THE UNIVERSITY *of* TEXAS

HEALTH SCIENCE CENTER AT HOUSTON
SCHOOL *of* HEALTH INFORMATION SCIENCES

# Introduction to C++ Part I

For students of HI 5323
“Image Processing”

Willy Wriggers, Ph.D.
School of Health Information Sciences

http://biomachina.org/courses/processing/02.html

# History

- Merges notions from Smalltalk and notions from C
- The class concept was borrowed from Simular67
- Developed by Bjarne Stroustrup at Bell Labs
- Bell Lab's internal standard language for system programming
- Keeps C's original executing speed while enabling object-oriented programming (OOP).
- C++ is a strongly typed language (provides strong guarantees about run-time behavior)

# History of C and C++

- C evolved from two other programming languages, BCPL and B
- ANSI C
  - Established worldwide standards for C programming
- 1980: “C with Classes”
  - Improve program structure (Simula67)
  - Maintain run-time efficiency
  - Support rather than *enforce* effective programming techniques
- 1985: C++ 1.0
- 1995: Draft standard
- C++ “spruces up” C
  - Provides capabilities for object-oriented programming
    - Objects are reusable software components that model things in the real world
    - Object-oriented programs are easy to understand, correct and modify

# C++: ISO/IEC 14882

- 1998: International standardization
  - New type *bool*
  - *static_cast*, *dynamic_cast* etc
  - Run-Time Type Information (RTTI)
- “Standard Library” (STL)
  - Generic containers
  - Generic algorithms
- Compatibility with ANSI C
- Greater Portability

# C versus C++

**Claimed advantages:**

1. Faster development time (code reuse)
2. Creating/using new data types is easier
3. Memory management: easier more transparent
4. Stricter syntax & type checking => less bugs
5. Data hiding easier to implement
6. OO concepts in C++

# Procedural Programming

- The original programming paradigm is

  Decide which procedures you want;
  use the best algorithms you can find

- The focus is on the processing - the algorithm needed to perform the desired computation
- Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions
- The literature related to this way of thinking is filled with discussion of ways to pass arguments, ways to distinguish different kinds of arguments, different kinds of functions, etc.

# Modular Programming

- The emphasis in the design of programs has shifted from the design of procedures and toward the organization of data

Decide which modules you want;
partition the program so that data is hidden in modules

a set of related procedures
with data they manipulate

This paradigm is also known as ***data-hiding principles***

# Object-Oriented Programming

- Objects
  - Reusable software components that model real world items
  - Meaningful software units
    - Date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects, etc.
    - Any noun can be represented as an object
  - It is claimed that this is more understandable, better organized and easier to maintain than procedural programming
  - Favors modularity

# Data Abstraction

- User-Defined Types
  - C++ allows a user to directly define types that behave in (nearly) the same way as built-in types
  - Such a type is often called an *abstract data type*
  - The programming paradigm is:

*Decide which types you want;*
*provide a full set of operations for each type*

More on this next session!

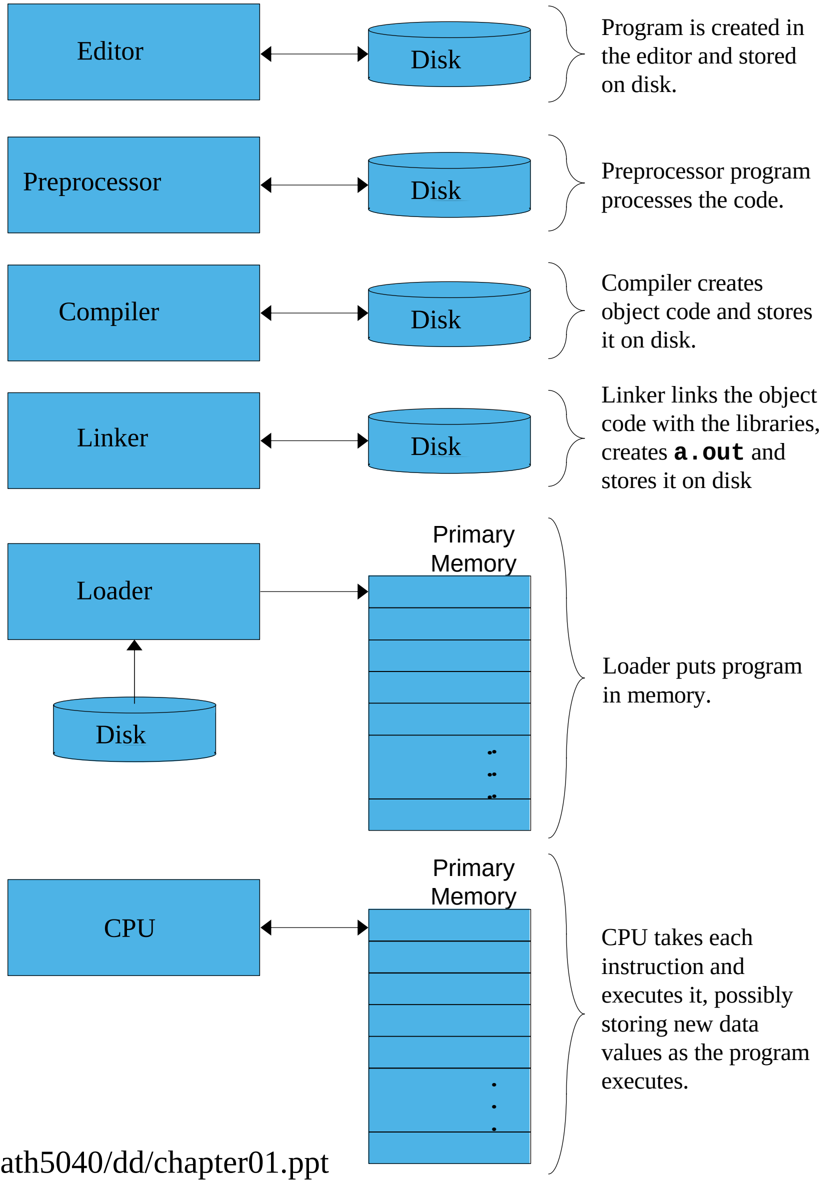
# C++ OOP Features

- C++ is a hybrid language (can program in procedural or OO fashion or both)
- Concepts for Object Orientation
  - Encapsulation (Data-hiding)
  - Inheritance
  - Polymorphism
  - Ability to be dynamic (concept of objects)
- (see next session)

# A Typical C++ Environment

Running C++ Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute



Program is created in the editor and stored on disk.

Preprocessor program processes the code.

Compiler creates object code and stores it on disk.

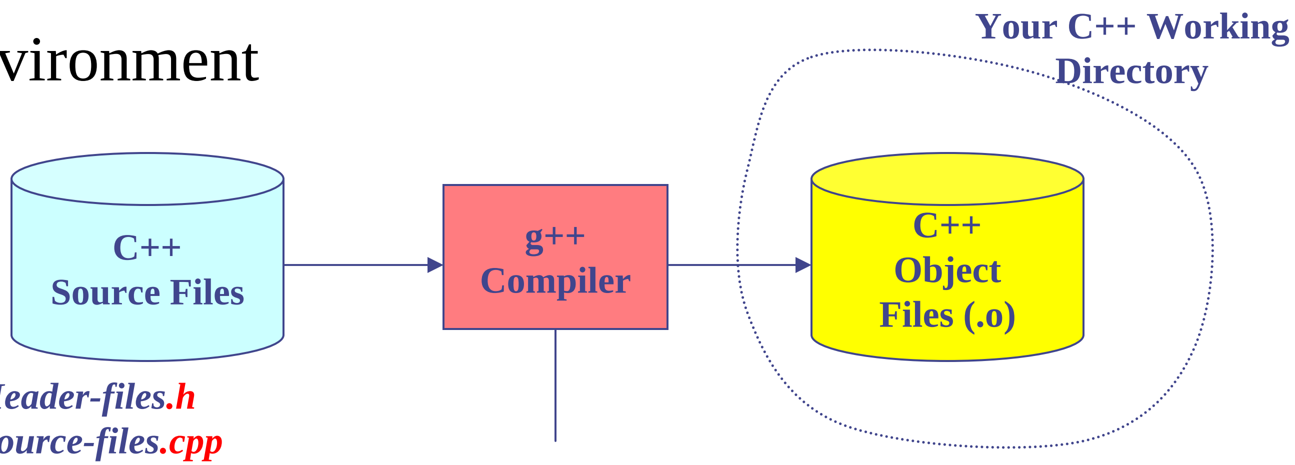
Linker links the object code with the libraries, creates **`a.out`** and stores it on disk

Loader puts program in memory.

CPU takes each instruction and executes it, possibly storing new data values as the program executes.

# Getting Started

## The C++ Environment



*Header-files.h*
*Source-files.cpp*

.h files contain the class interfaces

.cpp files contain the class implementation details

The program entry point is the main() function that must be defined in a .cpp file.

```
>g++ -c filename.cpp
```

One or more compilation-only steps. Each successful compilation produces a .o file. Only .cpp files are compiled. .h files will be compiled when a #include statement is processed.

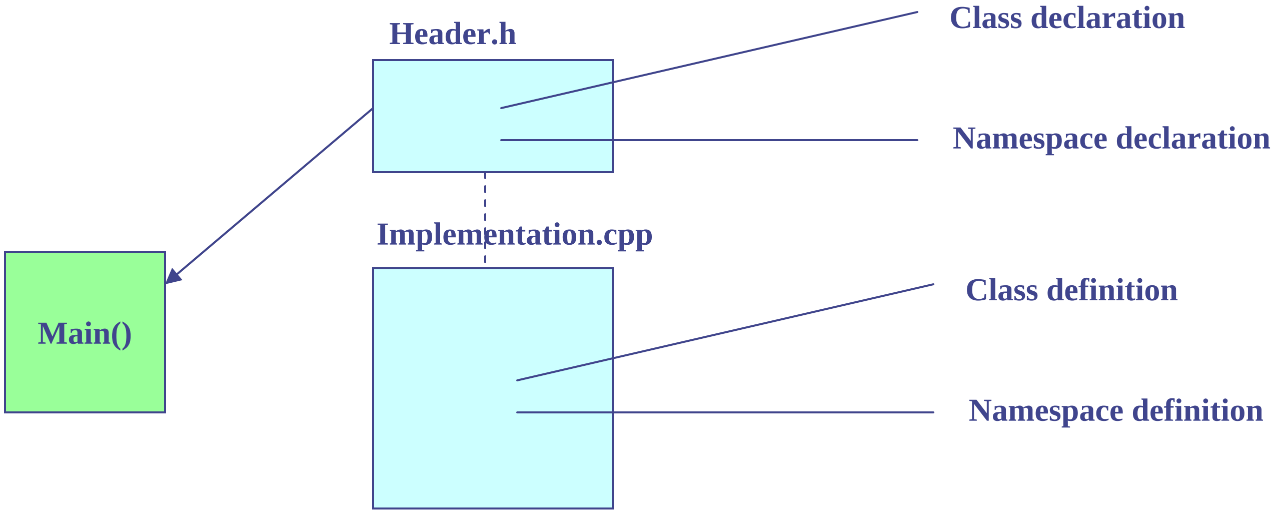
```
>g++ -c filename.cpp >& filename.msg
```

This illustrates how to redirect *stderr* to a file to save compilation messages.

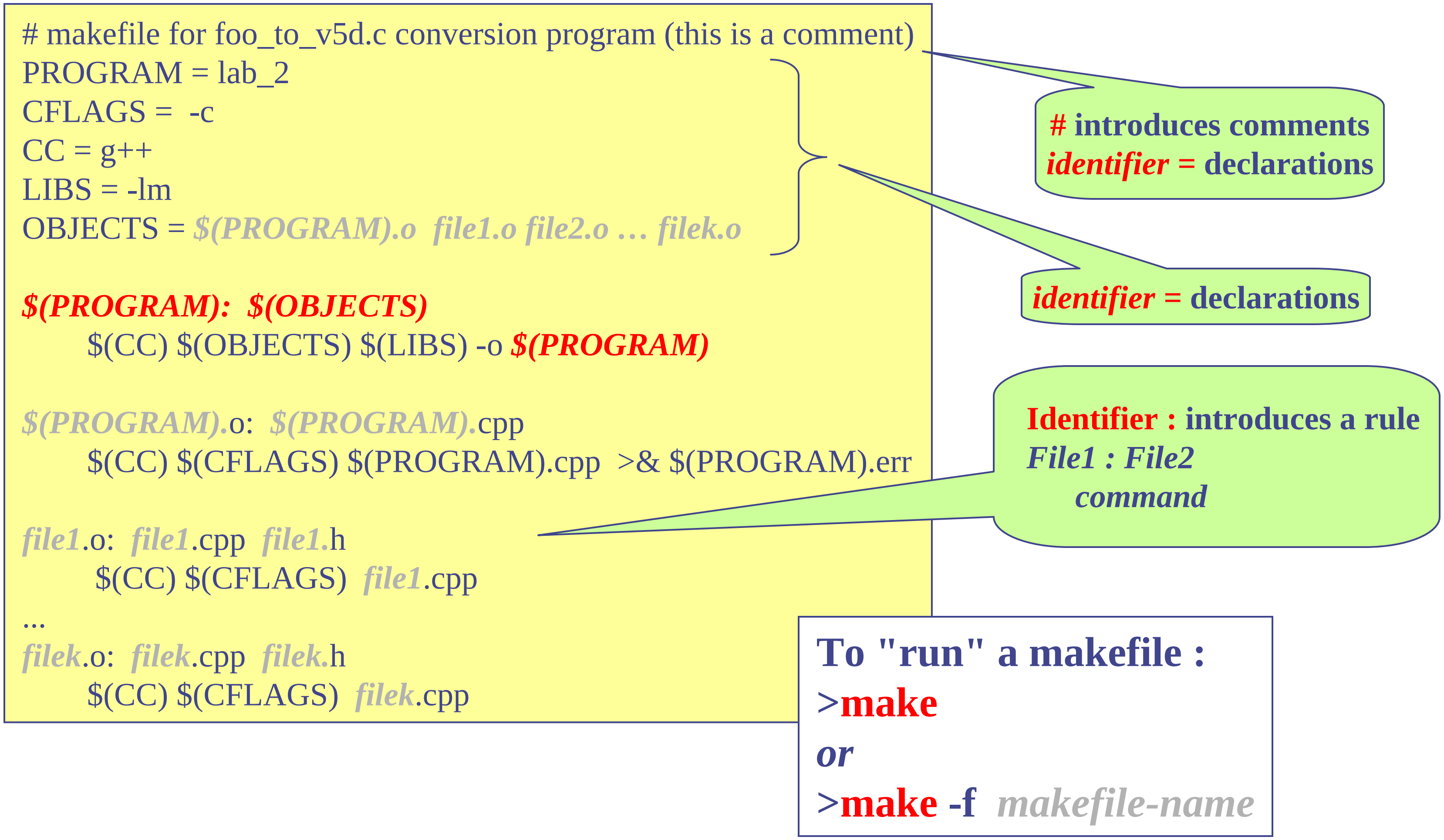
```
>g++ mainfile.o file1.o file2.o … -o executable-name
```

Compile the main file, link .o files, and produce an executable.

# C++ Program Structure



# Make Files



```
# makefile for foo_to_v5d.c conversion program (this is a comment)
PROGRAM = lab_2
CFLAGS =  -c
CC = g++
LIBS = -lm
OBJECTS = $(PROGRAM).o  file1.o file2.o … filek.o

$(PROGRAM):  $(OBJECTS)
	$(CC) $(OBJECTS) $(LIBS) -o $(PROGRAM)

$(PROGRAM).o:  $(PROGRAM).cpp
	$(CC) $(CFLAGS) $(PROGRAM).cpp  >& $(PROGRAM).err

file1.o:  file1.cpp  file1.h
	 $(CC) $(CFLAGS)  file1.cpp
...
filek.o:  filek.cpp  filek.h
	$(CC) $(CFLAGS)  filek.cpp
```

**# introduces comments**
***identifier*** **= declarations**

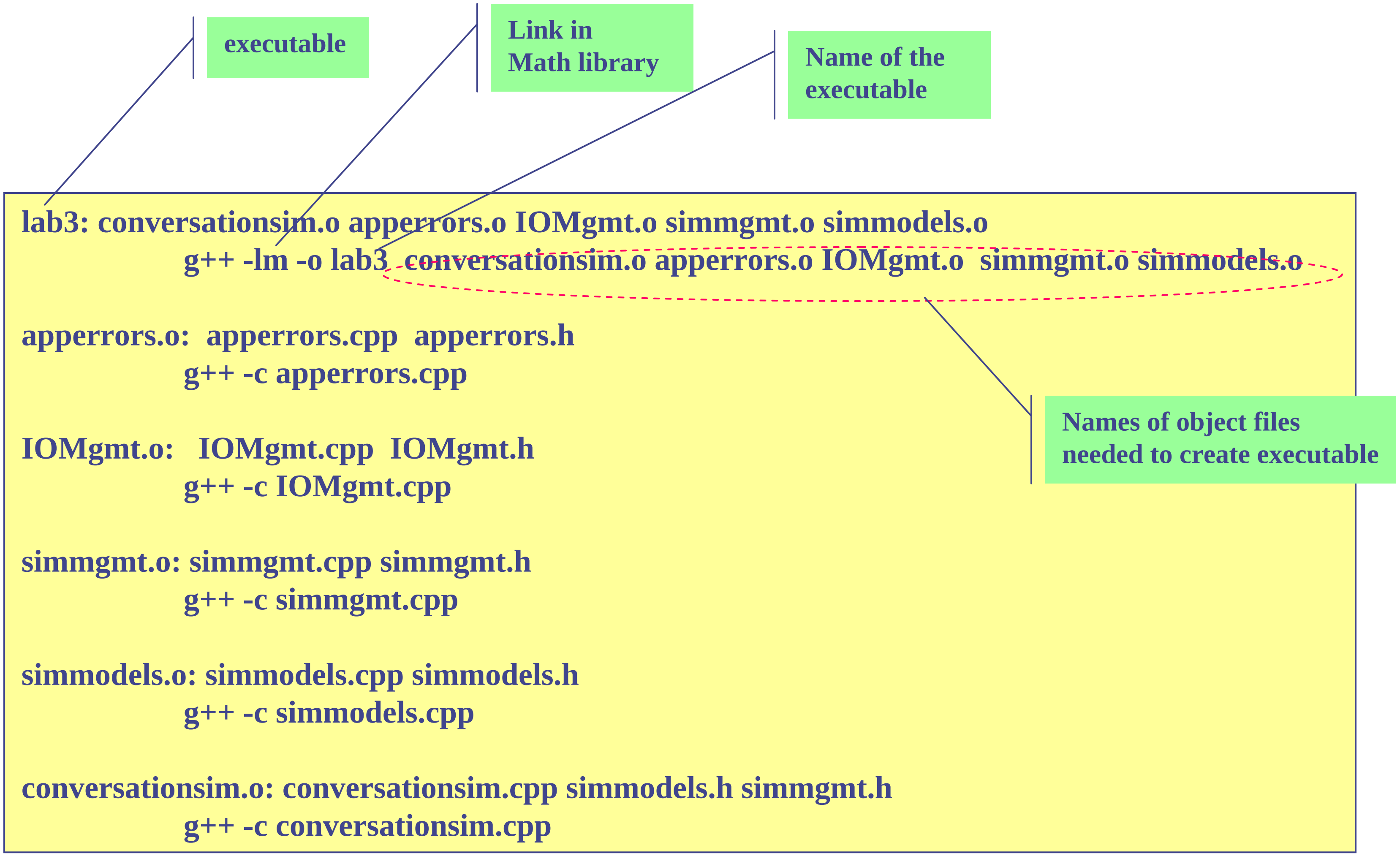
***identifier*** **= declarations**

**Identifier : introduces a rule**
***File1 : File2***
    ***command***

**To "run" a makefile :**
**>make**
***or***
**>make -f** ***makefile-name***

© 2005 David A. Workman, www.cs.ucf.edu/~workman/cop4232/Cpp-Fall05.ppt

# Makefile: Example



executable

Link in Math library

Name of the executable

Names of object files needed to create executable

```
lab3: conversationsim.o apperrors.o IOMgmt.o simmgmt.o simmodels.o
            g++ -lm -o lab3  conversationsim.o apperrors.o IOMgmt.o  simmgmt.o simmodels.o

apperrors.o:  apperrors.cpp  apperrors.h
            g++ -c apperrors.cpp

IOMgmt.o:   IOMgmt.cpp  IOMgmt.h
            g++ -c IOMgmt.cpp

simmgmt.o: simmgmt.cpp simmgmt.h
            g++ -c simmgmt.cpp

simmodels.o: simmodels.cpp simmodels.h
            g++ -c simmodels.cpp

conversationsim.o: conversationsim.cpp simmodels.h simmgmt.h
            g++ -c conversationsim.cpp
```

# Review

- What is a computer program?
- Where are programs entered?
- Where are programs stored after they are typed into the computer?
- What is a compiler?
- What is a linker?
- Where do the statements of your programs reside while your program is executing?
- What numbering system do computers use?
- Explain how the letter A will look in the main memory of the computer.
- How does the computer keep track of data values or programming statements while in main memory?
- What steps does a programmer need to take before they start to write a C++ program?

# A Sample C++ Program

```
#include <iostream.h>
int main()
// total pay for employee
{
  double rate;
  int hours;
  double pay;

  cout << "Enter the hourly rate for the employee ";
  cin >> rate;
  cout << "Enter the number of hours: ";
  cin >> hours;
  pay=rate*hours;
  cout << "Total pay is "<< pay << endl;
  return 0;
}
```

# Program Layout

To begin the main function of the program

```
int main()
{
```

- To end the main function

```
return 0;
}
```

Main function ends with a return statement

# Includes

Include Directives

**#include <iostream>**

- Tells compiler where to find information about items used in the program
- iostream is a library containing definitions of cin and cout

# Comments

- Can use C form of comments `/* a comment */`
- Can also use `//` form:
  - when `//` encountered, remainder of line ignored
  - works only on that line

# Variable Declaration

- **Variables are declared before they are used**
  - Typically variables are declared at the beginning of the program
  - Statements (not always lines) end with a semi-colon
- **Variable declaration line**

```
double rate;
int hours;
double pay;
```

  - int means that the variables represent integers
  - double means the variables represent a number with a fractional component

# cout

Program statement

```
cout << "Enter the hourly rate for the employee ";
```

- cout (see-out) used for output to the monitor
- “<<“ inserts “Enter the …for the employee” in the data bound for the monitor
- Think of cout as a name for the monitor
  - “<<“ points to where the data is to end up

# cin

Program statement

```
cin >> rate;
```

- cin (see-in) used for input from the keyboard
- “>>” extracts data from the keyboard
- Think of cin as a name for the keyboard
  - “>>” points from the keyboard to a variable where the data is stored

# Arithmetic

- Program statement

```
pay=rate*hours;
```

  - Performs a computation
  - ‘*’ is used for multiplication
  - ‘=‘ causes pay to get a new value based on
    the calculation shown on the right of the equal sign

# Text Formatting

- Compiler accepts almost any pattern of line breaks and indentation
- Programmers format programs so they are easy to read
    - Place opening brace ‘{‘ and closing brace ‘}’ on a line by themselves
    - Indent statements
    - Use only one statement per line

# Testing and Debugging

- Bug
  - A mistake in a program
- Debugging
  - Eliminating mistakes in programs
  - Term used when a moth caused a failed relay on the Harvard Mark 1 computer. Grace Hopper and other programmers taped the moth in logbook stating: “First actual case of a bug being found.”

# Program Errors

- Syntax errors
  - Violation of the grammar rules of the language
  - Discovered by the compiler
    - Error messages may not always show correct location of errors
- Run-time errors
  - Error conditions detected by the computer at run-time
- Logic errors
  - Errors in the program's algorithm
  - Most difficult to diagnose
  - Computer does not recognize an error

# Variables and Assignments

- Variables are like small blackboards
  - We can write a number on them
  - We can change the number
  - We can erase the number
- C++ variables are names for memory locations
  - We can write a value in them
  - We can change the value stored there
  - We cannot erase the memory location
    - Some value is always there

# Identifiers

- Variables names are called identifiers
- Choosing variable names
  - Use meaningful names that represent data to be stored
  - C++ is a case-sensitive language!
  - First character must be
    - a letter
    - the underscore character
  - Remaining characters must be
    - letters
    - numbers
    - underscore character

# Keywords

- Keywords (also called reserved words)
  - Are used by the C++ language
  - Must be used as they are defined in the programming language
  - Cannot be used as identifiers

# Declaring Variables

- Before use, variables must be declared
- Two locations for variable declarations

- Immediately prior to use

  (new in C++)

```
int main()
{
    …
    int sum;
    sum = score1 + score 2;
    …
    return 0;
  }
```

- At the beginning

  (C style)

```
int main()
{
     int sum;
    …
    sum = score1 + score2;
     …
    return 0;
 }
```

# Assignment Statements

- An assignment statement changes the value of a variable
  - pay=rate*hours;
    - pay is set to the quotient of rate and hours
  - Assignment statements end with a semi-colon
  - The single variable to be changed is always on the left of the assignment operator '=‘
  - On the right of the assignment operator can be
    - Constants -- age = 21;
    - Variables -- my_cost = your_cost;
    - Expressions -- circumference = diameter * 3.14159;

# Assignment Statements

- The ‘=‘ operator in C++ is not an equal sign
  - The following statement cannot be true in algebra

```
number_of_bars =  number_of_bars  +  3;
```

  - In C++ it means the **new** value of number_of_bars is the **previous** value of number_of_bars plus 3

# Initializing Variables

- Declaring a variable does not give it a value
  - Giving a variable its first value is **initializing** the variable
- Variables are initialized  in assignment statements

```
double mpg;        // declare the variable
mpg = 26.3;         // initialize the variable
```

# Constants

- Used for:
  - Value substitution
    - C: #define BUFSIZE 100 -> C++: const int bufsize = 100;
  - Safety constants
    - When the value of the variable shouldn’t change
- Constants must be initialized when declared

# Constants

- Pointers and references
  - const Fred* p
    - p cannot be used to change an object p pointing to
  - Fred* const p
    - Pointer value of p cannot be changed, but the object p pointing to can be changed
  - const Fred* const p
    - Nothing can be changed.
  - const Fred& p
    - p cannot be used to change an object that p is referencing

# Constants

- More places where constants can be used:
  - Function arguments & return values
  - Constants and classes
    - Constant members
    - Constant member functions

(see next session)

# C++ Standard Library

- C++ programs
  - Built from pieces called classes and functions
- C++ standard library
  - Provides rich collections of existing classes and functions for all programmers to use
  - Example: iostream.h

# *Using* Statements

- **`using`** statements
  - Eliminate the need to use the **`std::`** prefix
  - Allow us to write cout instead of **`std::cout`**
  - To use the following functions without the **`std::`** prefix, write the following at the top of the program

```
using std::cout;
using std::cin;
using std::endl:
```

- Note: Not needed when sourcing iostream.*h* instead of iostream
- See also “Namespaces” section in next session

# Input and Output

- A **data stream** is a sequence of data
  - Typically in the form of characters or numbers

- An input stream is data for the program to use
  - Typically originates
    - at the keyboard
    - at a file

- An output stream is the program’s output
  - Destination is typically
    - the monitor
    - a file

# Output Using cout

- cout is an output stream sending data to the monitor
- The insertion operator "<<" inserts data into cout
- Example:

  ```
  cout << number_of_bars << " candy bars\n";
  ```

  - This line sends two items to the monitor
    - The value of number_of_bars
    - The quoted string of characters " candy bars\n"

      Notice the space before the ‘c’ in candy

      The ‘\n’ causes a new line to be started following the ‘s’ in bars

    - A new insertion operator is used for each item of output

# Output Using cout

- This produces the same result as the previous example

```
cout << number_of_bars ;
cout << " candy bars\n";
```

- Here arithmetic is performed in the cout statement

```
cout << "Total cost is $" << (price + tax);
```

- Quoted strings are enclosed in double quotes ("Walter")
  - Don’t use two single quotes (')
- A blank space can also be inserted with

```
cout << " " ;
```

  if there are no strings in which a space is desired as in " candy bars\n"

# Escape Sequences

- Escape sequences tell the compiler to treat characters in a special way
- '\' is the escape character
  - To create a newline in output use

    \n – cout << "\n";

    or the newer alternative

    cout << endl;

© 2003 Pearson Education, from http:// www.eecs.lehigh.edu/~kalafut/cse17/Introduction.ppt

# Escape Sequences

| Escape Sequence | Description |
|---|---|
| **`\n`** | Newline. Position the screen cursor to the beginning of the next line. |
| **`\t`** | Horizontal tab. Move the screen cursor to the next tab stop. |
| **`\r`** | Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. |
| **`\a`** | Alert. Sound the system bell. |
| **`\\`** | Backslash. Used to print a backslash character. |
| **`\"`** | Double quote. Used to print a double quote character. |

# Formatting Real Numbers

- Real numbers (type double) produce a variety of outputs

```
double price = 78.5;
cout << "The price is $" << price << endl;
```

  - The output could be any of these:

```
The price is $78.5
          The price is $78.500000
          The price is $7.850000e01
```

  - The most unlikely output is:

```
The price is $78.50
```

# Showing Decimal Places

- cout includes tools to specify the output of type double

- To specify fixed point notation
  - setf(ios::fixed)
- To specify that the decimal point will always be shown
  - setf(ios::showpoint)
- To specify that two decimal places will always be shown
  - precision(2)

- Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout      << "The price is "
          << price << endl;
```

# Input Using cin

- cin is an input stream bringing data from the keyboard
- The extraction operator (>>) removes data to be used
- Example:

```
cout << "Enter the number of bars in a package\n";
cout << " and the weight in ounces of one bar.\n";
cin >> number_of_bars;
cin >> one_weight;
```

- This code prompts the user to enter data then reads two data items from cin
  - The first value read is stored in number_of_bars
  - The second value read is stored in one_weight
  - Data is separated by spaces when entered

# Reading Data From cin

- Multiple data items are separated by spaces
- Data is not read until the enter key is pressed
  - Allows user to make corrections
- Example:

```
cin >> v1 >> v2 >> v3;
```

  - Requires three space separated values
  - User might type

```
34  45  12   <enter key>
```

# Designing Input and Output

- Prompt the user for input that is desired
  - cout statements provide instructions

    ```
    cout << "Enter your age: ";
                 cin >> age;
    ```

    - Notice the absence of a new line before using cin
- Echo the input by displaying what was read
  - Gives the user a chance to verify data

    ```
    cout << age << " was entered." << endl;
    ```

# Data Types and Expressions

- 2  and 2.0 are not the same number
  - A whole number such as 2 is of type *int*
  - A real number such as 2.0 is of type *double*

- Numbers of type int are stored as exact values
- Numbers of type double may be stored as approximate values due to limitations on  number of significant digits that can be represented

# Other Number Types

- Various number types have different memory requirements
    - More precision requires more bytes of memory
    - Very large numbers require more bytes of memory
    - Very small numbers require more bytes of memory

# Integer Types

- long  or long int  (often 4 bytes)
  - Equivalent forms to declare very large integers

```
long  big_total;
long int big_total;
```

- short or short int  (often 2 bytes)
  - Equivalent forms to declare smaller integers

```
short small_total;
short int small_total;
```

# Floating Point Types

- long double  (often 10 bytes)
  - Declares floating point numbers with up to 19 significant digits

```
long double big_number;
```

- float  (often 4 bytes)
  - Declares floating point numbers with up to 7 significant digits

```
float not_so_big_number;
```

# Type char

- Computers process character data too
- char
  - Short for character
  - Can be any single character from the keyboard

- To declare a variable of type char:

```
char letter;
```

# char Constants

- Character constants are enclosed in single quotes

```
char letter = 'a';
```

- Strings of characters, even if only one character is enclosed in double quotes
  - "a" is a string of characters containing one character
  - 'a' is a value of type character

# Reading Character Data

- cin skips blanks and line breaks looking for data
- The following reads two characters but skips any space that might be between

```
                char symbol1, symbol2;
        cin  >>  symbol1  >>  symbol2;
```

- User normally separate data items by spaces

```
J    D
```

- Results are the same if the data is not separated by spaces

```
JD
```

# Type bool

- bool is a new addition to C++
  - Short for boolean
  - Boolean values are either true or false

- To declare a variable of type bool:

```
bool old_enough;
```

# Type Compatibilities

- In general store values in variables of the same type
  - This is a type mismatch:

```
int int_variable;
int_variable = 2.99;
```

  - If your compiler allows this, int_variable will most likely contain the value 2, not 2.99

# int ←→ double

- Variables of type double should not be assigned to variables of type int

```
int int_variable;
double double_variable;
double_variable = 2.00;
int_variable = double_variable;
```

  - If allowed, int_variable contains 2, not 2.00

# int ←→ double

- Integer values can normally be stored in variables of type double

```
double double_variable;
double_variable = 2;
```

  - double_variable will contain 2.0

# char ←→ int

- The following actions are possible but generally not recommended!
- It is possible to store char values in integer variables

  int value = 'A';

  value will contain an integer representing 'A'

- It is possible to store int values in char variables

  char letter = 65;

# bool ← → int

- The following actions are possible but generally not recommended!
- Values of type bool can be assigned to int variables
  - True is stored as 1
  - False is stored as 0
- Values of type int can be assigned to bool variables
  - Any non-zero integer is stored as true
  - Zero is stored as false

# Arithmetic

- Arithmetic is performed with operators
  - + for addition
  - - for subtraction
  - * for multiplication
  - / for division

- Example: storing a product in the variable
  total_weight

```
total_weight  =  one_weight * number_of_bars;
```

© 2003 Pearson Education, from http:// www.eecs.lehigh.edu/~kalafut/cse17/Introduction.ppt

# Integer Remainders

- % (modulus) operator gives the remainder from integer division

```
int dividend, divisor, remainder;
dividend = 5;
divisor = 3;
remainder = dividend % divisor;
```

The value of remainder is 2

# Arithmetic

- Arithmetic calculations
  - Use **`*`** for multiplication and **`/`** for division
  - Integer division truncates remainder
    - **`7 / 5`** evaluates to 1
  - Modulus operator returns the remainder
    - **`7 % 5`** evaluates to 2
- Operator precedence
  - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Be sure to use parenthesis when needed
  - Example: Find the average of three variables a, b and c
    - Do not use: **`a + b + c / 3`**
    - Use: **`(a + b + c ) / 3`**

# Arithmetic

- Arithmetic operators:

| C++ operation | Arithmetic operator | Algebraic expression | C++ expression |
|---|---|---|---|
| Addition | `+` | *f + 7* | `f + 7` |
| Subtraction | `-` | *p – c* | `p - c` |
| Multiplication | `*` | *bm* | `b * m` |
| Division | `/` | *x / y* | `x / y` |
| Modulus | `%` | *r mod s* | `r % s` |

- Rules of operator precedence:

| Operator(s) | Operation(s) | Order of evaluation (precedence) |
|---|---|---|
| `()` | Parentheses | Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right. |
| `*`, `/`, or `%` | Multiplication Division Modulus | Evaluated second. If there are several, they re evaluated left to right. |
| `+` or `-` | Addition Subtraction | Evaluated last. If there are several, they are evaluated left to right. |

# Results of Operators

- Arithmetic operators can be used with any numeric type
- An operand is a number or variable used by the operator
- Result of an operator depends on the types of operands
  - If both operands are int, the result is int
  - If one or both operands are double, the result is double

# Arithmetic Expressions

- Use spacing to make expressions readable
  - Which is easier to read?

```
x+y*z      or   x + y * z
```

- Precedence rules for operators are the same as used in your algebra classes
- Use parentheses to alter the order of operations

```
 x + y * z     ( y is multiplied by z first)
(x + y) * z   ( x and y are added first)
```

# If-Else

- The if-else statement in C++ is used to compare two options:

```
if   (condition)
     {statement1; } will execute if the condition  is "True"
else
     {statement2;} will execute if the condition  is "False"
```

# Numerical Values of Conditions

- In relational (logical) expressions (=conditions), the value of the expression can only be an integer value of 1 or 0, which is interpreted as true or false, respectively.

# Equality and Relational Operators

| Standard algebraic equality operator or relational operator | C++ equality or relational operator | Example of C++ condition | Meaning of C++ condition |
|---|---|---|---|
| *Relational operators* | | | |
| > | **`>`** | **`x > y`** | **`x`** is greater than **`y`** |
| < | **`<`** | **`x < y`** | **`x`** is less than **`y`** |
| ≥ | **`>=`** | **`x >= y`** | **`x`** is greater than or equal to **`y`** |
| ≤ | **`<=`** | **`x <= y`** | **`x`** is less than or equal to **`y`** |
| *Equality operators* | | | |
| = | **`==`** | **`x == y`** | **`x`** is equal to **`y`** |
| ≠ | **`!=`** | **`x != y`** | **`x`** is not equal to **`y`** |

# Logical Operators

- AND, OR, NOT operators are called logical operators. These operators are represented by the symbols &&, || and !, respectively.
- When the AND operator, &&, is used with two simple expressions, the condition is true only if both individual expression are true.

# Logical Operators

Here is a compound statement showing the AND operator.

(grade>89 )&& (semester <3)

is a true condition because both condition is evaluated a true condition.

On the other hand, for the logical OR (|| ) operators , the condition is satisfied if either one of the two expression is true. Thus the compound condition

(mile<40 || age>50)

will be true if the mile is less than 40 or age is greater than 50 .

# Loops

- The repetition statement defines the boundaries containing  the repeating section of code and also controls in what form the code will be executed or not.

- The three different forms of loops:
- **while** (expression) {statement; }
- **do** {statement; } **while** (expression)
- **for** (inital statement, expression, increment statement) {statement;}

# Switch

- Switch (case) statements are a substitute for long if statements. The basic format for using switch case is outlined below:

```
switch (variable) {
        case expression1:
            do something 1;
            break;
        case expression2:
            do something 2;
            break;
          ....
        default:
            do default processing;
    }
```

# Arrays

- An array is defined with this syntax:
  *datatype arrayName[size];*
- Elements are numbered 0,…,size-1!
- Examples:

  double temperatures[31];
  */* Could be used to store the daily temperatures in a month */*

  char name[20];
  */* Could be used to store a character string. C-style character strings are terminated be the null character, '\0'. */*

# Example Program

```
#include<iostream.h>

const int MAXNUMBERS=6;
int main( )
{
 int i, number[MAXNUMBERS];
 for(i=0;i<=MAXNUMBERS;  i++) //Enter the numbers
    {
    cout<<”Enter  a number :”
    cin>>number[i];
    }
cout<<endl<<endl;
for(i=0; i<MAXNUMBERS;i++)// Print the numbers
    {
    cout<<”Number ”<<number[i]<<”is” <<NUMBERS[i]<<endl;
    return 0;
    }
}
```

# Example Output

**OUTPUT:**

```
Enter a number:5
Enter a number:6
Enter a number:7
Enter a number:8
Enter a number:9
Enter a number:10
Number 1  is   5
Number 2  is 6
Number 3 is   7
Number 4 is   8
Number 5 is   9
Number 6 is   10
```

# Some Frequent Programming Errors

- Forgetting to close string sent to cout with a double quote
- Omitting or incorrectly typing the opening and closing braces of main and functions
- Omitting necessary semicolon at the end of statement
- Adding a semicolon at the end of the #include directive
- Wrong case or misspelled identifier
- Accidental use of keyword as identifier
- Typing the letter O for the number zero (0) or vice versa
- Forgetting to declare all the variables used in a program

# Resources and Further Reading

WWW:

http://www.desy.de/gna/html/cc/Tutorial/tutorial.html
http://www.cs.fit.edu/~mmahoney/cse2050/introcpp.html
http://www.acm.org/crossroads/xrds1-1/ovp.html
http://www.thefreecountry.com/compilers/cpp.shtml

Textbook this lecture is based on:

Bjarne Stroustrup, *The C++ Programming Language*,
3rd Ed,, Addison Wesley, 1997.