## UCLA Stat 130D
Statistical Computing and Visualization in C++/Java

**Instructor:** Ivo Dinov, Asst. Prof. in Statistics / Neurology

University of California, Los Angeles, Winter 2007
http://www.stat.ucla.edu/~dinov/courses_students.html

---

### Introduction to Computers and C++/Java Programming

- Computer Systems, Writing, compiling, making, packaging, distributing and running programs/software
  - Variables and assignments
  - Input/Output, Data types and expressions
  - Procedural (structured) vs. OOP
  - Classes, methods, abstract data types
  - Overloading (functions & classes)
  - Call-by-value vs. call-by-reference
  - I/O Streams
  - Multidimensional Arrays
  - Strings
  - Pointers, dynamic arrays
  - Recursion
  - GUI

---

### Computer Systems

- **Hardware** -- Parts of a computer you can touch
  - PC
  - Workstation
  - Mainframe
  - Grid (CCB/Wiki)
    - Infrastructure/Grid URL
  - Network
  - Input/output
  - Memory
    - Primary, secondary
    - Fixed, removable
  - CPU
  - Why 8 Bits = 1 Byte?

### Computer Systems

- **Software** -- Parts of a computer you cannot touch
  - Operating Systems
    - Macintosh
    - Windows
    - Linux
    - UNIX
  - High-Level Languages
    - Ada, C/C++, Java, BASIC, Lisp, Fortran, Python, Scheme
  - Compilers
    - Source program, object program, Linking
  - Editor: Integrated Development Environments (IDE)
    - IDEs combine editor, compiler and the execution environment (usually including a debugger)

---

### Programming and Problem Solving

- **Algorithms**
  - Idea is more general than ‘program’
  - Hard part of solving a problem is finding the algorithm
- **Program Design Process**
  - Problem Solving phase
  - Implementation phase
  - Problem definition

### Software Life Cycle

- Analysis and specification of task (problem definition).
- Design of the software (algorithm design).
- Implementation (coding).
- Testing.
- Maintenance and evolution of the system.
- Obsolescence.
Introduction to C++

- Origins of the C++ Language
  - Bjarne Stroustrup designed C++ for modeling (1985?).
  - C++ is an OOP extension of the C language.
  - C was developed as a systems programming language from the B language in the Unix environment. It grew into a general purpose programming language as its libraries were developed.

A Sample C++ Program

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

int main() {
    int number_of_pods, peas_per_pod, total_peas;
    cout << "Enter the number of pods:\n";
    cin >> number_of_pods;
    cout << "Enter the number of peas in a pod:\n";
    cin >> peas_per_pod;
    total_peas = number_of_pods * peas_per_pod;
    cout << "If you have \n";
    cout << number_of_pods;
    cout << " pea pods\n";
    cout << peas_per_pod;
    cout << " peas in each pod, then\n";
    cout << total_peas;
    cout << " peas in all the pods.\n";
    return 0;
}
```

Some details of a C++ Program

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

int main() {
    int number_of_pods, peas_per_pod, total_peas;
    cout << "Enter the number of peas in a pod.\n";
    cin >> peas_per_pod;
    cout << "Enter the number of pods: \n";
    cin >> number_of_pods;
    total_peas = number_of_pods * peas_per_pod;
    cout << "If you have: \n";
    cout << number_of_pods;
    cout << " pods\n";
    cout << peas_per_pod;
    cout << " peas in each pod, then\n";
    cout << total_peas;
    cout << " peas in all the pods.\n";
    return 0;
}
```

Some Details of a C++ Program

- `cout [see-out]` is the output stream. It is attached to the monitor screen. `<<` is the insertion operator.
- `cin [see-in]` is the input stream and is attached to the keyboard. `>>` is the extraction operator.
- "Press return after entering a number.\n" is called a `cstring` literal. It is a message for the user.
- `cout << "Press return ...\n"` sends the message to `cout`
- `cin >> number_of_pods;`
Some Details of a C++ Program

```cpp
total_peas = number_of_pods * peas_per_pod;
```
- The asterisk, `*`, is used for multiplication.
- This line multiplies the already entered values of `number_of_pods` and `peas_per_pod` to give a value which is stored (assigned to) `total_peas`.

Layout of a Simple C++ Program

```cpp
#include <iostream> using namespace std; int main() {
    variable_declarations
    Statement1;
    Statement2;
    ... Statement_last;
    return 0;
} (Other functions may follow)
```
- An include directive
- More later, for now “do it”
- Declares main function
- Start of main’s block
- Says “end program here”
- End of main’s block

Compiling and Running a C++ Program

- You write a C++ program using a text editor exactly as you would write a letter, create a home page or compose an e-mail.
- Compiling depends on the environment.
- You may be using an Integrated Development Environment or IDE. Each IDE has its own way to do things.
- Read you manuals and consult a local expert.
- You may be using a command line compiler. In that event, you may some thing like write:
  ```bash
c c myProgram.cpp /* cc=gcc, bcc32, c++ ... */
```
- Your compiler may require `.c`, `.cc`, `.cpp` or perhaps `.C`
- Linking is usually automatic. Again, read your manuals and ask a local expert.

Testing and Debugging

- An error in a program, whether due to design errors or coding errors, are known as bugs.
- Program Errors are classified as
  - design errors -- if you solved the wrong problem, you have a design error.
  - syntax errors -- violation of language’s grammar rules, usually caught by the compiler, and reported by compiler error messages.
  - run-time errors -- a program that compiles may die while running with a run-time error message that may or may not be useful.
  - logic error -- a program that compiles and runs to normal completion may not do what you want. May be a design error.

Kinds of Errors

- Design errors: occur when specifications are do not match the problem being solved.
- The compiler detects errors in syntax
- Run-time errors: result from incorrect assumptions, incomplete understanding of the programming language, or unanticipated user errors.

Summary

- Hardware: physical computing machines.
- Software: programs that run on hardware.
- Five computer components: input and output devices, CPU, main memory, secondary memory.
- There are two kinds of memory: main and secondary. Main memory is used when program is running. Secondary memory is usually nonvolatile.
- Main memory is divided into bytes, usually individually addressable.
- Byte: 8 bits. A bit is 0 or 1.
- KiloByte: 1KB = 2^10 ~ 1,000 Bytes.
- MegaByte: 1MB = 2^20KB = 1,000,000B.
- GigaByte: 1GB = 2^30MB; (Tera) 1TB = 2^30GB; (Peta) 1PB = 2^10TB
- Note: In reality, 1 byte = 9 bits, 9-th bit is for parity check
Summary (continued)

- A compiler translates high level language to machine language.
- Algorithm is a sequence of precise instructions that lead to a solution to a problem.
- A solution to a problem begins with algorithm design.
- Errors are of several kinds: syntax, run-time, and logic errors.
- A variable is used to name a number.
- cout << is an output statement
- cin >> is an input statement

Pitfall: Uninitialized Variables

- A variable that has not been set by your program will have the value left there by the last program to use the memory associated with that variable. This is an UNINITIALIZED variable. It contains garbage in the root sense of the word.
- This is illegal and incorrect but few compilers will catch this error.

```
int x = 3;
double pi = 3.14159;
int x(3);
double pi(3.14159);
int x;
```

Include Directories and Namespaces

- `#include <iostream>
  using namespace std;
- These lines provide declarations necessary to make iostream library available.
- C++ divides collections of names into namespaces. To access names in a namespace, the second line above, the using directive, is sufficient. It means that your program can use any name in the namespace std.
- A C++ namespace usually contains more than just names. They usually contain code as well.
- Older compilers will require the older style, <iostream.h>, and such compilers may not like the using directive. If your compiler doesn’t like the using directive, just omit it.

Escape sequences

- The \ (backslash) preceding a character tells the compiler that the next character does not have the same meaning as the character by itself.
- An escape sequence is typed as two characters with no space between them.
- \ is a real backslash character, not the escape character, a backslash that does not have the property of changing the meaning of the next character.
- \n newline
- \t tab character (same as control-h)
- \a alert, or bell
- \" double quote (that does not end a string literal).
- \r return

Formatting for Numbers with a Decimal Point

- The following statements will cause your floating point output to be displayed with 2 places of decimals and will always show the decimal point even when the output has a zero fractional part.
  ```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
//Output format:
78.50
```

Input using cin

- When a program reaches a cin >> statement, it waits for input. You are responsible for prompting the user of your program for proper input.
- Syntax:
  ```
cin >> number >> size;
cin >> time_to_go
  >> points_needed;
```
Designing input and output

- Echoing your input is frequently requested by problem statements. Even when not requested, it is usually better to echo your input.
- Example:
  ```
  cout << "Enter ... \n";
  cin >> user_entry_variable;
  cout << "You Entered: " << user_entry_variable;
  ```
- Scientific vs. floating point notation:
  - 3.14159
  - 0.00314159 x 10³
  - 0.00314159e3

Two Additional Variable Types
char and bool

- char is a special type that is designed to hold single members of the ASCII character set.
  - Some vendors have extended ASCII character encoding to include more characters than upper and lower case letters, digits and punctuation. (Notably: IBM, on the PC, which has been adopted in nearly all Microsoft software.)
  - cstring (from C) and string (from the Standard Library) are for more than one char value.
  - bool is a type that was added to C++ in the 1995 Draft ANSI Standard. There are only two values: true and false. Most all compilers support the bool type.
- Why do we need additional (bool/char) variable types?

Arithmetic Operators and Expressions
Precedence

When two operators appear in an arithmetic expression, there are PREFERENCE rules that tell us what happens. Evaluate the expression, X + Y * Z by multiplying Y and Z first then the result is added to X.

Rule: Do inside most parentheses first, then multiplication and division next, additions and subtractions next, and break all ties left to right.

Other Number types

<table>
<thead>
<tr>
<th>Type</th>
<th>Memory used</th>
<th>Range</th>
<th>Precision - meaningful digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) char</td>
<td>1 byte</td>
<td>[-32,767, 32,767]</td>
<td>NA</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,767 to 32,767</td>
<td>NA</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2,147,483,647 to 2,147,483,647</td>
<td>NA</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>same as int</td>
<td>NA</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>approximately 10⁻¹⁰ to 10¹⁰</td>
<td>7 digits</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>approximately 10⁻¹⁵ to 10¹⁵</td>
<td>15 digits</td>
</tr>
<tr>
<td>long double</td>
<td>10 bytes</td>
<td>approximately 10⁻³⁸ to 10³⁸</td>
<td>15 digits</td>
</tr>
</tbody>
</table>

Arithmetic Operators and Expressions
division /, and modulus %, for integer values

- Division /, and modulus % are complementary operations. Mod, or modulus, %, works ONLY for integer types.
- 4 / 3 is the quotient
- 12 % 3 = 12 mod 3 is the remainder

Simple Flow of Control
A simple branching mechanism

- Making decision in computer programs requires changing the execution from next instruction next to some other instruction next. This is called flow of control.
- There are two types of flow of control: selection and looping.
- Looping repeats an action, and will be discussed later.
- Selection chooses between alternative actions.
- Selection:
  - if (expression) Control Expression returns a bool value
  - Affirmative clause. Executed if Expression is true
  - Negative clause. Executed if Expression is false
Comparison Operators

C++ provides comparison operators for making decisions in computer programs. These operators return a value of type bool: true or false.

<table>
<thead>
<tr>
<th>Math</th>
<th>C++ Symbol</th>
<th>C++ Notation</th>
<th>Example</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
<td>==</td>
<td>x + 7 == 2 * y</td>
<td>x + 7 = 2y?</td>
</tr>
<tr>
<td>≠</td>
<td>not equal to</td>
<td>!=</td>
<td>ans != 'n'</td>
<td>ans ≠ 'n'?</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>&lt;</td>
<td>count &lt; m + 3</td>
<td>cout &lt; m + 3?</td>
</tr>
<tr>
<td>≤</td>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>time &lt;= limit</td>
<td>time s'limit?</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>&gt;</td>
<td>time &gt; limit</td>
<td>time &gt; limit?</td>
</tr>
<tr>
<td>≥</td>
<td>greater than or equal to</td>
<td>&gt;=</td>
<td>age &gt;= 21</td>
<td>age ≥ 21?</td>
</tr>
</tbody>
</table>

Logical (Boolean) Operators

- The 'and' operator & &
  Syntax: (Comparison_1) & & (Comparison_2)
  Example, in an assignment to a bool variable:
  ```cpp
tool in_range;
in_range = (0 < score) & & (10 < score);
```

- The 'or' operator ||
  Example – in an if-else statement
  ```cpp
  if ( (x ==1) || (x == y) )
    cout << "x is 1 or x equals y. in n";
  else cout << "x is neither 1 nor equal to y in n";
  ``

Not operator (!, ^, ~): E.g., ~x is true x is false

PITFALL: strings of inequalities

Suppose x, y and z are integer values.

if (x < y < z) // Unfortunately, this is WRONG BUT IT COMPILES.
cout << "z is between x and y"
Here is why the expression is WRONG.
- In mathematics x < y < z is short hand for x < y & & y < z.
- In C++, this is not true. It is still C++, but isn’t what you expect from the mathematics. In C++ the precedence rules require x < y < z be evaluated like this:
  (x < y) < z
- The parenthesized expression returns a bool value. The < requires the same type on both sides. The bool value gets converted to the int value 0 (for false) or 1 (for true). Then 0 < z or 1 < z compiles. And gives (most of the time) a wrong answer!

PITFALL: using = instead of ==

If (x = 12)                             // The = should have been ==
cout << "x is equal to 12";
else
  cout << "x is not equal to 12";

If the second expression is NEVER executed, regardless of the value of x before this statement is encountered.
WORSE, after this if statement executes, the expression x = 12 HAS ASSIGNED the value 12 to x.
Why? The expression x = 12 returns the value 12, which is assigned to the bool value true, which is used by the if.
Always write:
```cpp
if (12 == x) // Compiler will say: 12 = x "non-lvalue on left"
cout << "x is equal to 12"
else
  cout << "x is not equal to 12";
```

Simple Loop Mechanisms

Most programs include a mechanism to repeat a block of code multiple times. 30 Students, 30 grades on each test, 100 workers, pay check generator block runs 100 times.

C++ provides loops named
```cpp
while
for
do while
```
The piece of code the loop executes is called the body. Each loop execution of the body is called an iteration.

Display 2.10 A while loop

```cpp
while (count_down > 0)
{
  cout << "Hello ";
  count_down = count_down - 1;
}
```

while (bool expression) Do not put a semicolon here
```cpp
{ several statements }
```

Do this usually causes an infinite loop.
```cpp
do {
  cout << "Hello ";
  count << "Do you want another greeting?\n"
  << "Press y for yes, n for no, n\n";
  cin >> ans;
  while (ans == 'y' || ans == 'Y')
  { Don’t forget the semicolon
```
Increment and decrement operators

C++ provides the ++ and -- operators, each in each of two forms, prefix and postfix.

The text, for good teaching reasons, leaves the use of expressions using ++ and -- to provide a value until later.

For now, we use \texttt{n++;} as a synonym for \texttt{n = n + 1;} and \texttt{n--} for a synonym \texttt{n = n - 1;}

Programming Style

Comments

- The most difficult part of any programming language to learn to use properly comments.
- A comment should always tie the code to the problem being solved. In some circumstances, a comment could explain 'tricky' code. (It is better to write clear code and omit the comment.)
- /* comment in this style */ may span more than one line.
- // these comments run from the // to the end of the line.

Indenting

- **Indenting:** elements considered a group should be indented to look like a group (E.g., C++ Builder vs. Notepad).
- if-else, while, and do-while should be indented as in the sample code.
- The affirmative clause and negative clause of if-else statements should be indented more than surrounding code.
- The body of loops should be indented more than surrounding code.
- CONSISTENCY of style is more important than any particular style standard.

Predefined Functions

- Libraries

```cpp
#include <cmath> // include declarations of math library functions
#include <iostream> // include definitions of iostream objects
using namespace std; // make names available

int main() {
    double budget, area, length_side;
    cout << "Enter the amount budgeted for your dog house $"; cin >> budget;
    area = budget / COST_PER_SQ_FT; // the returned value to cout
    length_side = sqrt(area); // the function call
    cout << sqrt(3.0) << endl; // call math library function
    cout << endl; // output
}
```

Program header comments

Comments should be placed at the start of the program that describes the essential information about the program:

```cpp
/**The file name (part of what package)
* The author
* The address or other means to contact the author
* The purpose of the program
* What/How the program does
* Data Input/Output
* Execution syntax
* The date written or version number */
```

A Function call

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

int main( ) { 
    double COST_PER_SQ_FT = 10.50; 
    double budget, area, length_side; 
    area = budget/COST_PER_SQ_FT; 
    length_side = sqrt(area); // the function call 
}
```
Programmer Defined Functions

A Function Definition

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

double total_cost(int number_par, double price_par);
//Computes total cost, including 5% sales tax, on number_par items at a cost of price_par each.

int main() {
    double price, bill;
    int number;
    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;
    bill = total_cost(number, price);
The function call
    cout.setf(ios::fixed);       cout.setf(ios::showpoint);     cout.precision(2);
    cout << number << " items at " << "$" << price   <<   " each."
    << "Final bill, including tax, is $" << bill  << endl;
    return 0;
}
```

Predefined Functions

Type changing functions

Question: 9/2 is an int, but we want the 4.5 floating point result. We want the integer part of some floating point number. How do we manage?
Answer: Type casting, or “type changing functions”.
C++ provides a function named `double` that takes a value of some other type and converts it to double.
Example:
```cpp
int total, number;
double ratio;
// input total, number
winnings = double(total) / number;
```

Programmer Defined Functions

- A function Definition (Slide 2 of 2)

```
double total_cost(int number_par, double price_par) {
    // Computes total cost, including 5% sales tax, on number_par items at a cost of price_par each.
    const double TAX_RATE = 0.05; // 5% sales tax
    double subtotal;
    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

PITFALL: Problems with Library functions

- Some compilers do not comply with the ISO Standard.
- If your compiler does not work with `#include <iostream>`
  ```cpp
  use #include <iostream.h>
  ```
- Similarly, for headers like `cstdlib`: use `stdlib.h`, and for `cmath`, use `math.h`
- Most compilers at least coexist with the headers without the .h, but some are hostile to these headers.
Consider the function call:
\[
\text{bill} = \text{total\_cost(number, price)};
\]

- The values of the arguments number and price are “plugged” in for the formal parameters.
- A function of the kind discussed in this chapter does not send any output to the screen, but does send a kind of “output” back to the program. The function returns a return-statement instead of cout-statement for “output.”

Alternate form for Function Prototypes

- The parameter names are not required:
  \[
  \text{double total\_cost(int number, double price)};
  \]

- It is permissible to write:
  \[
  \text{double total\_cost(int, double)};
  \]

**PITFALL**

- Arguments in the wrong order

- When a function is called, C++ substitutes the first argument given in the call for the first parameter in the definition, the second argument for the second parameter, and so on.
- There is no check for reasonableness. The only things checked are:
  1. that there is agreement of argument type with parameter type and
  2. that the number of arguments agrees with the number of parameters.
- If you do not put correct arguments in call in the correct order, C++ will happily assign the “wrong” arguments to the “right” parameters.
- Function-overloading

**Procedural Abstraction**

Principle of Information Hiding

- David Parnas, in 1972 stated the principle of information hiding.
  - A function’s author (service provider programmer) should know everything about how the function does its job, but nothing but specifications about how the function will be used.
  - The client programmer -- the programmer who will call the function in her code -- should know only the function specifications, but nothing about how the function is implemented.

**Local Variables**

Namespaces

- All our use of namespaces has amounted to
  ```
  #include <iostream>
  using namespace std;
  ```
  - While this is correct, we are sidestepping the reason namespaces were introduced into C++, though we have done this for good teaching reasons.
  - In short, we have been “polluting the global namespace.” So long as our programs are small, this is not a problem.
  - This won’t always be the case, so you should learn to put the using directive in the proper place.
Local Variables
Using Namespaces

// Computes the area of a circle and the volume of a sphere.
// Uses the same radius for both calculations.
#include <cmath>
const double PI = 3.14159;

// Computes the area of a circle and the volume of a sphere.
// Uses the same radius for both calculations.
#include <iostream>
#include <cmath>

Local Variables
Using Namespaces

double area(double radius);
// Returns the area of a circle with the specified radius.
double volume(double radius);
// Returns the volume of a sphere with the specified radius.
int main( )
{
    using namespace std;
    double radius_of_both, area_of_circle, volume_of_sphere;
    cout << "Enter a radius to use for both a circle
" << "and a sphere (in inches): ";
    cin >> radius_of_both;
    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);
    cout << "Radius = " << radius_of_both << " inches
" << "Area of circle = " << area_of_circle
" << "square inches
" << "Volume of sphere = " << volume_of_sphere
" << "cubic inches
";
    return 0;
}

double area(double radius)
{
    using namespace std;
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    using namespace std;
    return ((4.0/3.0) * PI * pow(radius, 3));
}

Overloading Function Names

C++ distinguishes two functions by examining the function name and the argument list for number and type of arguments.

The function that is chosen is the function with the same number of parameters as the number of arguments and that matches the types of the parameter list sufficiently well.

This means you do not have to generate names for functions that have very much the same task, but have different types.

Overloading Function Names
Overloading a Function Name

// Illustrates overloading the function name ave.
# include <iostream>
double ave(double n1, double n2);
// Returns the average of the two numbers n1 and n2.
double ave(double n1, double n2, double n3);
// Returns the average of the three numbers n1, n2, and n3.
int main( )
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is 
" << ave(2.0, 2.5, 3.0) << endl;
    cout << "The average of 4.5 and 5.5 is 
" << ave(4.5, 5.5) << endl;
    return 0;
}

Overloading Function Names
Automatic Type Conversion

We pointed out that when overloading function names, the C++ compiler compares the number and sequence of types of the arguments to the number and sequence of types for candidate functions.

In choosing which of several candidates to use when overloading function names, the compiler will choose an exact match if one is available.

An integral type will be promoted to a larger integral type if necessary to find a match. An integral type will be promoted to a floating point type if necessary to get a match.
Sorter Program Design

Data Input
Sorting
Result Report

Header Files
- variable declarations

Collect user input

Perform the number sorting

Report the reordered numbers to the user (perhaps on stdout)

Histogram Computation

HistogramEqualization.html

Raw data
Student's grades
HW1

Counts

Score

What is the grade Distribution?
Score Ranges

Normal Curve – Normal (Gaussian) Distribution

Analytic Definition

Most General Form of $N(\mu, \sigma^2)$

Original Process (Data)

Most of the Data (68%) Is between $[\mu - \mu; \mu + \mu]$

// Example
// Simple Histogram Computation
#include <iostream>

void getUserInput();
void computeHistogram();
int reportHistogram();

int main()
{
    getUserInput();
    computeHistogram();
    cout << "Output of the Histogram Report is " << reportHistogram() << " 0 indicates ok!";
    return (0);
}

void getUserInput()
{
    // get user input
}

void computeHistogram()
{
    // integers in [0;11], histogram bins [0;3] , [4;7] , [8, 11]
    int bin1, bin2, bin3;
}

int reportHistogram()
{
    // report number of integers inside each bin
}
The call-by-value mechanism we have used until now is not adequate to certain tasks. Input subtasks should be carried out with a function call. This is not adequate for more than one return value. We need another mechanism.

With a Call-by-Value parameter, the corresponding argument is only read for its value. The argument can be variable, but this is not necessary. The parameter is initialized with the value of the value-parameter.

With Call-by-Reference, the corresponding argument must be variable, and the behavior of the function is as if the variable were substituted for the parameter.

To make a parameter a call-by-reference, parameter, an ampersand ( & ) is placed between the type name and the variable name in the function header and in any prototypes that are to declare this function in other places. A call-by-reference parameter

Example: void Get_Input( double & f_variable)
{
    using namespace std;
    cout << "Enter a Fahrenheit, I will return Celsius\n";
    cin >> f_variable;
}

Call by Reference parameters (1 of 2)
#include <iostream>
void get_numbers(int & input1, int & input2);
//Reads two integers from the keyboard.
void swap_values(int & variable1, int & variable2);
//Interchanges the values of variable1 and variable2.
void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.
int main()
{
    int first_num, second_num;
    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

Call by Reference parameters (2 of 2)
//Uses iostream:
void get_numbers(int & input1, int & input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}
void swap_values(int & variable1, int & variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
//Uses iostream:
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: ";
    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
}

The whole truth is - it is the address of the argument is used in place of the parameter, and the address is used to fetch values from the argument as well as to write to the argument.
Parameters and Arguments

If you keep these points in mind, you can handle all the parameter passing language.

1. The formal parameters for a function are listed in the function prototype and function definition header. A formal parameter is a place holder and a local variable that is filled at the time the function is called.

2. Arguments appear in a comma separated list in the call to any function, and are used to fill in the corresponding formal parameters. When the function is called, the arguments are plugged in for the formal parameters.

3. The terms call-by-value and call-by-reference refer to the mechanism that is used in the “plugging in” process.

Parameters and Arguments

In the call-by-value method, the arguments are read, and the parameters are initialized using a copy of the value of the argument.

In the call-by-reference method, the argument must be a variable. The behavior is exactly as if the argument were substituted for the parameter. Then if the code assigns the parameter, it is the argument that is changed. The mechanism is to pass the address of the argument and then the parameter mechanism knows where the argument is so when the parameter is written, the argument is where the writing is done.

Parameters and Arguments

Mixed Parameter Lists

It is entirely feasible to have value parameters (call-by-value parameter) mixed in with reference parameters (call-by-reference parameters).

Function prototype definition:

```c
void example(int& par1, int par2, double & par3, int &n);
```

Function Call:

```c
example(arg1, 17, arg3, local_n);
```

Here 17 is permissible because `par2` is a value parameter. This code may (but by no means must) change the values of `arg1` and `arg3`.

PITFALL: Inadvertent Local Variables

Omitting an ampersand (&) when you intend a reference parameter is a mistake that bites twice.

1. It makes you code run incorrectly, the compiler probably won’t catch it!

2. The Bug is very difficult to find because it looks right.

See the following

```c
// Inadvertent local variables (2 of 2)
void swap_values(int variable1, int variable2)
{
    int temp;
    temp = variable1;                  // Forget the & here, which
    variable1 = variable2;            // Makes these inadvertent local variables
    variable2 = temp;                 // Variable swap stays local to swap_values
}
```

//Uses iostream:
```c
void get_numbers(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1 >> input2;
}
```

//Uses iostream:
```c
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: ";
    output1 << " " << output2 << endl;
}
```

What happens if there's a discrepancy between function prototype and header definitions?
Using Procedural Abstraction
Functions calling functions

- A function may call another function, or itself, and the second function could call back the first.
- The situation is exactly the same as if the first call had been in the main function.
- Swaps the values of two variables if the values are out of order.

Preconditions and Postconditions

- The prototype comment should be broken into a precondition and a postcondition.
- The precondition is what is required to be true when the function is called.
- The postcondition describes the effect of calling the function, including any returned value and any effect on reference parameters.

Testing and Debugging Functions

- Every function should be designed, coded and tested as a separate unit from the rest of the program.
- Every function should be tested in a program in which every other function in that program has already been completely tested and debugged.
- This is catch 22. You need a framework to develop and test, but the framework must be debugged as well. How to get around?
Every function should be designed, coded and tested as a separate unit from the rest of the program. This is the essence of the top-down design strategy. How do you test a function? By writing a simple, short program called a driver that calls the function. The driver should be simple enough that we can confirm its correctness by inspection.

How do you test a program that needs a function, before you have written the function? By writing a simple, short program called a stub that provides the program with the same prototype, and provides enough data to the caller so the caller can be tested. The stub should be simple enough that we can confirm its correctness by inspection.

How do you test a program using stubs when the program needs several functions? We write stubs for all the functions, then write the real functions, putting them into the program one at a time. This way the complete program and already written code continues to be tested, while the new functions are written and tested until the final program is produced. Imagine how impossible it may be to debug a program which has errors in two, or more functions?!

Every function should be tested in a program where every other function in that program is functional, has already been completely tested and debugged.
Fundamental Rule for Testing Functions

- Every function should be tested in a program where every other function in that program is functional, has already been completely tested and debugged.

MAIN

Needs Driver & 3 stubs

// Program that uses a Stub (part 1 of 4)
void introduction();
// Postcondition: Description of program is written on the screen.

void get_input(double &cost, int &turnover);
// Precondition: User is ready to enter values correctly.
// Postcondition: The value of cost has been set to the
// wholesale cost of one item. The value of turnover has been
// set to the expected number of days until the item is sold.

double price(double cost, int turnover);
// Precondition: cost is the wholesale cost of one item.
// turnover is the expected number of days until sale of the item.
// Returns the retail price of the item.

// Program that uses a Stub (part 2 of 4)
void report_output(double cost, int turnover, double price);
// Precondition: cost is the wholesale cost of one item; turnover is the
// expected time until sale of the item; price is the retail price of the item.
// Postcondition: The values of cost, turnover, and price have been
// written to the screen.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;
    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    report_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

// Program that uses a Stub (part 3 of 4)
// Uses iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for an item at a Quick-Shop supermarket store.\n";
}

// Uses iostream:
void get_input(double &cost, int &turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item $\"; cin >> cost;
    cout << "Enter the expected number of days until sold: \"; cin >> turnover;
}

// This is only a stub:
// Not correct, but good enough for some testing.

double price(double cost, int turnover)
{
    return 9.99;  // Not correct, but good enough for some testing.
}

Streams and Basic File I/O

- A stream is a flow of characters (or other kind of data).
- Data flowing INTO your program is an input stream.
- Data flowing OUT OF your program is an output stream.
- We have dealt with two of the three standard streams already: cin and cout.
- If in_stream and out_stream are input and output streams, we could write:

int the_number;
in_stream >> the_number;
out_stream << "the_number is " << the_number << endl;
The two code lines need to be embedded in the block of a function. The `#include` goes in the normal position at the top of the file.

Example:
```
#include <fstream>
```  
```
ifstream in_stream;
ofstream out_stream;
```

- These variables are not yet attached to a file, so are not usable.
- The `fstream` library provides a member function named `open` that ties the stream to a file the outside world knows about. (More later on members of an object.)

Example:
```
in_stream.open("infile.dat");  // infile.dat must exist on your system
out_stream.open("outfile.dat"); // outfile.dat will be created.
```

**WORDS OF WARNING:**

- In the example:
  ```
in_stream.open("infile.dat");  // infile.dat must exist on your system
out_stream.open("outfile.dat"); // outfile.dat will be created.
```

- For Windows, this is at worst "8+3", i.e. 8 characters for the name and 3 characters for the extension. Many recent systems allow long file names. Read your manuals and ask a local expert.

- The file name arguments for open is known as the external name for the file. This name must be legitimate file names on the system you are using. The stream name is the name of the file to your program.

- If you have a file named `outfile.dat` on your system, and open a file named the same, `outfile.dat`, in a program, the program will delete the old `outfile.dat` and replace it with data from your program.

Once we have declared the file variables, `in_stream` and `out_stream`, and connected them to files on our system, we can then take input from `in_stream` and send output to `out_stream` in exactly the same manner as we have for `cin` and `cout`.

Examples:
```
#include <fstream>
```  
```
// appropriate declarations and open statements
int one_number, another_number;
in_stream >> one_number >> another_number;
```  
```
out_stream << "one_number: " << one_number << "another_number: " << another_number;
```

Every file should be closed when your program is through fetching input or sending output to the file.

This is done with the `close()` function.

Example:
```
in_stream.close();
out_stream.close();
```

- Note that the close function takes no arguments.
- If your program terminates normally, the system will close the arguments.
- If the program does not terminate normally, this might not happen.
- File corruption is a real possibility.
- If you want to save output in a file and read it later, then you must close the file before opening it the second time for reading.

**A File Has Two Names**

- Every input and every output file in a program has two names.
- The `external file name` is the real name of the file, which is the name known to the OS system. It is only used in your program in the call to the open function.
- The `stream name` is the name declared in the program, and that is tied to the `external file name` with the open statement.
- After the call to open, the program always uses the `stream name` to access the file.
Classes and Objects

- Consider the code fragment:
  ```
  #include <fstream>
  ...
  ifstream in_stream;
  ofstream out_stream;
  in_stream.open ("infile.dat");
  out_stream.open ("outfile.dat");
  ...
  in_stream.close();
  out_stream. close();
  ```

  Here the streams in_stream and out_stream are objects.

  An object is a variable that has functions as well as data associated with it.

  The functions open and close are associated with in_stream and out_stream, as well as the stream data and file data.

Definition of a hypothetical Integer Object

- Variables/Fields: min-value; max-value
- Functions (operating on Integer objects)
  - `int1.compareTo(int2) == 0`?
  - `int1.doubleValue()`
  - `Integer.parseInt("123") == 123`, as an integer
  - `int1.writeString()`

Introduction to Classes and Objects (continued)

- Consider the code fragment:
  ```
  #include <fstream>
  ...
  ifstream in_stream;
  ofstream out_stream;
  in_stream.open ("infile.dat");
  out_stream.open ("outfile.dat");
  ...
  in_stream.close();
  out_stream. close();
  ```

  The functions and data associated with an object are referred to as members of the object.

  Functions associated with the object are called member functions.

  Data associated with the object are called data members.

  Note that data members are not (usually) accessible to functions that aren’t members. (More on this later.)
Summary of Classes and Objects

- An object is a variable that has functions/data associated with it.
- Functions associated with an object are called member functions.
- A class is a type whose variables are objects, e.g., `std::string`.
- The object’s class determines which member functions the object has.

Syntax for calling a member function of an object:

```
Calling_Object.member_function(Argument_List);
```

Examples:

1. `std::ifstream` operator
   ```
in_stream.open("infile.dat");
```
2. `std::ofstream` operator
   ```
out_stream.open("outfile.dat");
```
3. Setting an output precision
   ```
out_stream.precision(2);
```
4. The meaning of the Member Function Name is determined by class (type of) the Calling Object.

Programming Tip

Checking that a file was opened successfully

- A very common error is attempting to open a file for reading where the file does not exist or is NOT readable. The member function open fails then.
- Your program must test for this failure, and in the event of failure, manage the error.
- Use the `std::ifstream` member function `fail` to test for open failure.
  ```
in_stream.fail();
```
  This function returns a `bool` value that is `true` if the stream is in a fail state, and `false` otherwise.

Example:

```
#include <cstdlib> // for the predefined exit(1); library function
                   // (6 of 2)
#include <cstdlib> // exit is defined in the header file cstdlib.h
using namespace std; // to gain access to the names
exit(integer_value); // to exit the program
```

1. When the exit statement is executed, the program ends immediately.
2. By convention, 1 is used as an argument to exit to signal an error.
3. By convention, 0 is used to signal a normal successful completion of the program. (Use of other values is implementation defined.)
4. The exit is defined in the `cstdlib` library header file, so any use requires:
   - `#include <cstdlib>`
   - A directive to gain access to the names

```
#include <cstdlib> // exit is defined in the header file cstdlib.h
using namespace std; // to gain access to the names
exit(integer_value); // to exit the program
```

The exit Statement

- The `exit` statement is written
  ```
  // exit statement is written
  #include <cstdlib> // exit is defined in the header file cstdlib.h
  using namespace std; // to gain access to the names
  exit(integer_value); // to exit the program
  ```

File I/O with Checks on `open` (1 of 2)

```
int main( )
    {
using namespace std;
   ifstream in_stream;
   ofstream out_stream;
in_stream.open("infile.dat");
   if (in_stream.fail( ))
   {
      cout << "Input file opening failed. 
";
      exit(1);
   }
   out_stream.open("outfile.dat");
   if (out_stream.fail( ))
   {
      cout << "Output file opening failed. 
";
      exit(1);
   }
   first, second, third;
in_stream >> first >> second >> third;
   out_stream << "The sum of the first 3\n"
   << "numbers in infile.dat\n"
   << "is \"("; do 
   (first + second + third)
   << end; 
    in_stream.close( );
   out_stream.close( );
   return 0;
    }
```

File I/O with Checks on `open` (2 of 2)

```
int first, second, third;
in_stream >> first >> second >> third;
out_stream << "The sum of the first 3\n"
   << "numbers in infile.dat\n"
   << "is \"("; do 
   (first + second + third)
   << end;
   in_stream.close( );
   out_stream.close( );
   return 0;
    }
```

File Names as Input (1 of 4)

- So far, we have used only `std::string` literals for stream names for input and output files.
- You can specify your own file names from the keyboard or from file input. Here’s how:
- A variable that can hold a file name is a `std::string` variable.
  - A `std::string` is what the textbook calls string in this chapter. This is the string type that C++ inherits from its C parentage.
  - A `std::string` is declared as in the following example:
    ```
    char file_name[256];
    ```
- The `std::string` file name, declared here, can hold at most 255 characters, (NOT 256), indices: 0, 1, 2, ..., 255.
- Why 255 and not 256?
Some notes on behavior of the cstring variables.

- "Why 255, not 256?"
- The character position just beyond the last character entered is used to hold a special character (end-of-string) that signals that is the last character. If you enter 255 characters, there are really 256 characters in file_name, just as the definition suggests.
- You can access the cstring variable file_name for input and output exactly as you access a variable of type int or a double. (The terminating character is automatically inserted by the istream insertion mechanism.)

Example:
```c++
char file_name[256];
cout << "Enter a file name (maximum of 255 characters):";
cin >> file_name;
cout << "I will process file: " << file_name << endl;
// Try to open file file_name for input. Test for success.
ifstream in_stream;
in_stream.open(file_name); // Notice the cstring variable
if (in_stream.fail())
{
    cout << "Failed to open file " << file_name << " for input"
    << endl;
    exit(1);
}
```

Notes on related ideas:

There are several generalizations of the cstring notion.

- Arrays of any type.
- The string class provided in the C++ Standard Library.
- The vector class from the Standard Template Library.
- Linked lists are also a generalization of cstring in the sense that a linked list is a container for objects.

Inputting a File Name (1 of 2)
```c++
// Reads three numbers from the file specified by the user, sums the numbers, and writes the sum to another file specified by the user.
#include <fstream>
#include <iostream>
#include <cstdlib>
int main()
{
    using namespace std;
    char in_file_name[256], out_file_name[256];
in_stream.open(in_file_name);
if (in_stream.fail())
{
    cout << "Input file opening failed."
    << endl;
    exit(1);
}
out_stream.open(out_file_name);
if (out_stream.fail())
{
    cout << "Output file opening failed."
    << endl;
    exit(1);
}
int first, second, third;
in_stream >> first >> second >> third;
out_stream << "The sum of the first 3
    numbers is " << in_file_name << endl
    << "is " << (first + second + third) << endl;
in_stream.close();
out_stream.close();
cout << "End of Program."
return 0;
}
```

Inputting a File Name (2 of 2)
```c++
in_stream.open(in_file_name);
if (in_stream.fail())
{
    cout << "Input file opening failed."
    << endl;
    exit(1);
}
```

Formatting Output with Stream Functions

Setting Stream Flags
```c++
cout.setf(ios::showpoint);
```
- The ostream member function setf sets flags in the stream. A stream flag is a variable that controls how the stream behaves. The output stream has many flags.
- The ios::showpoint is a flag-value defined in the class ios
  
  [2.34 (US) vs. 2.34 (EU) vs 234 (Non std)].
- Sending the value ios::showpoint to the self member function causes the decimal point always to be displayed.
## Formatting Flags for `setf`

<table>
<thead>
<tr>
<th>Flag value</th>
<th>Effect</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios::fixed</code></td>
<td>floating output not in scientific notation. Unsets scientific flag</td>
<td>not set</td>
</tr>
<tr>
<td><code>ios::scientific</code></td>
<td>floating point output, will be scientific notation as needed</td>
<td>not set</td>
</tr>
<tr>
<td><code>ios::pos</code></td>
<td>plus (+) sign prefixes not set</td>
<td>not set</td>
</tr>
<tr>
<td><code>ios::point</code></td>
<td>a decimal point and trailing zeros are printed for floating point output.</td>
<td>not set</td>
</tr>
</tbody>
</table>

## I/O Manipulators

An I/O manipulator is a function that is called in a nonstandard manner. Manipulators are called by insertion into the output stream as if they were data. The stream calls the manipulator function, changing the state of the I/O stream.

### Example:

```cpp
void make_neat(ifstream& messy_file, ofstream& neat_file, int number_after_decimalpoint, int field_width);
```

### Example (2 of 3)

```cpp
void make_neat(ifstream& messy_file, ofstream& neat_file, int number_after_decimalpoint, int field_width);  // Precondition: The streams `messy_file` and `neat_file` have been connected to files using the function open.
// Postcondition: The numbers in the file connected to `messy_file` have been written to the screen and to the file connected to the stream `neat_file`. The numbers are written one per line, in fixed point notation (i.e., not in scientific notation), with `number_after_decimalpoint` digits after the decimal point; each number is preceded by a plus or minus sign and each number is in a field of `field_width` digits. (This function does not close the files.)
```

## Formatting Flags for `setf`

<table>
<thead>
<tr>
<th>Flag value</th>
<th>Effect</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios::right</code></td>
<td>if a width is set and output fits within, output is right justified within the field</td>
<td>set</td>
</tr>
<tr>
<td><code>ios::left</code></td>
<td>if a width is set and output fits within, output is left justified within the field</td>
<td>not set</td>
</tr>
</tbody>
</table>

## Streams as Arguments to Functions

A stream may be an argument for a function just like any other object type. Because the effect of any function that uses a stream it to change the stream, it is necessary to pass the stream by reference.
void make_neat(ifstream& messy_file, ofstream& neat_file, 
int number_after_decimalpoint, int field_width) 
{
    neat_file.setf(ios::fixed);
    neat_file.setf(ios::showpoint);
    neat_file.setf(ios::showpos);
    neat_file.precision(number_after_decimalpoint);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(number_after_decimalpoint);

    double next;
    while (messy_file >> next)
    {      cout << setw(field_width) << next << endl;
        neat_file << setw(field_width) << next << endl;
    }
}
A Note on compilers

- If you are using Emacs editor under Windows2K you may find that the input functions get and getline fail to work in peculiar fashion. Perhaps this is true even if we use the Borland command line compiler.
- A workaround is to run the programs in a DOS Window.

'\n' and "\n"

- The text says that "\n" is a string having exactly one character. The string "\n" has only one character stored in it, the newline.
- By contrast, recall the terminating character we had to leave space for when we were inputting file names.
- There we saw that cstrings had to have space for one character signal the end of string, beyond what we could use. The string literal "\n" also has a terminating character in addition to the newline character stored in the cstring.

The putback Member Function (1 of 2)

- Sometimes you need to inspect but not process the next character from the input. To do this you can read the character, decide you didn’t want it, and push it back on the input stream.
- The member function, putback, is a member of every input stream. It takes an argument of type char, and replaces the character read from the input.

Programming Example: Checking Input(1 of 2)

If a program does not check input, a single bad character input can ruin the entire run of a program. This program allows the user to reenter input until input is satisfactory.

Example:
```cpp
#include <iostream>

void new_line( );
// Discards all the input remaining on the current input line.
// Also discards the 'n' at the end of the line.
// This version only works for input from the keyboard.

void get_int(int& number);
// Postcondition: The variable number has been given a value that the user approves of.
```

The putback Member Function (2 of 2)

Example:
```cpp
fin.get(next);
while (next != ' ')
{   fout.put(next);
    fin.get(next);
}
fin.putback(next);
```
This code reads characters until a blank is encountered, then puts the blank back on the input.

Final notes:
The character put back need NOT be the one we read! It can be any character.
The input file will not be changed, but the program will behave as if it were (only the local, RAM, copy of the stream is effected).
Programming Example:
Checking Input (2 of 2)

```cpp
int main()
{
    using namespace std;
    int n;
    get_int(n);
    cout << "Final value read in = " << n << endl << "End of demonstration.\n";
    return 0;
}
```

```cpp
// Uses istream:
void new_line()
{
    using namespace std;
    char symbol;
    do
    {    cin.get(symbol);
    }    while (symbol != '\n');
}
```

```cpp
//        Uses iostream:
void get_int(int& number)
{
    using namespace std;
    char ans;
    do
    {    cout << "Enter input number: ";
        cin >> number;
        cout << "You entered " << number;
        cout << "Is that correct? (yes/no): ";
        cin >> ans;
        new_line();
    } while ((ans != 'Y') && (ans != 'y'));
}
```

Pitfall: Unexpected ‘\n’ in Input (1 of 3)

- Example with a problem:
  ```cpp
  cout << "enter a number: \n";
  int number;
  cin >> number;
  cout << "enter a letter: \n";
  char symbol;
  cin >> symbol;
  ```

- With the input
  enter a number: 21
  Enter a letter: A

Unfortunately, with this code, the variable `number` gets 21, but character `symbol` gets a newline, not 'A'.

Pitfall: Unexpected ‘\n’ in Input (2 of 3)

- With get, one must account for every character on the input, even the new-line characters.
- A common problem is forgetting to account for the newline at the ends of every line.
- While it is legal to mix cin >> style input with cin.get() input, the code in the previous slide can cause problems.

You can rewrite this using the newline() function from the Preceeding Programming Example to dump the characters remaining on the input.

Pitfall: Unexpected ‘\n’ in Input (3 of 3)

Two workable rewrites of the code on slide 1 of 3 above

```cpp
cout << "enter a number \n";
int number;
cin >> number;
cout << "enter a letter: \n";
char symbol;
cin >> symbol;  // as opposed to: cin.get(symbol);
```

Or You may write:

```cpp
cout << "enter a number \n";
int number;
cin >> number;
new_line();
cout << "enter a letter: \n";
char symbol;
cin.get(symbol);
```

The `eof` Member Function (1 of 2)

Every input-file stream has a function to detect end-of-file called `eof`. The letters stand for end of file.

This function returns a `bool` value, true if the input file is at end-of-file, false otherwise.

The result of this function can be used to control a while-loop, a for-loop or an if statement.

Typically, what we are really interested in is when we are not at end of file. The code usually reads,

```cpp
while(! file_stream.eof())
{
    // do something with the file_stream
}
```

The `eof` Member Function (2 of 2)

The while loop might look like this:

```cpp
char next;
in_stream.get(next);
while(! in_stream.eof())
{
    cout << next;  // This could be cout.put(next);
in_stream.get(next);
}
```

This loop reads the file attached to in_stream into the char variable next character by character.

There is an end of file marker that can be read. The `eof` function does not change from false to true until this marker is read. You can read this marker but writing it out produces unpredictable results.
Call by Reference parameters (1 of 3)
// Program to create a file called cplusad.dat which is identical to the file
// cad.dat, except that all occurrences of 'C' are replaced by "C++".
// Assumes that the uppercase letter 'C' does not occur in cad.dat, except
// as the name of the C programming language.

#include <fstream>
#include <iostream>
#include <cstdlib>
using namespace std;

void add_plus_plus(ifstream& in_stream, ofstream& out_stream);
// Precondition: in_stream is connected to an input file with open().
// out_stream has been connected to an output file with open().
// Postcondition: The contents of the file connected to in_stream is
// copied into the file connected to out_stream, but with each 'C' replaced
// by "C++". (The files are not closed by this function.)

Call by Reference parameters (2 of 3)
int main()
{
    ifstream fin;
    ofstream fout;
    cout << "Begin editing files."
    fin.open("cad.dat");
    if (fin.fail())
        { cout << "Input file opening failed."
          exit(1);
        }
    fout.open("cplusad.dat");
    if (fout.fail())
        { cout << "Output file opening failed."
          exit(1);
        }
    add_plus_plus(fin, fout);
    fin.close();
    fout.close();
    cout << "End of editing files."
    return 0;
}

void add_plus_plus(ifstream& in_stream, ofstream& out_stream)
{
    char next;
    in_stream.get(next);
    while (! in_stream.eof())
    {
        if (next == 'C') // NOTE char comparison!
            out_stream << "C++"
        else
            out_stream << next;
        in_stream.get(next);
    }
}

Some predefined character functions (1 of 2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>toupper(char_expr)</td>
<td>if char_expr is lowercase transform char_expr to uppercase return this value</td>
<td>char c = toupper('a'); cout &lt;&lt; c &lt;&lt; &quot;A&quot;;</td>
</tr>
<tr>
<td></td>
<td>else return char_expr</td>
<td></td>
</tr>
<tr>
<td>tolower(char_expr)</td>
<td>lowercase version of toupper</td>
<td>char c = 'a'; if (tolower(c)) cout &lt;&lt; c &lt;&lt; &quot;a&quot;;</td>
</tr>
<tr>
<td></td>
<td>if arg is uppercase return true</td>
<td>else return false;</td>
</tr>
<tr>
<td></td>
<td>else return false</td>
<td></td>
</tr>
<tr>
<td>isalpha(char_expr)</td>
<td>if ('a' &lt;= char_expr &amp;&amp; char_expr &lt;= 'z')</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(isalpha()) cout &lt;&lt; c &lt;&lt; &quot;is a letter.&quot;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else cout &lt;&lt; c &lt;&lt; &quot;is not a letter.&quot;;</td>
<td></td>
</tr>
<tr>
<td>isdigit(char_expr)</td>
<td>if ('0' &lt;= char_expr &amp;&amp; char_expr &lt;= '9') return true; * Otherwise return false;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(isdigit()) cout &lt;&lt; c &lt;&lt; &quot;is a digit.&quot;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else cout &lt;&lt; c &lt;&lt; &quot;is not a digit.&quot;;</td>
<td></td>
</tr>
<tr>
<td>isspace(char_expr)</td>
<td>if char_expr is any of whitespace, such as tab, newline or blank</td>
<td>return true; * Otherwise return false;</td>
</tr>
<tr>
<td></td>
<td>return true; * Otherwise return false;</td>
<td></td>
</tr>
</tbody>
</table>

Inheritance

- One of the most powerful features of C++ is the use of derived classes.
- A class is derived (subclass) from another class (superclass, base-class) means that the derived class is obtained from the superclass by adding features while retaining all the features in the superclass.
- We speak of the derived class inheriting from the base-class, and this mechanism as inheritance.
- We use the words inherit, inheriting, and speak of a derived class inheriting from a base class.
Inheritance

**NUMBER class**
- string toString();  // Every number should
- // be printable as string

**Integer class**
- int min=-2147483647;
- int max=2147483647;
- bool compareTo(int);
- int increment(int);

**Double class**
- string toString();
- int getPrecision();   // 15 digits

**Super Class**

**Sub-classes**

Inheritance Among Stream Classes (1 of 3)
- Recall that an object has member data and functions.
- Also a class is type whose variables are objects.
- It turns out that ALL file streams are derived from other I/O streams.
- In particular, ifstream inherits from istream, and
- ofstream inherits from ostream.
- Notice that ifstream has the open() member function but
- istream does not, remember NUMBER – INTEGER class
- relation. So, sub-class (ifstream) not only inherits from
- the super-class (or base-class) istream, but it has extra
- members.

Inheritance Among Stream Classes (2 of 3)
- We say one class is derived from another class if the derived class is
- obtained from the other super-class by keeping all the features of
- the super-class and adding additional members: data/variables and functions.
- Input-file streams are derived from istreams.
- The class ofstream is derived from class ostream.
- Any input stream (incl. ofstream outf_stream) is an object of type ostream,
- but NOT the other way around. Example:
- void say_hello(ofstream& outf_stream)
  {   outf_stream << "Hello!" << endl;   }
- However, the istream object cout does
- NOT have a method close() as a member!
- cout.close(); if illegal

Inheritance Among Stream Classes (3 of 3)
- If class B inherits from class A, then A is said to be
- the base class (superclass), and B is said to be the
- derived class (subclass).
- If class B inherits from class A, then any object of
- class B is also an object of class A. Wherever a
- class A object is used, we can substitute it by a
- class B object.
- If class B inherits from class A, then class A is called
- the parent class and class B is called the child
- class.
- Some textbooks prefer base class and derived class.

ofstream objects is an ostream object

Making Stream Parameters Versatile
- Suppose you define a function that takes a input stream as
- an argument and you want the argument to be cin in some
- cases and an input-file stream in others. To accomplish
- this, you can use istream as a formal parameter type. Then
- you can use either an ifstream object or an istream object
- as parameter.
- Similarly, you can define a function that uses an ostream
- formal parameter then use either ostream arguments or
- ofstream arguments.
- void say_hello(ostream& o_stream)
  {   o_stream << "Hello!" << endl;   }

Programming Example: Another new_line function

By using an istream formal parameter, we can use this
new_line function with either ifstream or istream arguments.

```cpp
// Uses istream
void new_line(istream& in_stream)
{
  char symbol;
  do
  {   in_stream.get(symbol);
      if (symbol != "in");
  }```

```
Default Arguments (1 of 2)

An alternative to writing two versions of new_line() we can write one version with a default argument:

```cpp
// Uses iostream
void new_line(istream& in_stream = cin) {
    char symbol;
    do {
        in_stream.get(symbol);
    } while (symbol != 'n');
}
```

Default Arguments (2 of 2)

If we call this as `new_line();` then the default argument, cin, is used.

If we call this as `new_line(fin);` where fin is an ifstream object, then the function uses this as the argument.

If several parameters are to have default arguments, but some parameters do not have default arguments, those parameter supplied with default arguments must follow those not supplied with default arguments. If arguments are provided in a call, there must be enough arguments to provide for parameters without defaults. Arguments beyond this number will replace default arguments.

Structures

- A class is a data type that can be made to behave the same as built-in data types (e.g., int). Such data types are called Abstract Data Types. A class encapsulates both data and functions.
- A structure may be thought of as an object **without** member functions.
- A structure definition defines a type.
- **struct CDAccount**
  ```cpp
  struct CDAccount {
      double balance;
      double interest_rate;
      int term;                  //                   months until maturity
  };
  ```

- **DON'T FORGET THE SEMICOLON**

Structs

- Given the structure definition on the previous slide, **structure variables (of this type)** can be defined by:
  ```cpp
  CDAccount my_account, your_account;
  ```

- This structure variable definition creates member variables balance, interest_rate, and term associated with the structure, for each structure variable.
- Member variables are **accessed** using the **dot operator**.
  ```cpp
  my_account.balance; // type is double
  ```
- Other structure variable's members may also be accessed:
  ```cpp
  your_account.balance;
  ```

- These variables may be used exactly like any other variables.
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);  
cout << "When your CD matures in 
<< account.term << " months,in"
<< "it will have a balance of $";
<< account.balance << endl;
return 0;
}

// Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: 
";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity
";
    cin >> the_account.term;
}

struct CDAccount
{
    double balance;  
    double interest_rate;  
    int term;  // months to run
};

int main( )
{
    CDAccount account;  
    account.balance = 1000.00;  
    account.interest_rate = 4.7;  
    account.term = 11;
    balance 1000.00  
    interest_rate 4.7  
    term 11
}

The Dot Operator

The dot operator is used to specify a member variable of a structure variable.

Syntax:  Structure_Variable_Name.Member_variable_name

Example:  
struct StudentRecord
{
    int student_number;
    char grade;
};

int main()
{
    StudentRecord your_record;
    your_record.student_number = 2001;
    your_record.grade = 'A';
}

The dot operator is also called "structure member access operator".

Structures as Function Arguments

A function can have
- call-by-value parameters of structure type and/or
- call-by-reference parameters of structure type and/or
- return type that is a structure type

Example, a wrapper that instantiates (constructs) a structure from 3 values:

CDAccount shrink_wrap( double the_balance,
                          double the_rate,  int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}

void exampleFunc(CDAccount& acc1);
void exampleFunc1(CDAccount acc2);

Programming Tip(1 of 2)
Use Hierarchical Structures

- If a structure has a subset of its members that may be considered an entity, consider nested structures.
- Example:  
  A PersonInfo struct might include a birthday

      struct Date
      {
          int month;
          int day;
          int year;
      }

      struct PersonInfo
      {
          double height; // inches
          int weight; // pounds
          Date birthday;
      };

Programming Tip(2 of 2)
Use Hierarchical Structures

- Declare a variable of Personinfo type as usual:
PersonInfo person1;

Person1.birthday
// This is a Date structure, with members accessible
// as in any other structure variable.

If the structure variable person1 has been set, the year a person was born can be obtained by:

cout << person1.birthday.year;
   // structure member contained in (inner) structure.
Initializing Structures

- A structure may be initialized at the time it is declared.

```c
struct Date {
    double hour;
    int month;
    int day;
    int year;
};
```

```c
Date due_date = { 1520.00, 12, 31, 2001};
```

- The sequence of values is used to initialize the successive variables in the struct. The order is essential.

- It is an error to have more initializers than variables.

- If there are fewer initializers than variables, the provided initializers are used to initialize the first few data members. The remainder are initialized to 0 for primitive types.

Defining Classes and Member Functions

- A class is a data type whose variables are objects.

- An object is a variable that has member functions and member variables.

- A class definition specifies the function members and the data members.

- A data members for a class are defined much as we have defined structure members.

- Programmer defined member functions of a class are called exactly as we showed for predefined classes, e.g., `ostream` and `ofstream`.

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

class DayOfYear {
public:
    void output( ); // calls to the member function Output

    int month;
    int day;
};

int main( ) {
    DayOfYear today, birthday;
    cout << "Enter today's date:
"
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:
"
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    // Uses iostream:
    void DayOfYear::output( ) { // member function definition
        cout << "month = " << month << ", day = " << day << endl;
    }

    // Class with a Member Function (1 of 2)
    // Program to demonstrate a very simple example of a class.
    // A better version of the class DayOfYear will be given next.
    if (today.month == birthday.month && today.day == birthday.day)
        cout << "Happy Birthday!"
    else
        cout << "/, sorry your B-Day is not today!"
    return 0;
}
```

Encapsulation

Combining several items such as variables, or variables and functions, into a single package, such as an object of some class, is called **encapsulation**.
The Dot Operator and the Scope Resolution Operator

Both the dot operator and scope resolution operator are used with member names to specify the thing they are a member of. For example, suppose you have declared a class called DayOfYear, and you declare an object called today of type DayOfYear:

```cpp
DayOfYear today; // today is an object of class DayOfYear
```

You use the dot operator . to specify a member of this object. For example, output( ) is a member function of class DayOfYear (see Display 6.3) and this call will output data stored in the particular object today:
```
[ today.output( ); ]
```

You use the scope resolution operator :: to specify the class name when giving the function definition for a member function. For example, the heading of the function definition for the member function output( ) is:
```
void DayOfYear::output()
```

Remember, the scope resolution operator :: is used with a class name, while the dot operator is used with an object of that class.

Class with Private Members (2 of 3)

```cpp
int main() {
    DayOfYear today, bach_birthday;
    cout << "Enter today's date:
";
    today.input();
    cout << "Today's date is ";
    today.output();
    bach_birthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bach_birthday.output();
    if (today.get_month() == bach_birthday.get_month()) &&
        today.get_day() == bach_birthday.get_day())
        cout << "Happy Birthday Johann Sebastian!\n";
    else cout << "Today is NOT Johann Sebastian's B-Day!\n";
    return 0;
}
```

Class with Private Members (3 of 3)

```cpp
// Uses iostream:
void DayOfYear::input() {
    cout << "Enter the month as a number: [1;12]\n";
    cin >> month;
    cout << "Enter the day of the month: [1;31]\n";
    cin >> day;
}
void DayOfYear::output() {
    cout << "month = " << month << " day = " << day << endl;
}
void DayOfYear::set(int new_month, int new_day) {
    month = new_month;
    day = new_day;
}
int DayOfYear::get_month() {
    return month;
}
int DayOfYear::get_day() {
    return day;
}
```

Public and Private Members (1 of 2)

With an ideal class definition, the class author should be able to change the details of the class implementation without necessitating changes in any program using the class (code using the class is called "client code").

This requires enough member functions to access the data members whenever necessary. This will allow the representation of the data to be changed as required by changes in implementation without changing client code.

Everything (functions or data members) defined after "private:" line are accessible only in member functions of the class. (See the remark in the next slide.)

The keyword public is used to state that the members defined after the "public:" line are accessible in any function that can see the class definition. (Again, see the next slide.)

Public and Private Members (2 of 2)

Remark:

It is not quite true that everything (functions or data members) defined after "private:" line are accessible only in member functions of the class.

There can be several public and private sections in a class, and there is one other access keyword we will talk about later. Members defined after "public:" up to the next "private:" or other access specifier keyword are accessible by all functions. Members defined after "private:" up to the next public: or other access keyword are accessible only by all functions defined in the class.

While we won’t have several public and private sections in our classes, you may find code that makes use of multiple public and private sections, so we mentioned this for the sake of completeness.
The Bank Account Class (1 of 4)
// Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;
// Class for a bank account:
class BankAccount
{
    // Postcondition: The account balance has been set to $dollars.cents;
    // The interest rate has been set to rate percent.
    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update( );
    double get_balance( );
    double get_rate( );
    void output(ostream& outs);
    void fraction(double percent);
private:
    double balance;
    double interest_rate;
};

int main( )
{  
    BankAccount account1, account2;
    cout << "Start of Test:"
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:
    account1.output(cout);
    account1.set(100, 5.0);
    cout << "account1 with new setup:
    account1.output(cout);
    account1.update( );
    cout << "account1 after update (1 yr interest):
    account1.output(cout);
    account2 = account1;
    cout << "account2:
    account2.output(cout);
    return 0;
} 182

Programming Tips
- Make Data Members private. When defining a class, the
  normal practice is to make all member variables private. This
  means these variables can only be accessed or changed
  using member functions.
- Define Accessor Functions (get/set). The operator == does
  not apply to class objects without additional work. Functions
  get_day and get_month are accessors. Consider providing a
  complete set of accessors to data in useful formats. This will
  make comparing objects for equality easier.
- Using the Assignment Operator with Objects: The assignment
  operator = applies to struct and class objects. In the case
  where all member variables are primitive (char, short, int,
  long, float, double, long double, bool) operator = can be used
  to assign class objects. The members are each assigned.

Structures versus Classes
- The keyword struct was provided in C++ for
  backward compatibility with C. For this reason
  many authors treat the struct as a class though it
does not have function members.
- In fact, a struct can have function and data
  members exactly like a class. The only difference
  is that struct default access is public, whereas
class default access is private.
- We encourage use of the struct without function
  members and classes as developed in this
  chapter. This use follows C++ programming
custom.
Constructors for Initialization

- For automatic initialization of class objects at definition, C++ provides a special kind of member function known as a constructor.
- A class constructor has the same name as the class.
- A constructor does not return a value, not even void. In a constructor, a return statement is allowed only without an argument.
- Class constructors may be overloaded as needed.

```cpp
class BankAccount
{
    public:
        BankAccount(int dollars, int cents, double rate);
        
    private;
        double balance;
        
};
```

```cpp
BankAccount::BankAccount(int d, int c, double r)
{
    dollars = d;
    cents = c;
    rate = r;
}
```

Calling a Constructor

A constructor is called automatically when an object is declared, but you must give the arguments for the constructor when you declare the object. A constructor can be called explicitly to create a new object.

**Syntax**

(For an object declaration when you have constructors):

`Class_Name Object_Name(Arguments_for_Constructor);`

**Example:**

`BankAccount account1(100, 2.3);`

**Syntax**

(For an explicit constructor call):

`Object_Name = Constructor_Name(Arguments_for_Constructor);`

**Example:**

`account1 = BankAccount(200, 3.5);`

A constructor must have the same name as the class of which it is a member. Hence `Class_Name` and `Constructor_Name` are the same identifier.

Programming Tip: Always Include a Default Constructor (1 of 3)

```cpp
class SampleClass
{
    // This constructor requires two arguments
    public:
        SampleClass(int parameter1, double parameter2);
        void do_stuff();
    
    private;
        int data1;
        double data2;
    
};
```

```cpp
SampleClass my_object(7, 7.77); // OK, supplies required arguments
SampleClass my_object = SampleClass(); // Illegal - no Default constructor
```

A constructor with prototype `SampleClass();` is called a default constructor (trivial list of arguments).

Programming Tip: Always Include a Default Constructor (2 of 3)

```cpp
class SampleClass
{
    constructor requires two arguments
    public:
        SampleClass(int parameter1, double parameter2);
        SampleClass(); // a default constructor
        void do_stuff();
    
    private;
        int data1;
        double data2;
    
};
```

```cpp
SampleClass my_object(7, 7.77); // OK, supplies required arguments
SampleClass myObject = SampleClass(); // Legal - default constructor
```

The constructor `SampleClass()` is called a default constructor.
Programming Tip
Always Include a Default Constructor(3 of 3)

If no constructors are provided, the compiler will “implicitly”
generate a default constructor that does nothing but be present
to be called.
If any constructor is provided at all, no default constructor will be
-generated. In that case the attempted definition
SampleClass myObject;
will try to call a default constructor so will fail, as there will be none.

Pitfall:
Constructors with no arguments

- The declaration
  BankAccount object_name(100, 2.3);
invokes the BankAccount constructor that requires
two parameters.
- The function call
  f();
invokes a function f that takes no parameters
- Conversely,
  BankAccount object_name();
does NOT invoke the no-parameter constructor.
- Rather, this line of code defines a function that
  returns an object of BankAccount type.

Constructors with No Arguments

When you declare an object and want the constructor with zero
arguments to be called, you do not include parentheses. For example,
to declare an object and pass tow arguments, you might do this:
BankAccount account(100, 2.3);
However, to cause the constructor with NO arguments, to be called, you
declare the object:
BankAccount account;
You do NOT declare the object
BankAccount account();  //THIS IS NOT WHAT YOU WANT!!
(This declares account to be a function that has no parameters and
returns a BankAccount object as its function value.)

Abstract Data Types
Classes to Produce ADTs

- A data type has a set of values and a set of operations
  For example:
  the int type has values \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}
  and operations +, -, *, /, %.
- A data type is called an Abstract Data Type (ADT) if the
  programmers who use the type do not have access to the
details of how the values and operations are
implemented.
- Programmer defined types are not automatically ADTs.
  Care is required in construction of programmer defined
types to prevent unintuitive and difficult-to-modify code.

Classes to Produce ADTs
How to make an ADT:

- Make all the member variables private.
- Make each of the basic operations that the programmer needs a
  public member function of the class, and fully specify how to use
  each such function.
- Make any helping functions private member functions.
- The interface consists of the public member functions along with
  commentary telling how to use the member functions. The interface
  of an ADT should tell all the programmer need to know to use the
  ADT.
- The implementation of the ADT tells how the ADT is realized in C++
  code. The implementation consists of private members of the class
  and the definitions of all member functions. This is information the
  programmer should NOT NEED to use the class.

Programming Example
Alternative Implementation of a Class

- The client programmer does not need to know how data is stored, nor
  how the functions are implemented.
- Consequently alternative implementations may store different
  variables differently.
- Suppose we have interest_rate variables, but the different
  implementations store this value differently (as a percent, say 4.7, vs
  as a decimal fraction, say 0.047). There are also differences in the
  implementations get_balance() methods. Client user need not be
  concerned with these differences.
- There may also be other differences between the implementations,
  e.g., different computational algorithms, totally different program
  organization which may or may not produce the same output.
// Demonstrates an alternative implementation of the class BankAccount.  
#include <iostream>  
#include <cmath>  
using namespace std;  

private:  
    int dollars_part;  
    int cents_part;  
    double interest_rate;  
    double fraction(double percent);  
    //      Converts a percent to a fraction. For example, fraction(50.3)  
    //           returns 0.503.  
    double percent(double fraction_value);  
    //      Converts a fraction to a percent. For example, percent(0.503)  
    //               returns 50.3.  
};  

// Class for a bank account:  BankAccount look and behave exactly the same as before  
public:  
    BankAccount(int dollars, int cents, double rate);  
    //    Initializes the account balance to $dollars.cents and  
    //    initializes the interest rate to rate percent.  
    BankAccount(int dollars, double rate);  
    //    Initializes the account balance to $dollars.00 and  
    //    initializes the interest rate to rate percent.  
    BankAccount();  
    //   Initializes the account balance to $0.00 and the interest rate to 0.0%.  
    void update();  
    //    Postcondition: One year of simple interest has been added to  
    //    account balance.  
    double get_balance();  
    //     Returns the current account balance.  
    double get_rate();  
    //     Returns the current account interest rate as a percent.  
    void output(ostream& outs);  
    //    Precondition: If outs is a file output stream, then  
    //     outs has already been connected to a file.  
    //    Postcondition: Account balance and interest rate have been  
    //         written to the stream outs.  
    
int main( )  
{  
    BankAccount account1(100, 2.3), account2;  
    cout << "account1 initialized as follows:
";  
    account1.output(cout);  
    account2.output(cout);  
    account1 = BankAccount(999, 99, 5.5);  
    cout << "account1 reset to the following:
";  
    account1.output(cout);  
    return 0;  
}  

double BankAccount::get_balance( )  
{  
    return (dollars_part + 0.01*cents_part);  
}  

double BankAccount::percent(double fraction_value)  
{  
    return percent(fraction_value);  
}  

double BankAccount::get_rate( )  
{  
    return percent(interest_rate);  
}  

// Alternative BankAccount Implementation(4 of 6)  
BankAccount::BankAccount(int dollars, double rate)  
{  
    dollars_part = dollars;  
    cents_part = 0;  
    interest_rate = fraction(rate);  
}  
BankAccount::BankAccount( )  
{  
    dollars_part = 0;  
    cents_part = 0;  
    interest_rate = 0.0;  
}  

double BankAccount::fraction(double percent)  
{  
    return (percent/100.0);  
}  

double BankAccount::percent(double fraction_value);  
//      Converts a fraction to a percent. For example, percent(0.503)  
//               returns 50.3.  

double BankAccount::get_balance( );  
//    Returns the current account balance.  
double BankAccount::get_rate( );  
//    Returns the current account interest rate as a percent.  
void BankAccount::output(ostream& outs);  
//    Precondition: If outs is a file output stream, then  
//     outs has already been connected to a file.  
//    Postcondition: Account balance and interest rate have been  
//         written to the stream outs.  

int main( )  
{  
    BankAccount account1(100, 2.3), account2;  
    cout << "account1 initialized as follows:
";  
    account1.output(cout);  
    cout << "account2 initialized as follows:
";  
    account2.output(cout);  
    account1 = BankAccount(999, 99, 5.5);  
    cout << "account1 reset to the following:
";  
    account1.output(cout);  
    return 0;  
}  

void BankAccount::output(ostream& outs)  
{  
    outs.setf(ios::fixed);  
    outs.setf(ios::showpoint);  
    outs.precision(2);  
    outs << "Account balance $" << get_balance( ) << endl;  
    outs << "Interest rate " << get_rate( ) << "%" << endl;  
    //The new definitions of get_balance and get_rate  
    // ensure that the output will still be in the correct units.  
}
We discussed information hiding when we introduced functions in Chapter 3. We said that information hiding, as applied to functions means that you should write the function so that they can be used with no knowledge of how they were written: as if they were black boxes. We know only the interface and specification. This principle means that all the programmer needs to know about a function is its prototype and accompanying comment that explains how to use the function. The use of private member variables and private member functions in the definition of an abstract data type is another way to implement information hiding, where we now apply the principle to data values as well as to functions.

A company wants to transmit data over the telephone, but they are concerned that their phones may be tapped. All of their data are transmitted as four-digit integers. They have asked you to write a program that encrypts their data so that it can be transmitted more securely. Your program should read a four-digit integer and encrypt it as follows: Replace each digit by \((\text{the sum of that digit plus } K) \mod 10\). Then, swap the first digit with the third, swap the second digit with the fourth and print the encrypted integer. Write a separate program that inputs an encrypted four-digit integer and decrypts it to form the original number.

A company wants to transmit data over the telephone, but they are concerned that their phones may be tapped. All of their data are transmitted as strings of characters. They have asked you to write a program that encrypts their data so that it can be transmitted more securely. Your program should read a string from a file and encrypt it as follows: Replace each character by \((\text{the sum of its digit plus } K) \mod 256\). Then, invert every other character. Finally, report the string in reverse order (back-to-front). Ask the user for the integer \(K\) and if encoding or decoding is to be performed, read the data from a file and write the output message to another file.

### Solution

- **Problem Understanding**
- **Top-Down algorithmic design**
- **Details implementation**
- **Testing**
- **Debugging and redesign**
1. Problem Understanding

Encryption:
1. char → (char plus KEY) % 256
2. Then, invert every other character:
   - char → (256-char) if char_index = odd
   - char, if char_index = even
3. Report the string in reverse order

Description:
1. Report the string in reverse order
2. Then, invert every other character:
3. char → (char plus (256-KEY)) % 256

Communication Medium

Can we talk?

Use the Key!