More Flow Control Functions in C++

CS 16: Solving Problems with Computers I
Lecture #4

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

- **CHANGED T.A. OFFICE/OPEN LAB HOURS!**
  - Thursday, 10 AM – 12 PM Muqsit Nawaz
  - Friday, 11 AM – 1 PM Xiyou Zhou
- Syllabus is updated
- Linux Workshop Next Week!
  - HFH Conference Room (HFH 1132)
  - Friday, April 20th, 1:00 – 2:30 PM

Lecture Outline

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

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

Nested Loops

- The body of a loop may contain any kind of statement,
  including another loop

- When loops are nested, **all iterations of the inner loop**
  are **executed for each iteration of the outer loop**

- *ProTip:* Give serious consideration to making the inner loop a function call to make it easier to read your program
  - More on functions later…

Example of a Nested Loop

- You want to collect the total grades of 100 students in a class
- Each student has multiple scores
  - Example: multiple homeworks, multiple quizzes, etc…
- You go through each student – one at a time – and get their scores
  - You calculate a sub-total grade for each student
- Then after collecting every student score, you calculate a grand total grade of the whole class and a class average (grand total / no. of students)

Example of a Nested Loop

```
int students(100);
double grade(0), subtotal(0), grand_total(0);

for (int count = 0; count < students; count++)
{
    cout << "Starting with student number: " << count << endl;
    cout << "Enter grades. To move to the next student, enter a negative number.\n"
    cin >> grade;
    while (grade >= 0)
    {
        subtotal = subtotal + grade;
        cin >> grade;
    } // end while loop
    cout << "Total grade count for student " << count << "is " << subtotal << endl;
    grand_total = grand_total + subtotal;
    subtotal = 0;
} // end for loop

cout << "Average grades for all students= " << grand_total / students << endl;
```

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!

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 not 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!**
  - Its use is considered "sloppy" and unprofessional
  - In this class, **do NOT 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 *and nowhere else!*

- 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
  - Sometimes we want to define **constants** and use them as globals

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

Global Constants – Example

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

const double PI=3.14159;
int main( )
{
    double angle=0;
    while (angle <= 2*PI)
    {
        cout << "sin(" << angle << ") = ";
        cout << sin(angle);
        angle += PI/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;  //same as "\n"
    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)

FUNCTIONS in C++

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

- 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;
```

Programmer-Defined Functions

- In C++, you can create your own functions
  - You can have them "do things" based on ***input arguments***
  - These functions can also ***return*** a value or ***NOT***

- You have to declare functions as "types"
  - That is, what "type" of data they return (if any)
  - Example (here, **x** and **y** are the *input arguments*):

```
double functionX(int x, int y)
string functionX(int x, int y)
void functionX(int x, int y)
```

returns a double

returns a string

returns nothing

Programmer-Defined Functions

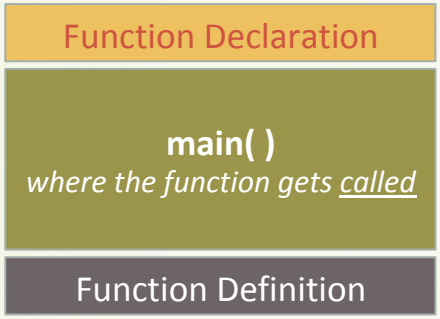
- There are 2 necessary components for using functions in C++

- **Function declaration** (a.k.a 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***

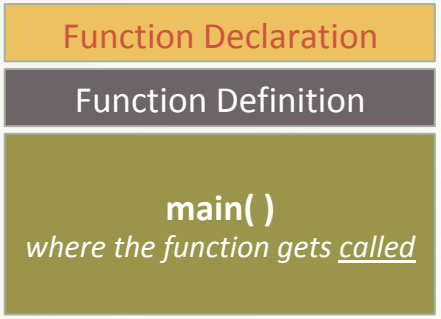
- **Function definition**
  - This is where you define the function itself (all the details go here)
  - Must be place *outside* the **main( )**
  - Can be before **main( )** or after it, *often* placed after it

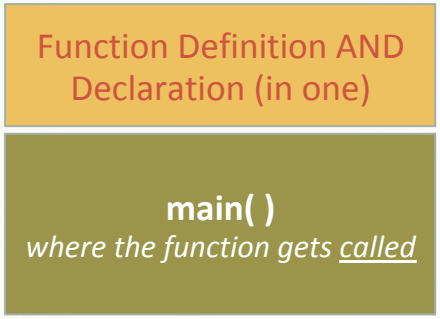
Block Placements for Functions

OK!

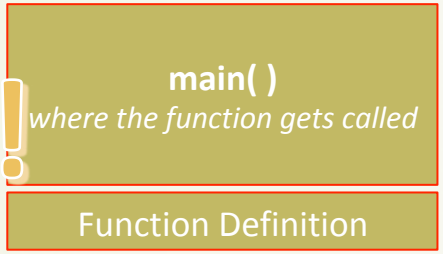


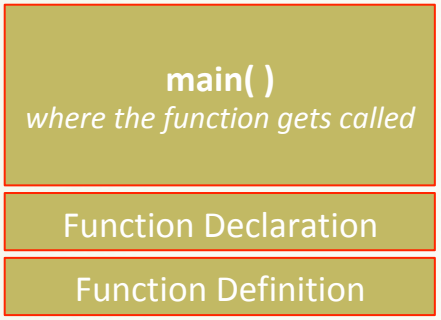
Most widely-used scheme, esp. with large programs





NOT OK!





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++

```
#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

Demo!

void Functions

- Sometimes, we want ***design subtasks*** to be implemented as functions.
  - Repetition involved, like printing some variable over and over again
  - We may not want to return anything

```
1  // void function example
2  #include <iostream>
3  using namespace std;
4
5  void printmessage ()
6  {
7    cout << "I'm a function!";
8  }
9
10 int main ()
11 {
12   printmessage ();
13 }
```

void Function: Simple Example

- Let's say, you want to pass a number to a function and then have it always ***print*** out its triple value (i.e. var * 3)

```
void tripleIt(double number)
{
   cout << number << "x 3 = " << number*3 << endl;
   return;
}
```

NOTE: the 'return' instruction here is OPTIONAL (why?)

Calling **void** Functions

- void-function calls are, essentially, *executable statements*
  - They do not need to be part of another statement
  - They end with a semi-colon

- Example from previous slide:

  Call it inside of main() with: `tripleIt(32.5);`

  NOT with: `cout << tripleIt(32.5);` ← **Will not compile!!!!**

This distinction is important and a typical rookie mistake to make!!!

void Functions: To Return or Not Return?

- In void functions, we need "return" to indicate the end of the function
  - Is it strictly necessary for that? *No, it's optional*
- Can we use "return" to signal an "interrupt" to the function…
  - …and end it prematurely? *Yes you can do that!*
- Example: What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
  - See example on next page of a void function that avoids division by zero with a return statement

Use of *return* in a *void* Function

Function Declaration

```
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

Function Definition

```
//Definition uses iostream:
void ice_cream_division(int number, double total_weight)
{
    using namespace std;
    double portion;

    if (number == 0)
        return;
    portion = total_weight/number;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Each one receives "
         << portion << " ounces of ice cream." << endl;
}
```

If number *is* 0, *then the function execution ends here.*

The **main** Function in C++ :
*Why is it an **int** type, not a **void** type???*

- The **main** function in a program **is used like a void function**
  - So why do we have to end the program with a return statement?
  - And why isn't it DEFINED as a void function?

- The **main** function is defined to return a value of type **int**,
  therefore a return is needed
  - It's a matter of what is "legal" and "not legal" in C++
  - **void main ( )** is not legal in C++ !! (this ain't Java)
  - Most compilers **will not accept a void main** *(none of the ones we're using, anyway…)*
  - Solution? **Stick to what's legal**: it's ALWAYS **int main ( )**

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

int main ( )
{
    srand(time(0));
    int throw_times, die;

    cout << “How many times shall we throw the die?!\n”;
    cin >> throw_times;

    for (int i=0; i < throw_times; i++)
    {
            die = (rand( ) % 6) + 1;
            cout << “We threw a ” << die << endl;
    }
    return 0;
}
```

What Does This Program Do?

YOUR TO-DOs

- Finish Lab2 by next Monday
- Prepare Lab3 for next Monday
  - description will be put up over the weekend
- Do HW4 by next Tuesday

- Visit Prof's and TAs' office hours if you need help!

- Reverse global warming
  - Bonus points for ending world hunger

</LECTURE>