Advanced Flow Control

CS 16: Solving Problems with Computers I
Lecture #5

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Demos done in class can be found at:**

http://www.cs.ucsb.edu/~zmatni/cs16s17/demos

- **Turn in homework #4 !**

- **Lab #2 due today (at noon)**

- **Homework #5 assigned**
- **Lab #3 will be assigned later on today**
  - Note, this lab is due later than usual

- **Midterm #1 is coming!!!!!**

MIDTERM IS COMING!

- Material: ***Everything*** we've done**, incl. up to Th. 4/20**
  - Homework, Labs, Lectures, Textbook
- **Tuesday, 4/25** in this classroom
- **Starts at 12:30pm **SHARP****
- **Pre-assigned seating**
- Duration: **1 hour long**

- 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

- Boolean Expressions in Flow Control
- Multiway Branches
- Switch Branching
- Command Line Inputs to C++ Programs

- Functions in C++ (on Wednesday)

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 expression
- 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?
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.

- **There are pitfalls to writing nested if/else statements, so be careful in how you write these!!!**
  - Watch your indentations
  - Make use of **{ … }** brackets to make it clear what your intentions are

What’s Wrong With This Code?

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
else
    cout << "Fuel over 3/4. Don't stop now!\n";
```

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**

Alternative for constructing multi-way branches

Syntax is:

```
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 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 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

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

- Variables declared within a block, are **local to the block**
  - i.e. They have the block as their ***scope***.

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

Local vs. Global Variables

- **Local variables** only work in a specified block of statements
- **Global variables** work in the entire program

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

- For example, C++ standard (ANSI) requires that a variable declared in the for-loop initialization section be local to the block of the for-loop

Note on Increments: **num++** vs **++num**

- **(num++)** returns the current value of num, *then* increments it
  - An expression using (num++) will use the value of num BEFORE it is incremented

- **(++num)** increments num *first* and returns its new value
  - An expression using (++num) will use the value of num AFTER it is incremented

- **num has the same value after either version!**

- Example on the next page…

Example: **num++** vs **++num**

```
int num = 2;
int value_produced = 2 * (num++);
cout << value_produced << " " << num;
```

- Displays: 4 3

```
int num = 2;
int value_produced = 2* (++num);
cout << value_produced << " " num;
```

- Displays: 6 3

- In either case, num ends up being 3.
- Works the same way with decrements (-- operator)

Command Line Arguments with C++

- In C++ you can accept **command line arguments**

- These are arguments that are passed into the program from the OS command line

- To use command line arguments in your program, you must add **2 special arguments** in the **main()** function
  - Argument #1 is the number of elements (**argc**)
  - Argument #2 is a full list of all of the command line arguments: ***argv[]**
    - This is an array pointer … more on those in a later class…

Command Line Arguments with C++

- The main() function should be written as:

```
int main(int argc, char* argv[]) { … }
```

- In the OS, to execute the program,
  the command line form should be:

```
$ program_name  argument1 argument2 … argumentn
```

example:

```
$ sum_of_squares 4 5 6
```

DEMO:

```
int main ( int argc, char *argv[] ) {
    cout << "There are " << argc << " arguments here:" << endl;

    for (int i = 0; i < argc; i++)
        cout << "argv[" << i << "] is : " << argv[i] << endl;

    return 0;
}
```

argv[*n*] Is Always a Character!

- All you get from the command-line is character arrays
  - So, the data type of argument being passed is always an array of characters (a.k.a. a C-string)
- To treat an argument as another type, you have to ***convert it inside your program***
- **<cstdlib>** library has pre-defined functions to help!

What If I Want an Argument That’s a Number?

- **<cstdlib>** library has pre-defined functions to help!

- Examples: **atoi( )**, **atol( )**, and **atof( )**
  Convert a **character array** into **int**, **long**, and **double**, respectively.

Example:

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

int main(int argc, char *argv[]) {
   for(int i = 1; i < argc; i++)
        cout << atoi(argv[i]) << endl;
   return 0;  }
```

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* before it
  - Must be placed *before* the function is defined & called

- **Function definition**
  - This is where you define the function itself
  - Must be place *outside* the main()
  - Can be before it or after it

Programmer-Defined Functions

- **Function declaration**
  - Shows how the function is *called* from main() or 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
```

;

Only needed for declaration statement

- **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);

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

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

Declaration

Call

Definition

TO DOs

- Readings
  - Ch. 4 of textbook
- Homework #5 for Thursday
- Lab #3 for 5/1
- Prep for Midterm Exam #1!

</LECTURE>