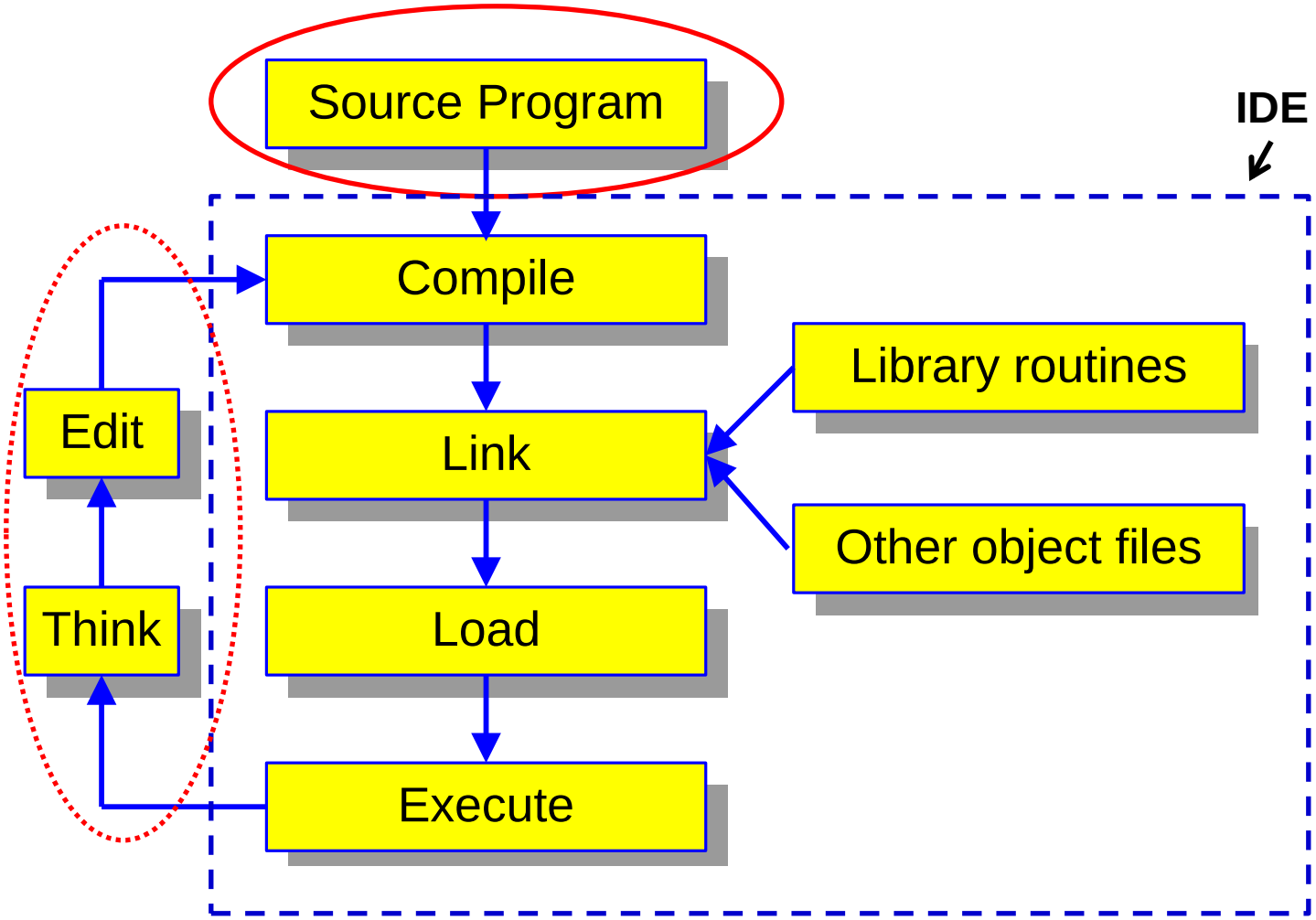
Introduction to Programming

Mohamed Saied

Section 1 - The Basics

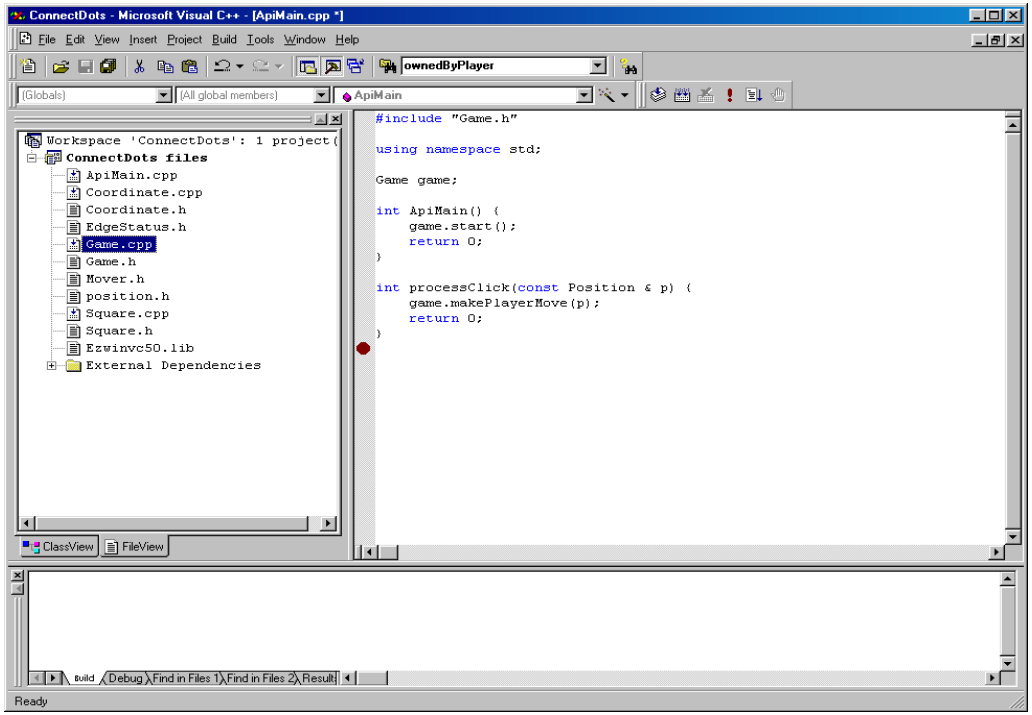
Software Development Cycle



IDEs

- **Integrated Development Environments or IDEs**
  - **Supports the entire software development cycle**
    - **E.g. MS Visual C++, Borland, Code Warrior**

- **Provides all the capabilities for developing software**
  - **Editor**
  - **Compiler**
  - **Linker**
  - **Loader**
  - **Debugger**
  - **Viewer**



What are the Key Ingredients of a Program?

- **Common elements in programming languages:**
  - **Keywords (aka reserved words)**
    - Defined in C++. Have a special meaning
  - **Programmer-defined entities**
    - Names made up by programmer
    - Used to represent variables, functions, etc.
  - **Operators**
    - Defined in C++
    - For performing arithmetic, comparison, etc. operations
  - **Constructs**
    - Defined in C++
    - Sequence, selection, repetition, functions, classes, etc.

Every programming language has a syntax (or grammar) that a programmer must observe

The syntax controls the use of the elements of the program

Object Oriented Programming

- **C++ is derived from C programming language – an extension that added object-oriented (OO) features.**

  - Lots of C++ syntax similar to that found in C, but there are differences and new features.

- **Popular because of the reuse of objects**
  - **Classes – template or blueprint for an object**
  - **Objects**
  - **Methods**

- **Objects are organized in a hierarchy**
  - **Super Classes**
  - **Sub Classes**

The Evolution of C++

- C (1972)
- ANSI Standard C (1989)
- Bjarne Stroustrup adds features of the language Simula (an object-oriented language designed for carrying out simulations) to C resulting in …
- C++ (1983)
- ANSI Standard C++ (1998)
- ANSI Standard C++ [revised] (2003)
- ANSI Standard C++ (2011)

- The present C++
  - A general-purpose language that is in widespread use for systems and embedded
  - The most commonly used language for developing system software such as databases and operating systems

… the future: another Standard (2014 and 2017?)

C++ - an Object-Oriented Programming Language

Other examples: C# and Java

The modern approach to developing programs

- Objects in a real-world problem are modeled as software objects

Warning: OO features were added to C++ as an "afterthought"

- It is not a "true" OO language such as C# or Java

Program Organization

- **Program statement**
  - Definition, e.g. function prototype
  - Declaration, e.g. variables and constants
  - Action
- **Executable unit - Action**
  - Set of program statements – may be named
  - Different languages refer to named executable units by different names
    - Subroutine: Fortran and Basic
    - Procedure: Pascal
    - Function: C++
    - Method: C++, C#, Java

The simplest C++ program consists of a single function named main.

The syntax of such programs is shown below:

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

int main()
   {
   declaration(s)
   statement(s)
   return 0;
   }
```

The portions of the program shown in blue should always be present. The declarations specify the data that is used by the program. These declarations can declare either constants or variables, for example.

The statements specify the algorithm for the solution to your problem.

Example – Algorithm for Converting Miles to Kilometers

- **Problem Input**

  miles *distance in miles*

- **Problem Output**

  kms *distance in kilometers*

- **Algorithm will use Conversion Formula**

  1 mile = 1.609 kilometers

- **Formulate the algorithm that solves the problem.**

- **Algorithm**
  1. **Get the distance in miles.**
  2. **Convert the distance to kilometers by multiplying by 1.609**
  3. **Display the distance in kilometers.**

- **Now convert the algorithm to program code!**

```
// miles.cpp
// Converts distance in miles to kilometers.

#include <iostream>
using namespace std;

int main()                                  // start of main function
{
      const float KM_PER_MILE = 1.609;      // 1.609 km in a mile
      float miles,                          // input: distance in miles
            kms;                            // output: distance in kilometers

      // Get the distance in miles.
      cout << "Enter the distance in miles: ";
      cin >> miles;


      // Convert the distance to kilometers.
      kms = KM_PER_MILE * miles;

      // Display the distance in kilometers.
      cout << "The distance in kilometers is " << kms << endl;

      return 0;                                 //Exit the main function
}
```

```
Enter the distance in miles: 10.0
The distance in kilometers is 16.09
```

Statements

Statements are the executable "units" of a program.

Statements are the translation of an algorithm into program code.

Statements may be

- **Simple – typically one line of program code**

- **Structured – a grouping of a number of statements**
  E.g. control structures; functions; etc.

Statements must be terminated with a ; symbol.

Program statements may be executed by one of three control structures:

- **Sequence – execute statements one after the other**

- **Selection – select which statement(s) to execute**

- **Repetition – repeatedly execute statement(s)**

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

  int main()
     {
     declaration(s)
     statement(s)
     return 0;
     }
```

The #include Directive

- Preprocessor directive
- Inserts the contents of another file into the program

  iostream is C++ library of input/output functions

  This includes cout and cin
- Do not use ; at end of #include statement

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

  int main()
     {
     declaration(s)
     statement(s)
     return 0;
     }
```

The namespace Directive

This directive allows the cout and cin statements to be used in a program without using the prefix std::

With this directive, may write

cin or cout

Otherwise, we would have to write

std::cout or std::cin

Must use ; at end of namespace directive

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

  int main()
     {
     declaration(s)
     statement(s)
     return 0;
     }
```

Declarations - Constants and Variables

Constants are data that having unchanging values.

Variables, as their name implies, are data whose values can change during the course of execution of your program.

For both constants and variables, the name, the data type, and value must be specified.

Any variable in your program must be defined before it can be used

Data type specifies whether data is integral, real, character or logical.

Constants

Syntax

```
const  type  name = expression;
```

The statement must include the reserved word const, which designates the declaration as a constant declaration.

The type specification is optional and will be assumed to be integer.

Examples

```
const float TAXRATE = 0.0675;
const int NUMSTATES = 50;
```

Convention is to use uppercase letters for names of constants.

The data type of a constant may be one the types given below

Integer	a sequence of digits
Real	a sequence of digits containing a decimal point
Character	a single character enclosed in single quotation marks
Logical	one of the reserved words `true` or `false`

A Simple, Yet Complete, C++ Program

Program producing output only

```
// Hello world program

#include <iostream>
using namespace std;

int main()
 {
 cout << "Hello world!" << endl;
 return 0;
 }
```

Comment

Preprocessor directives

Function named main() indicates start of the program

Ends execution of main() which ends the program

Let us look at the function main()

```
int main()
        {
        cout << "Hello world!" << endl;
        return 0;
        }
```

This example introduces the notion of a text string

"Hello world!"

The basic data types supported by C++ are shown in the table below

Integer	`short int long`
Real	`float double`
Character	`char`
Logical	`bool`

The three integer types allow for whole numbers of three different sizes.
The two real types permit numbers with decimal parts with two different amounts of precision.
Finally, there is one type for character data that consists of a single character, and one type for logical values that are either true or false.

The cout Output Statement

- Used to display information on computer screen
- It is declared in the header file iostream
- Syntax

```
cout << expression;
```

- Uses << operator to send information to computer screen

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

- Can be used to send more than one item

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

- **To get multiple lines of output on screen**

  Either

  - Use the endl function

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

  Or

  - Use \n in the output string

```
cout << "Hello, there!\n";
```

Example

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

int main()
 {
 cout <<  "    *     " << endl;
 cout <<  "  ***   " << endl;
 cout << " ***** " << endl;
 cout << "    *     " << endl;
 cout << "    *     " << endl;
cout << "    *     " << endl;
 }
```

Example

```
#include <iostream>
using namespace std;
int main() {
  double radius;
  double area;

  // Step 1: Set value for radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << endl;
}
```

The cin Statement

- Used to read input from keyboard
- It is declared in the header file iostream
- Syntax

  cin >> variable

- Uses >> operator to receive data from the keyboard and assign the value to a variable
- Often used with cout to display a user prompt first
- May use cin to store data in one or more variables

- **Can be used to input more than one value**

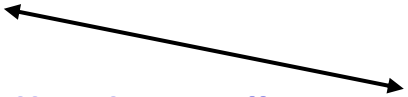
```
cin >> height >> width;
```

- **Multiple values from keyboard must be separated by spaces**

- **Order is important: first value entered goes to first variable, etc.**

Example Program – Use of cout to prompt user to enter some data

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "What is your name? ";
       cin >> name;

    cout << "Hello there, " << name;
    return 0;
}
```



Names

- Used to denote language- and programmer-defined elements within the program

- A valid name is a sequence of
  - Letters (upper and lowercase)
  - Digits
    - A name cannot start with a digit
  - Underscores
    - A name should not normally start with an underscore

- Names are case sensitive
  - MyObject is a different name than MYOBJECT

- There are two kinds of names
  - Keywords
  - Identifiers

Keywords

- Keywords are words reserved as part of the language - syntax
  Example int, return, float, double
- They cannot be used to name programmer-defined elements
- They consist of lowercase letters only
- They have special meaning to the compiler

Identifiers

- **Identifiers are the names given to programmer-defined entities within the program – variables, functions, etc.**

- **Identifiers should be**
  - **Short enough to be reasonable to type (single word is the norm)**
    - **Standard abbreviations are fine**
  - **Long enough to be understandable**
    - **When using multiple word identifiers capitalize the first letter of each word – but this is just a convention**

- **Examples**
  - **Min**
  - **Temperature**
  - **CameraAngle**
  - **CurrentNbrPoints**

Example - Valid and Invalid Identifiers

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

Integer Data Types

- Designed to hold whole numbers
- Can be signed or unsigned
  - 12 -6 +3
- Available in different sizes (in memory): short, int, and long
- Size of short ≤ size of int ≤ size of long

Floating-Point Data Types

- Designed to hold real numbers

  12.45 -3.8

- Stored in a form similar to scientific notation

- All numbers are signed

- Available in different sizes (space in memory: float, double, and long double

- Size of float ≤ size of double ≤ size of long double

Numeric Types in C++

Number Types	
Type	**Typical Range**
`int`	–2,147,483,648 . . . 2,147,483,647 (about 2 billion)
`unsigned`	0 . . . 4,294,967,295
`short`	–32,768 . . . 32,767
`unsigned short`	0 . . . 65,535
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits

Defining Variables

- **Variables of the same type can be defined**

  - **On separate lines:**

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

  - **On the same line:**

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

- **Variables of different types must be in different definitions**

Declaring Variables

```
int x;        // Declare x to be an
              // integer variable;

double radius; // Declare radius to
              // be a double variable;

char a;       // Declare a to be a
              // character variable;
```

Variable Assignments and Initialization

Assignment

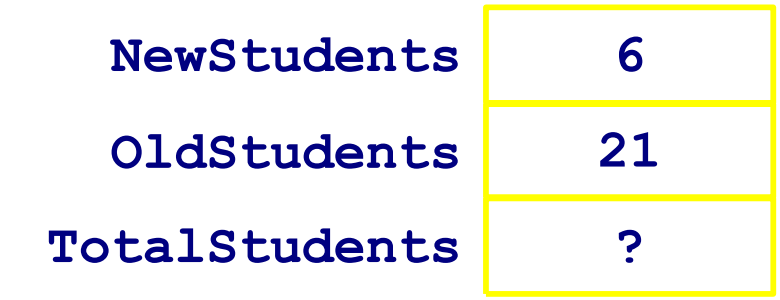
- Uses the = operator
- Has a single variable on the left side and a value (constant, variable, or expression) on the right side
- Copies (i.e. assigns) the value on the right to the variable on the left. (An expression is first evaluated)
- Syntax

```
variable = expression ;
```

```
item = 12;  // constant
Celsius = (Fahrenheit - 32) * 5 / 9;  // expression
y = m * x + b;  // expression
```

Assignment Statement

```
int NewStudents = 6;
int OldStudents = 21;
int TotalStudents;
```




```
TotalStudents = NewStudents + OldStudents;
```

Variable Assignments and Initialization

Initialization

- Initialize a variable: assign it a value when it is defined:

  ```
  int length = 12;
  ```

- Or initialize it later via assignment

- Can initialize some or all variables:

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

- A variable cannot be used before it is defined

Arithmetic Operators

- **Used for performing numeric calculations**

Basic Arithmetic Operators

Operator	Meaning
+	addition
–	subtraction
*	multiplication
/	division
%	modulus

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

/ Operator

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

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

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

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

% Operator

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

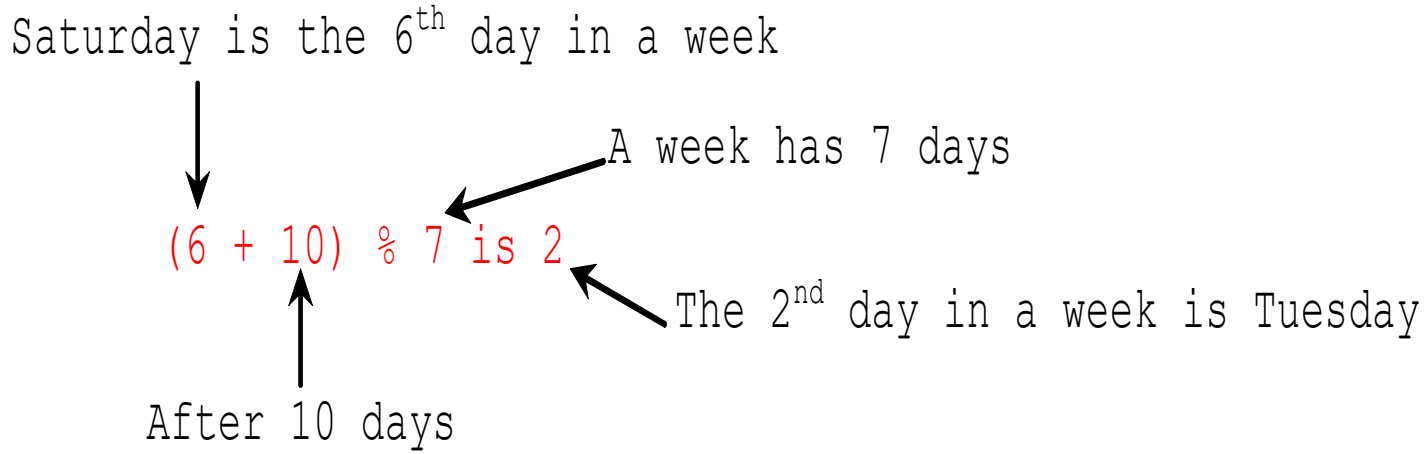
```
cout << 13 % 5;   // displays 3
```

- % requires integers for both operands

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

Remainder Operator

Remainder is very useful in programming.
For example, an even number % 2 is always 0 and an odd number % 2 is always 1.
So you can use this property to determine whether a number is even or odd.
Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression:



Expressions

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

```
area = 2 * PI * radius;

cout << "border length of rectangle is: " << 2*(l+w);
```

Expressions

Expressions are a grouping of variables, constants and operators.

C++ has a defined order of precedence for the order in which operators are used in an expression

- It is not just a left-to-right evaluation

Brackets (aka parentheses) can be used to change the defined order in which operators are used.

Example

```
#include <iostream>
using namespace std;
int main()
{
  cout << (1 + 2 + 3) / 3 << endl; // not the same as 1+2+3/3
  return 0;
}
```

Expressions

- **Multiplication requires an operator:**

  Area=lw **is written as** **Area = l * w;**

- **There is no exponentiation operator:**

  Area=s^2 **is written as** **Area = pow(s, 2);**

- **Parentheses may be needed to specify the order of operations**

  Example

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

is written as **m = (y2-y1) /(x2-x1);**

Note the use of brackets to group variables into expressions and inform the compiler the order of evaluation.

Example

```
m = (y2-y1) /(x2-x1);
```

Compound Assignment

- **C++ has a set of operators for applying an operation to an object and then storing the result back into the object**

- **Examples**

```
int i = 3;
i += 4;              // i is now 7   Equivalent to i = i + 4


float a = 3.2;
a *= 2.0;            // a is now 6.4   Equivalent to a = a * 2.0
```

Shorthand Assignment Operators

Operator	Example	Equivalent
+=	i += 8	i = i + 8
-=	f -= 8.0	f = f - 8.0
*=	i *= 8	i = i * 8
/=	i /= 8	i = i / 8
%=	i %= 8	i = i % 8

Assignment Conversions

- **Floating-point expression assigned to an integer object is truncated**

- **Integer expression assigned to a floating-point object is converted to a floating-point value**

- **Consider**

```
float y = 2.7;
int i = 15;
int j = 10;
i = y;          // i is now 2

y = j;          // y is now 10.0
```

Operators and Precedence

Consider m*x + b - Which of the following is it equivalent to?

- **(m * x) + b** - **Equivalent**
- **m * (x + b)** - **Not equivalent but valid!**

Operator precedence tells how to evaluate expressions

Standard precedence order

- **()** — **Evaluate first. If nested innermost done first**
- *** / %** — **Evaluate second. If there are several, then evaluate from left-to-right**
- **+ -** — **Evaluate third. If there are several, then evaluate from left-to-right**

Operator Precedence

- **Example**

20 - 4 / 5 * 2 + 3 * 5 % 4

(4 / 5)

((4 / 5) * 2)

((4 / 5) * 2) (3 * 5)

((4 / 5) * 2) ((3 * 5) % 4)

(20 -((4 / 5) * 2)) ((3 * 5) % 4)

(20 -((4 / 5) * 2)) + ((3 * 5) % 4)

Increment and Decrement Operators

- **C++ has special operators for incrementing or decrementing a variable's value by one – "strange"effects!**
- **Examples**

```
int k = 4;
++k;                    // k is 5
k++;                    // k is 6   Nothing strange so far!
```

BUT

```
int i = k++;            // i is 6, and k is 7

int j = ++k;            // j is 8, and k is 8
```

Increment and Decrement Operators

The "strange"effects appear when the operators are used in an assignment statement

Operator	Name	Description
a=++var	pre-increment	The value in var is incremented by 1 and this new value of var is assigned to a.
a=var++	post-increment	The value in var is assigned to a and then the value in var is incremented by 1

Similar effects for the decrement operator

Operator	Name	Description
a=--var	pre-decrement	The value in var is decremented by 1 and this new value of var is assigned to a.
a=var--	post-decrement	The value in var is assigned to a and then the value in var is decremented by 1

Some Standard Libraries

fstream

- File stream processing

assert

- Assertion processing

iomanip

- Formatted input/output (I/O) requests

ctype

- Character manipulations

cmath

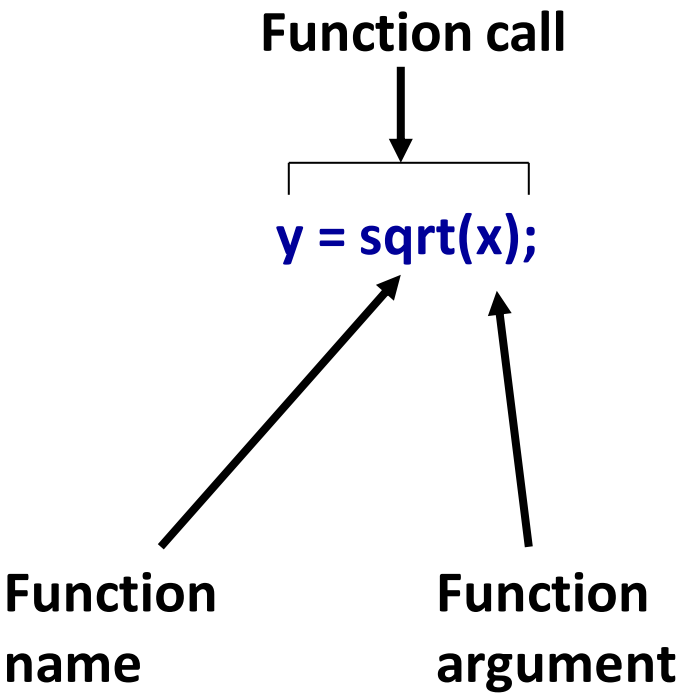
Trigonometric and logarithmic functions

Note

C++ has many other libraries

C++ cmath Library

- **Typical mathematical functions**
  e.g. sqrt, sin, cos, log
- **Function use in an assignment statement**



Some Mathematical Library Functions

Function	Standard Library	Purpose: Example	Argument(s)	Result
`abs(x)`	`<cstdlib>`	Returns the absolute value of its integer argument: if `x` is `-5`, `abs(x)` is `5`	`int`	`int`
`ceil(x)`	`<cmath>`	Returns the smallest integral value that is not less than `x`: if `x` is `45.23`, `ceil(x)` is `46.0`	`double`	`double`
`cos(x)`	`<cmath>`	Returns the cosine of angle `x`: if `x` is `0.0`, `cos(x)` is `1.0`	`double` (radians)	`double`
`exp(x)`	`<cmath>`	Returns e^x where e = `2.71828...`: if `x` is `1.0`, `exp(x)` is `2.71828`	`double`	`double`
`fabs(x)`	`<cmath>`	Returns the absolute value of its type `double` argument: if `x` is `-8.432`, `fabs(x)` is `8.432`	`double`	`double`
`floor(x)`	`<cmath>`	Returns the largest integral value that is not greater than `x`: if `x` is `45.23`, `floor(x)` is `45.0`	`double`	`double`

`log(x)`	`<cmath>`	Returns the natural logarithm of `x` for `x > 0.0`: if `x` is `2.71828`, `log(x)` is `1.0`	`double`	`double`
`log10(x)`	`<cmath>`	Returns the base-10 logarithm of `x` for `x > 0.0`: if `x` is `100.0`, `log10(x)` is `2.0`	`double`	`double`
`pow(x, y)`	`<cmath>`	Returns x^y. If `x` is negative, `y` must be integral: if `x` is `0.16` and `y` is `0.5`, `pow(x, y)` is `0.4`	`double`, `double`	`double`
`sin(x)`	`<cmath>`	Returns the sine of angle `x`: if `x` is `1.5708`, `sin(x)` is `1.0`	`double` (radians)	`double`
`sqrt(x)`	`<cmath>`	Returns the non-negative square root of `x` ($\sqrt{x}$) for `x ≥ 0.0`: if `x` is `2.25`, `sqrt(x)` is `1.5`	`double`	`double`
`tan(x)`	`<cmath>`	Returns the tangent of angle `x`: if `x` is `0.0`, `tan(x)` is `0.0`	`double` (radians)	`double`

Example

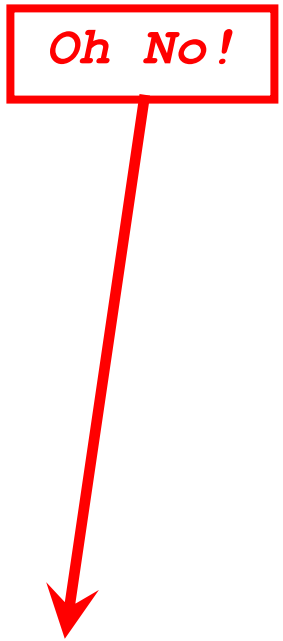
```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
  cout << "Enter Quadratic coefficients: ";
  double a, b, c;
  cin >> a >> b >> c;
  if ( (a != 0) && (b*b - 4*a*c > 0) ) {
     double radical = sqrt(b*b - 4*a*c);
     double root1 = (-b + radical) / (2*a);
     double root2 = (-b - radical) / (2*a);
     cout << "Roots: " << root1 << " " << root2;
  }
  else {
     cout << "Does not have two real roots";
  }
  return 0;
}
```

```
/*
This program computes the volume (in liters) of a six-pack of
   soda cans and the total volume of a six-pack and a two-
   liter bottle.
*/

int main()
{
   int cans_per_pack = 6;
   const double CAN_VOLUME = 0.355; // Liters in a can
   double total_volume = cans_per_pack * CAN_VOLUME;

   cout << "A six-pack of 12 cans contains "
      << total_volume << " liters." << endl;

   const double BOTTLE_VOLUME = 2; // Two-liter bottle

   total_volume = total_volume + BOTTLE_VOLUME;

   cout << "A six-pack and a two-liter bottle contain "
      << total_volume << " liters." << endl;

   return 0;
}
```

Common Error – Omitting Semicolons

Common error

Omitting a semicolon (or two)




```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello, World!" << endl
8     return 0;
9 }
```

Suppose you (accidentally of course) wrote:

```
cot << "Hello World!" << endl;
```

- This will cause a compile-time error and the compiler will complain that it has no clue what you mean by cot. The exact wording of the error message is dependent on the compiler, but it might be something like
  “Undefined symbol cot”.

Consider this:

```
cout << "Hollo, World!" << endl;
```

- *Logic errors* or *run-time errors* are errors in a program that compiles (the syntax is correct), but executes without performing the intended action.

not really an error?

Common Error – Using Undefined Variables

You must define a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_factor;
double liter_factor = 0.0296;
```

Statements are compiled in top to bottom order.

When the compiler reaches the first statement, it does not know that **`liter_factor`** will be defined in the next line, and it reports an error.

Common Error – Using Uninitialized Variables

Initializing a variable is not required, but there is always a value in every variable, even uninitialized ones.

Some value will be there, the flotsam left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

```
int bottles; // Forgot to initialize
int bottle_volume = bottles * 2;// Result is unpredictable
```

What value would be output from the following statement?

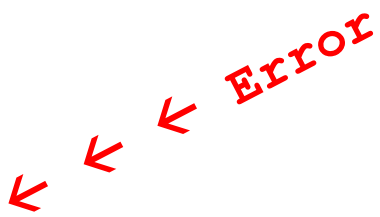
```
cout << bottle_volume << endl; // Unpredictable
```

Common Error – Unintended Integer Division

It is unfortunate that C++ uses the same symbol: **/**
for both integer and floating-point division.
These are really quite different operations.

It is a common error to use integer division by accident.
Consider this segment that computes the average of three integers:

```
cout << "Please enter your last three test scores: ";
int s1;
int s2;
int s3;
cin >> s1 >> s2 >> s3;
double average = (s1 + s2 + s3) / 3;
cout << "Your average score is " << average << endl;
```



What could be wrong with that?

Of course, the average of **`s1`**, **`s2`**, and **`s3`** is

```
(s1+ s2+ s3) / 3
```

Here, however, the **`/`** does not mean division in the mathematical sense.

It denotes integer division because
both **`(s1 + s2 + s3)`** and **`3`** are integers.

For example, if the scores add up to 14,
the average is computed to be 4.

WHAT?

Yes, the result of the integer division of 14 by 3 is 4
How many times does 3 evenly divide into 14?
Right!

That integer 4 is then moved into the floating-point variable **`average`**.

So 4.0 is stored.

That's not what we want!

The remedy is to make the numerator or denominator into a floating-point number:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

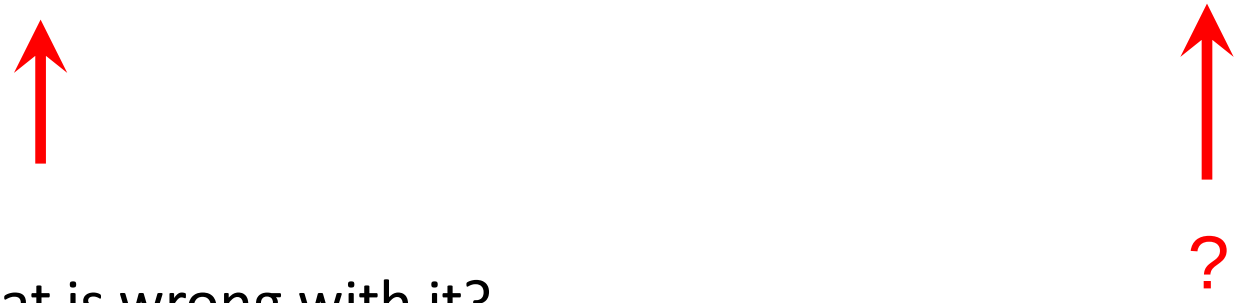
or

```
double average = (s1 + s2 + s3) / 3.0;
```

Common Error – Unbalanced Parentheses

Consider the expression

```
(-(b * b - 4 * a * c) / (2 * a)
```



What is wrong with it?

The parentheses are *unbalanced*.
This is very common with complicated expressions.

Common Error – Roundoff Errors

This program produces the wrong output:

```
#include <iostream>
using namespace std;
int main()
{
   double price = 4.35;
   int cents = 100 * price;
          // Should be 100 * 4.35 = 435
   cout << cents << endl;
          // Prints 434!
   return 0;
}
```

Why?

Common Error – Roundoff Errors

- In the computer, numbers are represented in the binary number system, not in decimal.
- In the binary system, there is no exact representation for 4.35, just as there is no exact representation for ⅓ in the decimal system.
- The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.
- The remedy is to add 0.5 in order to round to the nearest integer:

```
int cents = 100 * price + 0.5;
```