CSIS/COMP 1117B
Computer Programming

C++ Basics

C++ Basics

- Introduction
- Variables and assignments
- Simple flow of control
- Output and input
- Errors
- Functions

Introduction

- C++ is an object-oriented programming language developed based on C by Bjarne Stroustrup in the early 80s.
- Main characteristics of an OOPL:
  - program is structured as a collection of interacting objects
  - encapsulation
  - inheritance
  - polymorphism
- Main features of C++:
  - portable
  - efficient
  - **but *insecure***

Hello World to C++

```
//  hello.cpp
//  This program displays Hello World! on the screen
#include <iostream>
using namespace std;
int main()
{
      cout << “Hello World!” << endl;
      return 0;      // terminate program
}
```

Anatomy of Hello World

- hello.cpp : name of file containing the source program; the suffice indicates that the file contains an C++ source program.
- The first two lines are **comments** which are used to annotate programs for the benefits of their readers.
- Groups of one or more characters that have specific assigned meanings are called **tokens**, e.g., //, using, namespace, <<, =, cin

Tokens (1)

- **Identifiers** (**Names**): iostream, std, main, cout, endl, …
- **Keywords** (**Reserved names**): using, namespace, int, return, …
- **Symbols**: ;, <<, …
- **Delimiters**:
  - pairs of parentheses ()
  - braces {}
  - angle brackets <>
  - square brackets []
  - single quotes ‘’
  - double quotes “”

Tokens (2)

- **Constants**:
  - 0 (numeric constant)
  - “Hello World!” (string constant)
- **Comments**:
  - everything starting from (and including) // to (and including) the first *end-of-line* (*newline* character) are ignored by the compiler; can be put at the end of any lines of code or by itself.

Variables and Assignments

- Introduction
- Values and constants
- Variables and variable declarations
- Assignments and expressions

Identifiers

- A C++ program may contain many identifiers.
- A ***valid*** identifier begins with a letter, followed by any number of letters or digits.
  - **letters**: A, B, …, Z, a, b, …, z, _
  - **digits**: 0, 1, …, 9
- Case sensitive: “name” is different from “Name”
- ***Avoid using keywords as identifiers***
- Valid names: A, a2c, HKU_CS, _2, _B, _ _A
- Invalid names: 2c, A B, A%B, -D, $A, new

Keywords

asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
extern
false
float
for
friend
goto
if
inline
int
log
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_case
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while

Data Types

- A program processes data, every datum has a **type** attribute.
- Primitive types in C++: char, bool, int, long, float, double
- Every value belongs to a type, e.g., 5 is an int value
- In the version of C++ we use,
  - an int value occupies 4 bytes
  - a double value occupies 8 bytes
  - a bool value of **true** or **false** occupies 1 byte
    - a zero value is interpreted as false and ***any*** nonzero value is interpreted as true
  - a char value occupies 1 byte
- *Can we write a program to find out the size of the various data types?*

Constants

- A **constant** is an item representing a ***fixed*** value in a program.
- For example:
  - integer (int) constants: -35, 112, 0
  - real (double) constants: 0.0, 3.1416, -1e10
  - character (char) constants: ‘A’, ‘e’
  - Boolean (bool) constants: true, false
  - string constants: “I like C++”, “Hong Kong U”
    - strings are ***null-terminated***, an extra (binary) zero character is added at the end of a string

Special Characters

- Special characters are used to represent non-printable characters in a program, they all start with a \ (backslash)
  - \a alert (a beep sound)
  - \b backspace
  - \n newline
  - \t tab
  - \\ backslash itself
  - \’ single quote
  - \” double quote
  - \xdd the numeric value of one byte written as two hexadecimal digits, e.g., ‘\x41’ represents ‘A’

Variables

- A **variable** is a sequence of memory cells for storing a value such as: ‘c’, 24, 3.0
- Imagine that a variable being a ***box*** for storing a constant value. You can store a value into the box. Later, you can read this value from the box or store another value into the box.
- Each variable is *typically* identified by a **name**, e.g., C, two_4, three_point_zero
  - *a variable could have more than one name at the same time and at different times of program execution*

Variable Declarations (1)

- A variable is introduced into a program by way of a **variable declaration**.
- A variable declaration associates a name with a variable together with the type of the value that can be stored into that variable.
- A (simple) variable declaration takes the following form:

  <type> <identifier> ';'
- For example:

```
int c;          // declares an integer variable with name c
char two_4;     // declares a character variable with name two_4
```

- ***There is no relationship between the name of a variable and the type of values stored in it!***

Variable Declarations (2)

- All variables must be declared before they can be used in a program.
- The type of the value that can be stored into a variable is called the **type of the variable**.
- **Strict type compatibility**: only a value of the type specified in the declaration of a variable is allowed to be put into the variable
  - there are relaxations of this rule for convenience of programming:
    - values of a different type can be stored *unmodified* into a variable of a different type if the two different types of values are of the same size
    - numeric values of equal or different sizes *always* require *conversion*, e.g., int to float or vice versa

Constant Declarations

- Variable declarations can also be used to declare constants, a more complete variable declaration:

  [const] <type> <identifier> [‘=‘ <expression>] ‘;’
- The modifier const indicates that the value of this variable cannot be changed, effectively making the variable a constant (a **symbolic constant**).
- The expression after = should evaluate to a constant value of the appropriate type at compile time.
- For example:

  const double PI = 3.141592653589793;

  const int MAX_ENROLLMENT = 100;

Symbolic Constants

- Constants are often given names in a program to facilitate program maintenance -- changing the value of a constant requires changing only its declaration, no need to search the entire program for all occurrences of that particular constant
- It is good programming practice to define constants as symbolic constants with **meaningful names**, for example, PI for the value of π

Naming Conventions

- A variable or function name consists of lowercase letters, e.g., count, fahr, c2f, f2c
- The name of a symbolic constant consists of only capital letters, e.g., PI, LIGHT_VELOCITY
- If a name consists of multiple words, they are joined together with underscores, e.g., hello_world, left_button_clicked, hong_kong_university
- All names shall be ***meaningful***.
- Major identifiers, variable and constants, usually have inline comments explaining their intended usage.

A Simple C++ Program Skeleton

- Statements in a program are executed sequentially and there are many different kinds of statements in C++

```
#include <iostream>
using namespace std;
int main() {
    statement 1;
    statement 2;
    . . .
    statement n;
    return 0;
}
```

Assignment Statement (1)

- An **assignment statement** is used to update the value stored in a variable, it takes the following form:

  <identifier> ‘=‘ <expression> ‘;’
- The effect of this statement is that the expression on the right-hand-side is evaluated and the value is stored into the variable on the right-hand-side.
- For example:

  count = 5; // store 5 into the variable count
- The type of the value obtained by evaluating the expression (called the **type of the expression**) must *agree* (*compatible*) with the type of the variable.

Assignment Statement (2)

- When a variable is reassigned later, the previous value in the cells is overwritten, e.g.,

```
count = 5;
    . . .
count = 0;  // the new value in count is 0
```

- When the expression is a single variable , it represents the value stored in that variable, for example:

```
k = count;  // copy the value (stored) in count
                // and store it into the variable k
```

Operators and Expressions

- An **operator** specifies an operation for the CPU to carry out.
- For example, to compute the sum of the values stored in the variables x and y:

  x + y
- In this example, x and y are the **operands**
- An **expression** is a series of operations specified using constants, variables and operators.
- For example:

  7, x, x + y, count + 1, a - b - c, a * (b - c)
- The **value of an expression** is the value obtained after evaluating the expression.

Operator Precedence

- The (partial) syntax for (numeric) expressions:

  <expression> → <expression > (‘+’ | ‘–’) <term>

  <term> → <term> (‘*’ | ‘/’) <factor>

  <factor> → <constant> | <identifier> | ‘(‘ <expression> ‘)’
- The precedence rules of arithmetic operators are *usually* observed in (most) programming languages.
- Operators typically impose restrictions on the types of its operands. Note that the type of the value delivered by an operator (**the type of an operator**) may not be the same as the types of it operands (e.g., relational operators take numeric operands and return bool results).

Operator Overloading

- **Operator overloading** means that the same symbol is used to represent more than one operation. For example, + is used to mean both integer addition and floating-point addition.
- When an integer and a floating-point number are supplied as operands to an overloaded operator, the integer will be converted into a floating-point value which will be used with the other floating-point value to complete the operation. This implicit conversion is called **coercion**.
- For example:

```
2 + 3.0  // evaluates to 5.0
7 / 2    // evaluates to 3
7.0 / 2  // evaluates to 3.5
```

Type Conversion (Casting)

- Recall that assignment requires that both sides of the assignment agree in type; however . . .
- If the type of the expression is different from that of the variable, a type conversion (**casting**) is required, for example:

```
int x; double y;
    y = 7/3;            // int result (3) coerced to double
    x = 7.0/3;          // warning!!
    x = (int)(7/3.0);  // double -> int; correct
```

- In general, an expression can be prefixed by a **casting** to make sure that the value delivered is of the specified type. As described previously, actual data conversion may or may not be necessary.

Simple Flow of Control

- Selection
  - simple if statement
  - if statement
- Repetition
  - simple while loops

Simple if Statement

- In daily life, we often do (or do not do) something, depending on a condition.
- For example:
  - if he/she knows C++, date him/her
  - if it rains, bring an umbrella
  - if typhoon signal No. 8 is hoisted
    - if at home, stay home
    - if at school, go home
- We can attach a condition to a statement. If the condition is true, execute the statement; otherwise, skip to the next statement.

An Example

- For example, if the total hours worked for the week exceeds 40, the exceeded time will earn a higher salary:

```
if (work_hour > 40) {
    overtime_hour = work_hour – 40;
    overtime_pay = overtime_hour * OT_PAY_RATE;
}
```

- The assignment statements (then-part) will only be executed if work_hour > 40
- Note that 40 should be a defined constant
  - what if the number of basic work hours per week got reduced to 36?

What You should Try to Avoid!

- The previous example can be rewritten as:

  If ((overtime_hour = work_hour – 40) > 0)

  overtime_pay = overtime_hour * OT_PAY_RATE;

- The following is legal C, but is it correct?

  If (overtime_hour = work_hour – 40 > 0)

  overtime_pay = overtime_hour * OT_PAY_RATE;

  – *What does this program actually computes?*

if Statement

- The syntax of an if statement is:

  if ‘(‘ <expression> ‘)’ <statement>

  [else <statement>]
- The first <statement> is called the **then part** and the second <statement> is optional and is called the **else part**.
- The then part is executed if <expression> evaluates to true; otherwise, the else part, if there is one, will be executed. Execution will then proceed to the next statement.

Another Example

- Suppose we want to compute the sign of a number

  if (n > 0) sign = 1;

  else if (n < 0) sign = -1;

  else sign = 0;

- Does the program work for both integers and floats?
- *Any more tricks?*

Simple while Loops

- In daily life, we often do something ***repeatedly***. For example, we brush our teeth everyday. We construct a building floor by floor in more or less the same way.
- Moreover, we do something repeatedly as long as a certain ***condition*** is satisfied. For example, we breathe ***while*** we are alive. We construct a building floor by floor ***while*** the roof is not yet reached.
- In computation, we frequently execute a set of statements repeatedly while a condition (***while-condition***) is satisfied.

while Statement

- The syntax of a while statement:

  while ‘(‘ <expression> ‘)’ <statement>
- <expression> is called the **while condition** and <statement> is called the **while body**.
- The while condition is evaluated and if the result is true, the while body will be executed and then the whole process is repeated.
- In general, the while body should contain operations that will make the while condition ***false*** so that execution can break out of the loop and continue with the following statement.

An Example

- Suppose we want to find the greatest common divisor of two positive integers using the Euclidean algorithm:
  - if they are equal, we are done; otherwise, subtract the smaller from the larger and repeat:

  while (m != n)

  if (m > n) m = m – n; else n = n – m;
  - upon exit from the loop, either m or n is the GCD
  - *will the loop ever terminate?*

Wait a Minute

- Note that in the previous example, the original pair of numbers will be *destroyed* in most cases!
- We should work on copies of the two numbers if their values need to be preserved. The program should be more like:

```
gcd = number_1;   // loop initialization
n = number_2;     // loop initialization
while (gcd != n)  // check for continuation
   if (gcd > n) gcd = gcd – n; else n = n – gcd;
```

Output and Input

- Simple output using cout
- Simple input using cin
- Simple input validation

Simple Screen Output

- cout is a ***predefined object*** representing the standard output device. In a PC, it connects to the *console window* on the screen.
- cout is defined in the library iostream. We need to include this library in the beginning of a program before referring to cout.

  ```
  #include <iostream>
  ```

- To output a string, say, “I like C++.”, to cout, write

  ```
  cout  <<  “I like C++.” ;
  ```

- The << is the **insertion operator** for *sending* values to be output to cout.

Several Items in One Line

- Textual strings are commonly used to label output. For example, in the temperature conversion program:

  cout << "The temperature in Centigrade is: " << c << '\n';

- Here, three items are sent to cout in succession, the overall effect is that the temperature in Centigrade is output with the proper label followed by a line-break.
- Note that '\n' can be replaced by endl:

  cout << "The temperature in Centigrade is: " << c << endl;

Simple Formatting

- Output can be positioned at specific columns using the tab (\t) character. For example, the following will generate a 3-column table:

```
while ( . . . ) {
    . . .
    cout << a << '\t' << b << '\t' << c << endl;
}
```

- The formatting capabilities of C/C++ are very limited and text formatting is a very complicated problem in general.
- *Need any hint on outputting "?*

Output Manipulators

- endl is a **manipulator**. It is placed after an insertion operator and it causes the insertion of a new-line character.
- setw(<int value>) is another manipulator that can be used to set the width of the ***next*** output value. It is defined in iomanip
- An effect similar to tabbing can be achieved by:

```
#include <iomanip>
. . .
cout << setw(15) << a << setw(15) << b
        << setw(15) << c << endl;
```

Simple Keyboard Input

- **cin** is a ***predefined object*** representing the standard input device. In a PC, it connects to the keyboard.
- cin is defined in the library iostream. We need to include this library before referring to cin

  #include <iostream>
- To input a value to the int variable age, write

  cin >> age;
- The >> is the **extraction operator**
- When this statement is executed, the execution of the program is suspended. The computer waits for the user to type in an int value, e.g., 25. Execution resumes when the <Enter> key is pressed. The computer then extracts an int value and assigns it to age. ***Leading*** blanks, tabs and newline characters are skipped.

Multiple Inputs

- cin can be used to input a sequence of values in a manner similar to cout. For example, to input two ints followed by a float, write

```
int  a, b;
float   x;
. . .
cin >> a >> b >> x;
```

- cin will try to consume as many characters as necessary to obtain values for the three variables (note that leading and intervening blanks, tabs, and newlines are ignored).

Labeling Inputs

- Its is customary to ***prompt*** the user for a particular input value. For example, in the temperature conversion program, the input is *prompted* by a string:

```
cout << "Enter the temperature in Fahrenheit: ";
cin >> f;     // accepts input from user and place in f
```

- It is common to also include some indications of valid inputs to the user in the prompt:

```
cout << “Enter a month in a year (1 – 12)”
cin >> m;
```

Simple Input Validation

- cin >> f is an expression as well as an input statement; it returns true if the input operation succeeds, otherwise false.

```
cout << "Enter the temperature in Fahrenheit: ";
if (cin >> f) {
    c = (f - 32) * 5 / 9;
    cout << "The temperature in Centigrade is: ”
            << c << ‘\n’;
}
else cout <<"Invalid input!\n";
```

Errors

- To understand a C++ program, imagine that you are a computer and try to execute the program
  - for a very simple program, use your mind to keep track of the variables' values
  - for a simple program, use pencil and paper
  - for a big program, use a **debugger**, a software that allows us to *step through* the execution and *show* the values of variables
- A mistake in a program is called a **bug**. The process of eliminating bugs is called **debugging**.

Types of Errors

- Syntax errors
- Runtime errors
- Logical errors

Syntax Errors

- **Syntax errors** occur when a program contains components whose forms violate the syntax rules. For example, invalid names, misspelling of keywords, …
- Syntax errors can be detected during compilation. They are also called **compile-time errors**.
- Common syntax errors:
  - incorrect integer constants, e.g., 4,000
  - putting some binary operators side by side, e.g., a + / b
  - mismatch parentheses, e.g., a + (1 + b))
  - missing ‘;’ in the end of an assignment statement
  - mistyping ‘1’ (digit one) for ‘l’ (lowercase L)

Runtime Errors (1)

- **Runtime errors** occur when an illegal operation is specified by the executing program.
- Examples of illegal operations:
  - **division-by-zero** error occurs when i equals 0 in 3 / i
  - **invalid-input** error occurs when a program expects to read an integer value, e.g., cin >> j; but instead a non-digit character is found
  - **integer-overflow** error occurs when the result of an integer operation exceeds the possible range of integer values, e.g., a = a + 1 when a already contains the largest possible value

Runtime Errors (2)

- Runtime errors cannot be detected at compile-time. When it occurs, the execution may be terminated abnormally. An error message may or may not be issued (by the system).
- The worst possible case is that the program continues execution without any indication
  - the program executes fine and give correct answers most of the time
    - the runtime time error did not occur for this particular execution of the program
  - but occasionally generates wrong answers!
    - *how do we know that the answer is wrong?*
      - are we going to check all the output every time we run the program?

Logical Errors

- **Logic errors** (**design errors**) occur when a program is compiled alright (no syntax error), runs properly (no runtime error), but produces unexpectedly incorrect answers because of incorrect logic in the program. The program does not do what it is designed for.
- For example, errors due to
  - incorrect algorithms
  - missing parentheses, e.g., a * (b + c)
  - incorrect formulas, e.g., area = PI * radius * 2
  - incorrect condition in an if or while statement
- ***All nontrivial programs contain bugs!***

Functions

- A **function** is a small program that is invoked (called) to accomplish a task. For example:

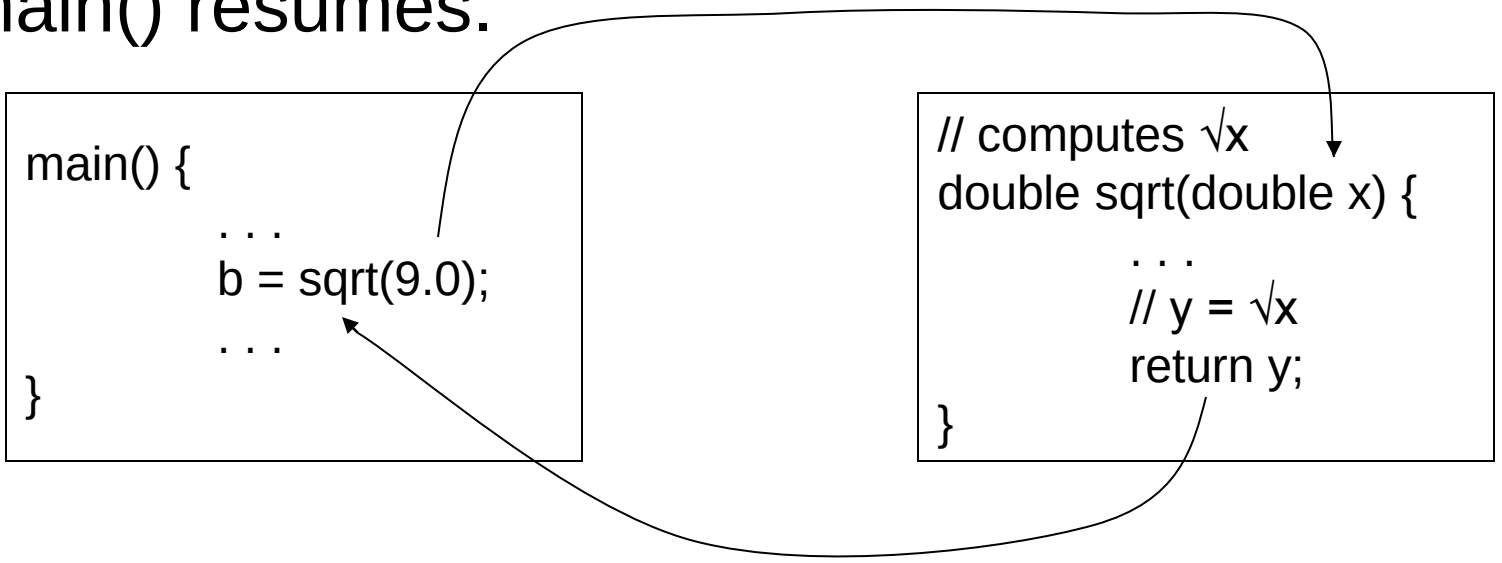
  b = sqrt(9.0) ;

  – sqrt is the name of a function; when the above statement is executed, 9.0 is sent to sqrt() as input

  – sqrt() computes the square root of its input and returns the result, 3.0; this value is then assigned to the variable b

Function Invocation

- The invocation of a function is similar to pressing a function key on your calculator. When the statement sqrt(9.0) is evaluated, the execution of main() is suspended and the execution of sqrt(x) starts. When sqrt(x) terminated, the execution of main() resumes.



Function Call as An Expression

- A function call is classified as an expression and has the following syntax:

  <identifier> ‘(‘ [<arguments>] ‘)’
- <identifier> is the name of the function being called and the optional argument list is a comma (,) separated list of expressions.
- Examples of function calls:

  sqrt(x*x + y*y)

  gcd(m, n)
- **Arguments** are used to provide input to a function. An argument is an expression. It is evaluated when the function is invoked. The value is then passed to the function.

Library Programs

- A **library** is a collection of functions. Some common C++ libraries are **cmath, cstdlib, ctime, iostream**, … The **#include** statement in the beginning of a program specifies the libraries that are needed in the program.
- For example:

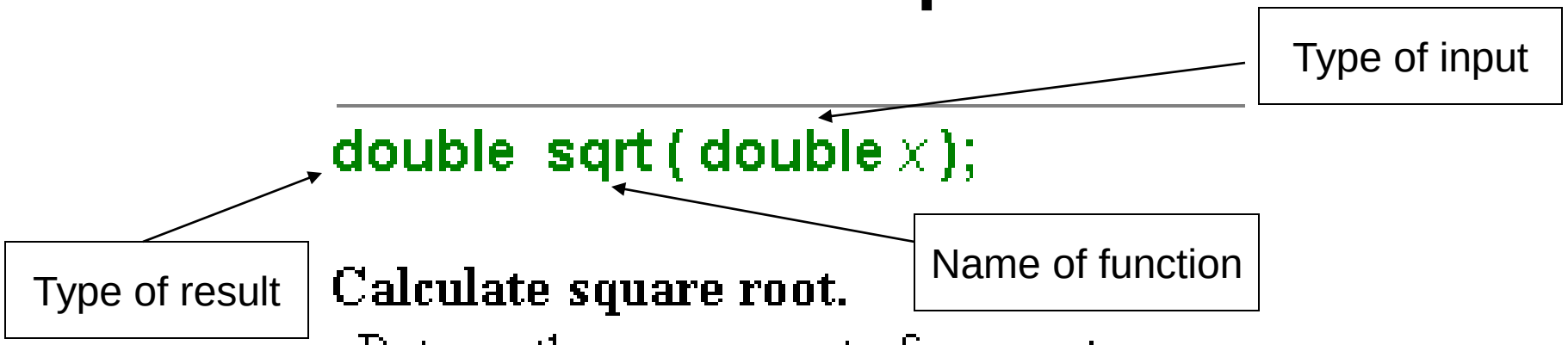
```
#include <iostream>
#include <cstdlib>
#include <cmath>
```

- The specification of the standard libraries can be found in *http://www.cplusplus.com/ref/*

Application Program Interface

- An **Application Program Interface** (**API**) explains how to use a function. It includes
  - the function name
  - what the function computes
  - input: the order and the types of parameters
  - output: the type of the return value
  - examples or other information may also be included

API of sqrt



Calculate square root.

Returns the square root of parameter *x*.

Parameters.

x

Non-negative floating point value.

Return Value.

Square root of *x*.

Another Example

---

double pow (double *x*, double *y*);

Calculate numeric power.

Returns *x* raised to the power of *y*. x^y

Parameters.

x

Base value.

y

Exponent value.

Return Value.

x raised to the power of *y*.

Time to Write A Function

- It is easy to package our simple program for computing GCD into a function:

```
int gcd (int m, int n) {
        while (m != n)
                if ( m > n) m = m – n; else n = n – m;
        return n;
 }
```

  - gcd takes 2 ints and returns an int
  - it computes the GCD of its input
  - for example, gcd(100, 35) returns 5

Function Declaration (1)

- Declaring a function is similar to declaring a variable in that we are defining a certain name to be used later in the program, except that this time we need to supply a lot more information:
  - the **function heading**
    - name of the function
    - the type of the return value (output)
    - the types of the input (if there is any)
  - the **function body**, the program that performs the operations

Function Declaration (2)

- The syntax of a function declaration:

  <funct_decl> → <head> <body>

  <head> → [<type>] <name> ‘(‘ [<formal_list>] ‘)’

  <formal_list> → <formal> { ‘,’ <formal> }

  <formal> → <type> <identifier>

  <type> → <identifier>

  <name> → <identifier>

  <body> → <compound_statement>

- **void** can be specified as the return type if the function does not return a value.
- ***Why is return type optional?***

More about Return

- If a function does not return a value, its execution will end after the last statement is executed; no return statement is needed.
- Functions that do not return any value is defined with a return type of **void**.
- In case we want to terminate the execution of the function before the last statement is executed, write a bare return statement:

  return;

- The syntax of the return statement is:

  return [<expression>] ‘;’

One More Example

- A **compound statement** is a sequence of one or more statement that can appear anywhere a statement can:

  <compound_stmt> → ‘{‘ { <statement> } ‘}’

- sign can be packaged into a function as:

```
int sign(int n) {
    if (n > 0) return 1;
    else if (n < 0) return -1;
    else return 0;
}
```