More Flow Control Functions in C++

CS 16: Solving Problems with Computers I
Lecture #4

Ziad Matni
Dept. of Computer Science, UCSB

CS16 Registration

REGISTRATION IS CLOSED FOR THIS CLASS

No more adds ☹

MIDTERM #1 IS COMING! October 19th!

- Material: ***Everything*** we've done**, incl. up to Tue. 10/17**
  - Homework, Labs, Lectures, Textbook
- **Thursday, 10/19** in this classroom
- **Starts at 2:00pm **SHARP** (come early)**
- **BRING YOUR STUDENT IDs WITH YOU!!!**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (single-sided) of written notes
  - Must be no bigger than 8.5" x 11"
  - **You have to turn it in with the exam**
- **You will write your answers on the exam sheet itself.**

Lecture Outline

- Multiway Branching and the `switch` command
- Local vs. Global Variables

- Pre-Defined Functions
- User-Defined Functions
- Void Functions

Compile vs. Run Time Errors

Compile Time Errors

- Errors that occur *during* ***compilation*** *of a program*.

Run Time Errors

- Errors that occur *during the* ***execution*** of a program
- Runtime errors indicate bugs in the program (bad design) or unanticipated problems (like running out of memory)
- Examples:
  - Dividing by zero
  - Bad memory calls in the program (bad memory address)
  - Segmentation errors (memory over-flow)

Short-Circuit Evaluation

- Avoid possible *run time errors* by using the right Boolean expressions
- If you strategically use the **&&** operator, then some Boolean expressions do not need to be completely evaluated
  - Especially if they can potentially cause run time errors
  - This is known as “short-circuit evaluation”
- Consider this if-statement:

```
if (pieces / kids >= 2) … etc… ← what’s a potential problem?
```

ANS: potential divide-by-0

FIX:

```
if ( (kids != 0) && (pieces / kids >= 2) ) … etc…
```

Multiway Branching

- Nesting (embedding) one if/else statement in another.

```
if (count < 10) {
    if ( x < y )
        cout << x << " is less than " << y;
    else
        cout << y << " is less than " << x;
}
```

- Note the tab indentation at each level of nesting.

Defaults in Nested IF/ELSE Statements

- When the conditions tested in an if-else-statement are mutually exclusive, the final if-else can sometimes be omitted

EXAMPLE:

```
if (guess > number)
    cout << “Too high.”;
else if (guess < number)
    cout << “Too low.”;
else if (guess == number)
    cout << “Correct!”;
```

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.”;
else cout << "Correct!";
```

i.e. All other possibilities

A Better Way… Using **switch**

An alternative for constructing multi-way branches

```
switch (variable)
{
 case variable_value1:
     statements;
     break;

 case variable_value2:
     statements;
     break;

 …   …   …

 default:
     statements;
 }
```

Controlling statement

"break" statement is important – you cannot forget it!

When you see this, it means I'm demonstrating code in class AND will have it available on the class website!

Demo!

The Controlling Statement

- A `switch` statement's controlling statement must return one of these basic types:
  - A **bool** value
  - An **int** type
  - A **char** type

- `switch` will <u>not</u> work with **strings** in the controlling statement.

Can I Use the **break** Statement in a Loop?

- Yes, technically, the **`break`** statement can be used to exit a loop (i.e. force it to) before normal termination

- **But it’s not good design practice!**
  - In this class, **do <u>NOT</u> use it outside of `switch`**

Note About Blocks

- ***Recall***: A block is a section of code enclosed by **{…}** braces

- Variables declared within a block, are **local to the block**
  - An exclusivity feature
  - These variable are said to have the block as their ***scope***.
  - They can used inside this block *<u>and nowhere else!</u>*

- Variable names declared inside the block **cannot** be re-used outside the block

Local vs. Global Variables

- **Local variables** only work in a specified block of statements
  - If you try and use them outside this block, they won’t work

- **Global variables** work in the entire program

- There are standards to each of their use
  - Local variables are **much preferred** as global variables can cause conflicts in the program

Local vs. Global Variables – Example

Demo!

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

int main( )
{
   int age(0);
   for (int c = 0; c < 10; c++)
   {
      cout << age*c << endl;
      age += (2*c + 4);
   }
   return 0;
}
```

Local to main( )

Local to the for-loop

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

int age(0);
int main( )
{
   for (int c = 0; c < 10; c++)
   {
      cout << age*c << endl;
      age += (2*c + 4);
   }
   return 0;
}
```

Globally declared

```
#include <iostream>
using namespace std;
int main( )
{
   int k;
   for (int j = 0; j < 3; j++)
   {
      k = 9;
      cout << “CS ”;
      while (k > 7)
      {
         cout << k;

         k--;
      }
      cout << “.”;
   }
   cout << endl;
   return 0;
}
```

Exercise

Complete the program to the left if you want the outputs to be:

```
CS 98.CS 98.CS 98
```

(there’s a newline character at the end)

Predefined Functions in C++

- C++ comes with “built-in” libraries of predefined functions

- Example: sqrt function (found in the library ***cmath***)
  - Computes and returns the square root of a number

```
the_root = sqrt(9.0);
```

  - The number 9 is called *the argument*

- Can variable **the_root** be either int or double?

Notes on the **cmath** Library

- Standard math library in C++
- Contains several useful math functions, like

`cos( ), sin( ), exp( ), log( ), pow( ), sqrt( )`

- To use it, you must import it at the start of your program

#include <cmath>

  – You can find more information on this library at: http://www.cplusplus.com/reference/cmath/

Other Predefined **cmath** Functions

- pow(x, y) --- `double value = pow(2, -8);`
  - Returns 2^{-8} , a double value (value = 0.00390625)
  - Arguments are of type double

- sin(x), cos(x), tan(x), etc… --- `double value = sin(1.5708);`
  - Returns sin($\pi/2$) (value = 1) – note it's in radians
  - Argument is of type double

Other Predefined **cmath** Functions

- abs(x) --- `int value = abs(-8);`
  - Returns absolute value of argument x
  - Return value is of type **int**
  - Argument is of type int

- fabs(x) --- `double value = fabs(-8.0);`
  - Also returns absolute value of argument x
  - Return value is of type **double**
  - Argument is of type double

Random Number Generation: Step 1

- Not true-random, but pseudo-random numbers.

Must

```
#include <cstdlib>
#include <ctime>
```

- First, ***seed*** the random number generator (only need to do this once)

```
srand(time(0)); //place inside main( )
```

  - **time( )** is a pre-defined function in the **ctime** library: gives current system time (it gives the current system time)
  - It's used here because it generates a *distinctive enough seed*, so that **rand( )** generates a "good enough" random number.

Random Number Generation: Step 2

- Next, use the **rand( )** function, which returns a random integer that is greater than or equal to 0 and less than RAND_MAX
(a library-dependent value, but is at least 32767)

```
int r = rand();
```

- But what if you want to generate random numbers in other ranges? Example, between 1 and 6?

Random Numbers

Demo!

- Use **%** and **+** to scale to the number range you want
- For example to get a random number bounded from 1 to 6 to simulate rolling a six-sided die:

```
int die = (rand( ) % 6) + 1;
```

Type Casting

- Recall the problem with integer division in C++:

```
int total_candy = 9, number_of_people = 4;
double candy_per_person = total_candy / number_of_people;
```

  – candy_per_person will be **2**, *not* 2.25!

- A **Type Cast** produces a value of one data type from another
  - **`static_cast<double>(total_candy)`**
    produces a *double* var representing the integer value of **total_candy**

Type Cast Example

```
int total_candy = 9, number_of_people = 4;
double candy_per_person =
 static_cast<double>(total_candy)/number_of_people;
```

– The numerator of this division is now 9.0

– So, candy_per_person is now 2.25

– The following would also work:

```
candy_per_person =    total_candy / static_cast<double>(number_of_people);
```

– This, however, would not! (why?)

```
candy_per_person = static_cast<double>(total_candy / number_of_people);
```

ANS: Because, in this example, integer division occurs ***before*** type cast!

Question

Demo!

- Can you determine the value of d?

```
int a(11), b(2);
double d = a / b;
```

- And now? Can you determine the value of d?

```
double d = 11 / 2;
```

Integer division occurs before type cast!

- What about this value of d?

```
double d = 11.0 / 2.0;
```

FUNCTIONS in C++

Programmer-Defined Functions

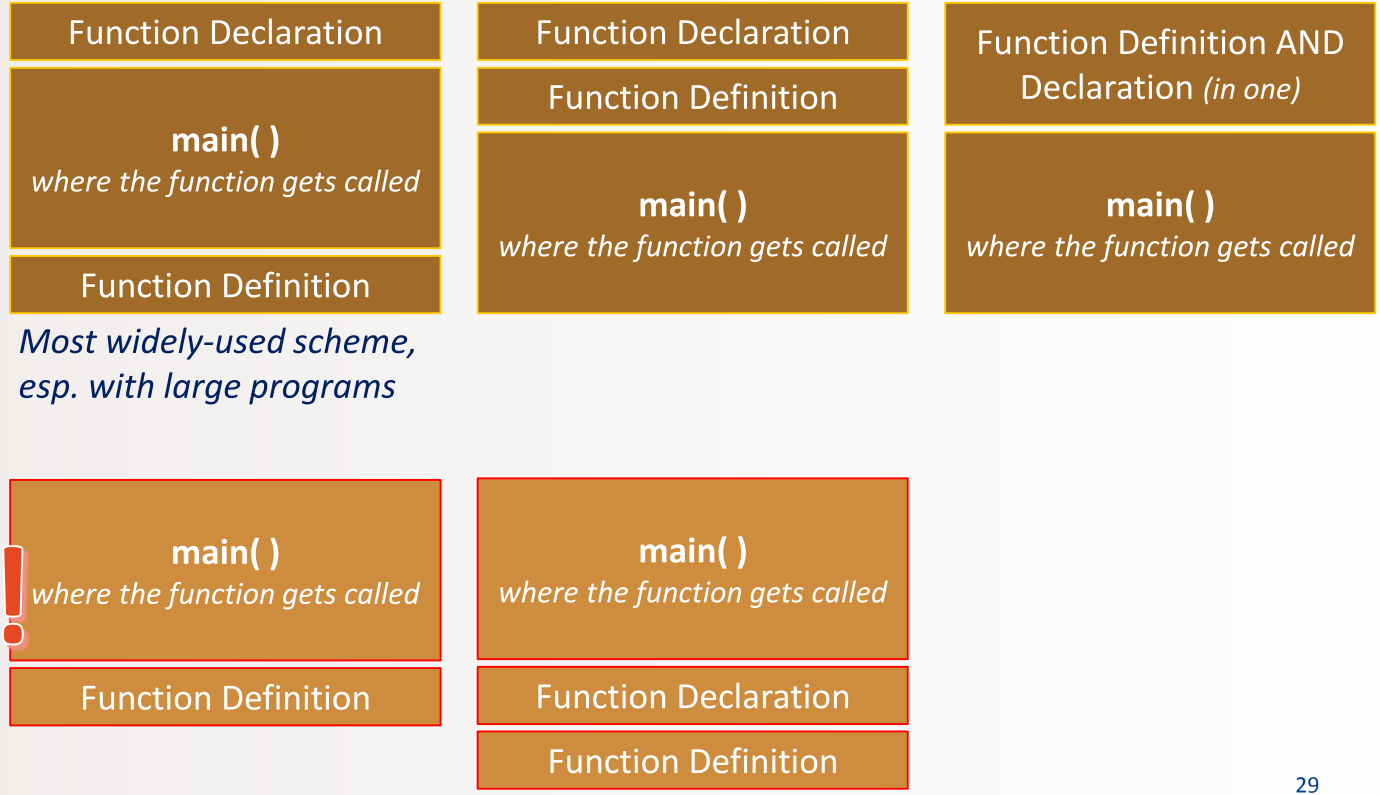
- There are 2 necessary components for using functions in C++

- **Function declaration** (or function prototype)
  - Just like declaring variables
  - Must be placed *outside* the **main( )**, *usually* just before it
  - Must be placed *before* the function is ***defined*** & ***called***

- **Function definition**
  - This is where you define the function itself (all the details go here)
  - Must be place *outside* the **main( )**
  - Can be before **main( )** or after it, *often* placed after it

Block Placements for Functions

OK!



Function Declaration

main( )
where the function gets called

Function Definition

Most widely-used scheme, esp. with large programs

Function Declaration

Function Definition

main( )
where the function gets called

Function Definition AND Declaration *(in one)*

main( )
where the function gets called

NOT OK!

main( )
where the function gets called

Function Definition

main( )
where the function gets called

Function Declaration

Function Definition

Function Declaration

- Shows how the function is ***called*** from **main( )** or from other functions
- **Must** appear in the code *before* the function can be called
- Syntax:

```
Type_returned Function_Name(Parameter_List);
//Comment describing what function does
```

Needed for declaration statement

;

E.g:

```
double interestOwed(double principle, double rate);
//Calculates the interest owed on a loan
```

Function Definition

- Describes ***how*** the function does its task
- Can appear before or after the function is called
- Syntax:

```
Type_returned  Function_Name(Parameter_List)
   {
         //code to make the function work
   }
```

Example of a Simple Function in C++

Demo!

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

int sum2nums(int num1, int num2); // returns the sum of 2 numbers

int main ( ) {
    int a(3), b(5);
    int sum = sum2nums(a, b);
    cout << sum << endl;
    return 0;
}

int sum2nums(int num1, int num2) {
    return (num1 + num2);
}
```

Declaration

Call

Definition

YOUR TO-DOs

- □ Finish reading up Chapter 4 and 5
- □ Turn in HW2

- □ Finish Lab2 by FRIDAY AT NOON (Fri, 10/13)
- □ Visit Prof’s and TAs‘ office hours if you need help!

- □ Send your mom a text

</LECTURE>