Using Loops to Process Files

- The stream extraction operator >> returns `true` when a value was successfully read, `false` otherwise

- Can be tested in a `while` loop to continue execution as long as values are read from the file:

```
while (inputFile >> number) ...
```

Closing Files

- Use the `close` member function:

```
infile.close();
outfile.close();
```

- Don't wait for operating system to close files at program end:
  - may be limit on number of open files
  - may be buffered output data waiting to send to file

Letting the User Specify a Filename

- In many cases, you will want the user to specify the name of a file for the program to open.
- In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function.

Breaking Out of a Loop

- Can use `break` to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand and debug
- When used in an inner loop, terminates that loop only and goes back to outer loop

The `continue` Statement

- Can use `continue` to go to end of loop and prepare for next repetition
  - `while`, `do-while` loops: go to test, repeat loop if test passes
  - `for` loop: perform update step, then test, then repeat loop if test passes
- Use sparingly – like `break`, can make program logic hard to follow

Chapter 6:

Functions

Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules
- Function: a collection of statements to perform a task
- Motivation for modular programming:
  - Improves maintainability of programs
  - Simplifies the process of writing programs

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
   statement;
   statement;
   statement;
}
```
main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```
function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```
function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```
function 4

Defining and Calling Functions

- Function call: statement causes a function to execute
- Function definition: statements that make up a function
- Note: The line that reads `int main()` is the *function header*.

Return type
Function name
Parameter list (This one is empty)
Function body

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

Function Definition

- Definition includes:
  - return type: data type of the value that function returns to the part of the program that called it
  - name: name of the function. Function names follow same rules as variables
  - parameter list: variables containing values passed to the function
  - body: statements that perform the function's task, enclosed in `{}`

Function Return Type

- If a function returns a value, the type of the value must be indicated:

```
int main()
```

- If a function does not return a value, its return type is `void`:

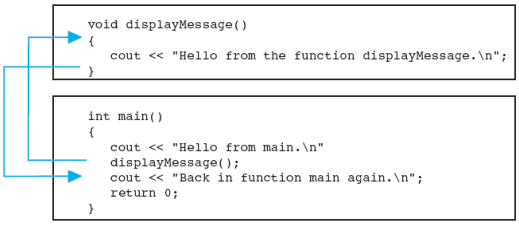
```
void printHeading()
{
        cout << "Monthly Sales\n";
}
```

Calling a Function

- To call a function, use the function name followed by `()` and `;`

```
printHeading();
```

- When called, program executes the body of the called function
- After the function terminates, execution resumes in the calling function at point of call.
- `main` can call any number of functions
- Functions can call other functions
- Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter



```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}

int main()
{
   cout << "Hello from main.\n"
   displayMessage();
   cout << "Back in function main again.\n";
   return 0;
}
```

Function Prototypes

- Ways to notify the compiler about a function before a call to the function:
  - Place function definition before calling function's definition
  - Use a function prototype (function declaration) – like the function definition without the body
    - Header: `void printHeading()`
- Prototype: `void printHeading();` Place prototypes near top of program
- Program must include either prototype or full function definition before any call to the function – compiler error otherwise
- When using prototypes, can place function definitions in any order in source file

Sending Data into a Function

- Can pass values into a function at time of call:

```
c = pow(a, b);
```

- Values passed to function are arguments
- Variables in a function that hold the values passed as arguments are parameters

A Function with a Parameter Variable

```
void displayValue(int num)
{
   cout << "The value is " << num << endl;
}
```

The integer variable `num` is a parameter. It accepts any integer value passed to the function.

- A parameter can also be called a formal parameter or a formal argument
- An argument can also be called an actual parameter or an actual argument

Parameters, Prototypes, and Function Headers

- For each function argument,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```
void evenOrOdd(int);   //prototype
void evenOrOdd(int num) //header
evenOrOdd(val);        //call
```

Function Call Notes

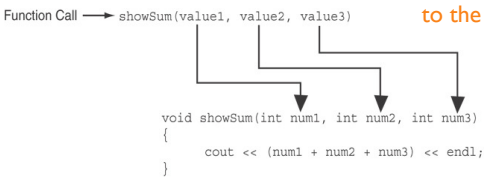
- Value of argument is copied into parameter when the function is called
- A parameter's scope is the function which uses it
- Function can have multiple parameters
- There must be a data type listed in the prototype () and an argument declaration in the function header () for each parameter
- Arguments will be promoted/demoted as necessary to match parameters

Passing Multiple Arguments

When calling a function and passing multiple arguments:

- the number of arguments in the call must match the prototype and definition
- the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

The function call passes value1, value2, and value3 as a arguments to the function.



Passing Data by Value

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.
- Changes to the parameter in the function do not affect the value of the argument
- Example: `int val=5;`
  `evenOrOdd(val);`

- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

The `return` Statement

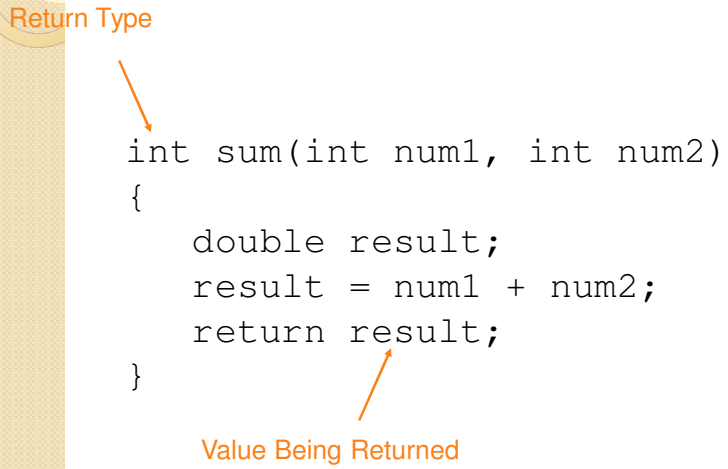
- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last `}`

Returning a Value From a Function

- A function can return a value back to the statement that called the function.
- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:

```
int sum(int num1, int num2)
{
  double result;
  result = num1 + num2;
  return result;
}
```

A Value-Returning Function



Returning a Value From a Function

- The prototype and the definition must indicate the data type of return value (not `void`)
- Calling function should use return value:
  - assign it to a variable
  - send it to `cout`
  - use it in an expression

Returning a Boolean Value

- Function can return `true` or `false`
- Declare return type in function prototype and heading as `bool`
- Function body must contain `return` statement(s) that return `true` or `false`
- Calling function can use return value in a relational expression

Local and Global Variables

- Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.
- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.
- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.
- Local variables are not automatically initialized. They must be initialized by programmer.

Initializing Local and Global Variables

- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined
- You should avoid using global variables because they make programs difficult to debug.
- 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.
- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.
- `static` local variables retain their contents between function calls.
- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Default Arguments

- A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.
- Must be a constant declared in prototype:

```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype
- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than one value

Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (&)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Reference Variable Notes

- Each reference parameter must contain `&`
- Space between type and `&` is unimportant
- Must use `&` in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

Overloading Functions

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);           // 1
void getDimensions(int, int);      // 2
void getDimensions(int, double);   // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);           // 1
getDimensions(length, width);    // 2
getDimensions(length, height);   // 3
getDimensions(height, base);     // 4
```

The `exit()` Function

- Terminates the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Requires `cstdlib` header file
- Example:

```
exit(0);
```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

Stubs and Drivers

- Useful for testing and debugging program and function logic and design
- Stub: A dummy function used in place of an actual function
  - Usually displays a message indicating it was called. May also display parameters
- Driver: A function that tests another function by calling it
  - Various arguments are passed and return values are tested