

Flow Control in C++ 2

CS 16: Solving Problems with Computers I
Lecture #5

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Homework #4 due today**
- **Lab #2 is due on Friday AT NOON!**
 - Use submit.cs
- **Class is closed to new registration**
- **No more switching lab times**
- **Re: Piazza**
 - “Great job” to the people asking and answering questions!

Lecture Outline

- Boolean Expressions in Flow Control
- Multiway Branches
- More about C++ Loop Statements

Precedence Rules on Operations in C++

Precedence Rules

The unary operators `+`, `-`, `++`, `--`, and `!`.

The binary arithmetic operations `*`, `/`, `%`

The binary arithmetic operations `+`, `-`

The Boolean operations `<`, `>`, `<=`, `>=`

The Boolean operations `==`, `!=`

The Boolean operations `&&`

The Boolean operations `||`

*Highest precedence
(done first)*



*Lowest precedence
(done last)*

Applying the Precedence Rule

$$4 * x + 1 > 2 \ || \ x + 1 < -3 \ \&\& \ x \geq 0$$

- Let's figure out what this will do according to precedence rules

- Answer: This is equivalent to:

$$(((4 * x) + 1) > 2) \ || \ (((x + 1) < -3) \ \&\& \ (x \geq 0))$$

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
 - Segmentation errors

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 ( (kids != 0) && (pieces / kids >= 2) )  
    cout << "Each child may have two pieces!";
```



- If the value of kids is zero,
short-circuit evaluation prevents evaluation of $(pieces / 0 \geq 2)$
 - Division by zero causes a run-time error

Boolean and Integer Crossovers in C++

- In C++, you can use an integer as if it's a Boolean
- An integer that's zero can be evaluated as "false"
- An integer that's not zero (usually 1), "true"
- Other languages (eg. Python) reserve actual non-numerical values for Booleans to avoid this
 - Much less confusing...

Potential Trouble With ! Use

- Consider these 2 expressions, where say, `time = 45` & `limit = 60`:
 `!time > limit` (1)
 `!(time > limit)` (2)
- If we are trying to express a Boolean test to verify that `time` is NOT larger than `limit`, then statement (2) is the correct usage.
 - (2): `!(time > limit)`, i.e. `!(45 > 60) = !(false) = true`
- Statement (1) evaluates differently because of precedence rules
 - (1): `(!time) > limit`, or **false**!!

Why?

Potential Trouble With ! Use

- Why does **(!time) > limit** evaluate to **false**?!
 - **time** = 45 (i.e. a non-zero number), so if it's assessed as a Boolean, *that's a true statement*.
 - So then **!time** *evaluates to false, or 0*
- So, the statement becomes:
 0 > limit, or
 0 > 60,
 which is false...

AVOID WEIRD LOGIC SITUATIONS:

1. Always use ()s and use them properly!
2. Before using the **! operator**, see if you can express the same idea without it.

Enumeration of Constants

Data Types

- You'll recall: you can define a constant in C++

```
const int luckyNumber = 7;
```

- You can also define a bunch of constants together with the **enum** type (has to be **int** types):

```
enum MonthLengths {  
    JanLength = 31;  
    FebLength = 28;  
    ...}
```

- Unless specified, the value assigned an enumeration constant is 1 more than the previous constant.
 - If not specified, the first constant is assigned value 0.
 - More on this in the textbook.

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

Simplify This Code

```
if (amount < 10)
    cout << "This number is less than 10\n";
else if ((amount >= 10) && (amount < 25))
    cout << "This number is between 10 and 25\n";
else if ((amount >= 25) && (amount < 40))
    cout << "This number is between 25 and 40\n";
```

```
if (amount < 10)
    cout << "This number is less than 10\n";
else if (amount < 25)
    cout << "This number is between 10 and 25\n";
else if (amount < 40)
    cout << "This number is between 25 and 40\n";
```

A Better Way... Using **switch**

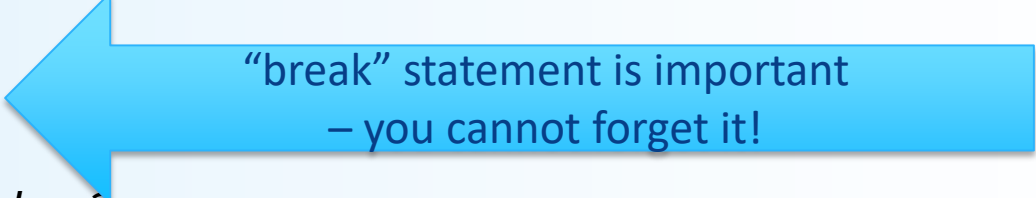
Alternative for constructing multi-way branches

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 enum constant
 - An integer type
 - A character type
- The value returned is compared to the constant values after each "case"
 - When a match is found, the code for that case is used
- Switch will not work with strings in the controlling statement.

Which to Use?

Nested IF/ELSE vs. **switch**

- Nested IF/ELSE statements are more versatile
- Switch statements can make code easier to read
 - Work v. well with “menu types” of applications

Can I Use Functions Inside Multiway Branching?

- Yes!
- Using function calls instead of multiple statements can make the switch or if-else statement much easier to read
- More on C++ functions in a later lecture...

Note About Blocks

- A block is a section of code enclosed by braces
- Variables declared within a block, are local to the block or have the block as their scope.
- Variable names declared in the block cannot be reused outside the block
 - Might not compile in some cases --- SEE DEMO

Note on Increments: **number++ vs ++number**

- (number++) returns the current value of number, then increments number
- An expression using (number++) will use the value of number BEFORE it is incremented
- (++number) increments number first and returns the new value of number
- An expression using (++number) will use the value of number AFTER it is incremented
- **number has the same value after either version!**
- Example on the next page...

Example: number++ vs ++number

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

- displays 4 3

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

- displays 6 3

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

Note on Semicolon Quirks in C++

- Placing a semicolon after nothing creates an empty statement that compiles but does nothing!

```
cout << "Hello" << endl;
;
cout << "Good Bye" << endl;
```

- Placing a semicolon after the parentheses of a **for loop** creates an empty statement as the body of the loop

```
for(int count = 1; count <= 10; count++);
cout << "Hello\n";
```

- This prints one "Hello", but not as part of the loop!

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

Note on Loop Choices

*Recall the differences between **while**, **do-while**, and **for** loops. The following are recommendations, not necessarily “must-dos”*

- **while** loops are the most versatile: Work for any occasion
- **do-while** loops are used for when the loop must always run at least once
- **for** loops typically used when doing numeric calculations, especially when using a variable changed by equal amounts each time the loop iterates.
- If there are circumstance when the loop body should not be executed at all, use a **while** loop.

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

- Yes, the break statement can be used to exit a loop before normal termination
- But it's not good design practice!
- See textbook section 3.3 for details

TO DOs

- Readings
 - The rest of Chapter 3 (i.e 3.4) and Ch. 4, of textbook
- Homework #5
 - Due on Tuesday, 10/11 and submit in class
 - Has a programming question that requires planning ahead!
- Lab #2
 - Due Friday, 10/7, at noon
- Lab #3
 - Given out this weekend

</LECTURE>