CSCI 240 Lecture Notes - Part 3

Cascading Decision Statement (cascading if)

Sometimes the two alternatives provided by **if** aren't enough.

Suppose you need to decide if a student is a Freshman, Sophomore, etc. based on the number of credit hours. There are more than two alternatives.

You can use this pattern - which is an extension of the simple **if**:

```
if (cond1)
  {
  action1;
  }
else if (cond2)
  {
  action2;
  }
else if (cond3)
  {
  action3;
  }
```

If none of cond1, cond2, and cond3 are true, no action is taken.

If you want some action to be taken when *none* of the conds are true, use this pattern:

```
if (cond1)
  {
  action1;
  }
else if (cond2)
  {
  action2;
  }
else if (cond3)
  {
  action3;
  }
else             // note no (cond) here
  {
  action4;
  }
```

action4 is done *if and only if* none of the conds is true.

In the first pattern, *zero or one action* is done. *Never* more than one.

In the second pattern, *one and only one* action is done.

So for the credit-hour - class standing problem, we could code the following:

```
int credits;
string classStanding;

cout<< "Enter number of credit hours: ";
cin >> credits;


if (credits >= 90)
  {
  classStanding = "Senior";
  }
else if (credits >= 60)
  {
  classStanding = "Junior";
  }
else if (credits >= 30)
  {
  classStanding = "Sophomore";
  }
else if (credits >= 0)
  {
  classStanding = "Freshman";
  }
else
  {
  classStanding = "Error: ....";
  }

cout << "\nClass is: " << classStanding;
```

Alternative Decision Statements

We have studied and used the **if** statement to implement the idea of a decision.

There are a couple more ways in C++ to implement decisions that you should know about.

switch Statement

The **switch** statement can cause 0 to many alternative blocks of code to execute based on the value of an integer expression. Normally, it is used to execute one and only one block of code.

Format:

```
switch( Integer Expression )
  {
  case Constant Expression:
    stmt1;
    stmt2;
    break;  //go to stmnt after closing }

  case Constant Expression:
    stmnt3;
    break;

  default:
    stmnt4;
    stmnt5;
  }
```

The ***Integer Expression*** can be any variable that is stored as an integer value or any expression that resolves to an integer value. So the data types int and char can be used with the switch statement, while double, float, and string cannot be tested.

The ***Constant Expression*** on the case statement may be an integer literal or an integer named constant. It cannot be a variable or a conditional expression (such as n == 100 or n < 5).

The ***default*** section will be executed if none of the Constant Expressions on the case statements match the Integer Expression. This section is optional. If it is included in a switch statement, it must come AFTER all off the case statements.

```
int num;

cout << "Enter an integer value: ";
cin >> num;

switch( num )
  {
  case 1:    cout << "You entered 1" << endl;
             break;

  case 3:    cout << "You entered 3" << endl;
             num = num * 3;
             break;

  case 5:    cout << "You entered 5" << endl;
             break;

  default:   cout << "You did not enter 1, 3, or 5" << endl;
  }
```

The *case* statements tell the program where to start executing code. The *break* statement has been used to tell the program where to stop execution within the switch statement. Without the *break*, the program would execute all of the lines from the matching *case* statement to the end of the switch statement.

This "fall through" functionality of not having a *break* statement can be useful if the same task should be performed for several values:

```
char letterGrade;
string result;

cout << "Enter your letter grade: ";
cin >> letterGrade;

switch( letterGrade )
  {
  case 'a':
  case 'b':
  case 'c':
  case 'd':   result = "pass";
              break;

  case 'f':   result = "fail";
              break;

  case 'i':
  case 'n':
  case 'w':   result = "other";
              break;

  default:    result = "invalid";
  }
```

If letterGrade is 'b', for example, it will match case 'b' and go there and then just "fall through" the case 'c', case 'd', assign "pass" to result, and then break out of the switch structure.

Condition Operator (? and :) - a "ternary" operator

A short and somewhat cryptic decision structure is allowed by C++. The general format is:

condition ? stmnt_to_do_if_true : stmnt_to_do_if_false;

Some compilers might require the condition to be enclosed in a set of parenthesis ().

For example:

```
x < y ? x++ : y++;
```

This is the same as writing:

```
if (x < y)
  x++;
else
  y++;
```

But there is one difference. The expression as a whole *becomes* the value computed in one of the two "branches". Here are a couple of examples. Note the () around the whole ternary expression:

```
char ch = 'y';
int x = 3, y = 2;

cout << ((ch == 'y') ? "Yes" : "No");

cout << "The value of the smaller of x and y is " << ((x < y) ? x : y);
```

This would display "YesThe value of the smaller of x and y is 2" This somewhat terse syntax can be handy sometimes.

Compound Conditions

Sometimes one condition is not enough.

Suppose we want to test if a point is inside a rectangle, and we know the x and y coordinates of the point; also we know the left, right, top, and bottom coordinates of the rectangle. So we declare:

```
int ptX, ptY;
int rectLeft, rectRight, rectTop, rectBot;
```

Now assume all of these variables have valid values and Cartesian coordinates. If we wanted to test if a point is inside the rectangle, then:

```
if ptX is greater than rectLeft  AND if ptX is less than rectRight AND
if ptY is greater than rectBot   AND if ptY is less than rectTop

THEN (meaning, if ALL of these are true...), the point is in the rectangle.
```

We can code this in C++, using **&&** for the AND's (newer C++ compilers also allow the word *and*):

```
if (ptX > rectLeft && ptX < rectRight && ptY > rectBot  && ptY < rectTop)
  {
  cout << "pt is in rect";
  }
else
  {
  cout << "pt is not in rect";
  }
```

Note that we required that the point be *inside* the rectangle because we used > and <.

We could also allow the point to be *on or inside* the rectangle by using <= and >=.

How could we check to see if the point lies ON the top edge of the rectangle?

(Try it)

We can also code **OR**'s of two conditions, using 2 vertical bars: || (again, newer compilers allow *or*):

Example: Suppose you want to ask if the user wants to continue:

```
cout << "Another? (y/n)";
cin >> choice;
```

But - you want to accept either "y" or "Y" to indicate "continue". (The user may have hit caps lock, or just carelessly hit shift).

```
do
  {
  ... some processing of one thing...
  cout << "Another? (y/n)";
  cin >> choice;
  }
while (choice == "y" || choice == "Y");
```

Another example: suppose your program is accepting a number representing a shoe size, and you know that shoe sizes cannot be smaller than 5 or greater than 18. Validate this number - that is, *make* the user type in a "valid" shoe size. To do this use a compound condition:

```
cout << "Enter a shoe size";
cin >> size;

if (size < 5 || size > 18)
  {
  cout << "invalid size....Try again: ";
  cin >> size;
  }
```

This is good, but there is a better way to do error checking:

Version 1:

```
cout << "Enter shoe size (5 to 18): "; //initial prompt
cin >> size;

while (size < 5 || size > 18)           // OR not AND!!
  {
  cout << "Shoe size must be in range 5 to 18; try again: ";
  cin >> size;
  }

cout << "You entered " << size;
```

This version needs two *input* lines, and uses two different messages. (The second only if a data entry error has been made.)

Sample script:

```
Enter the shoe size (5 to 18): 3
Shoe size must be in range 5 to 18; try again: 19
Shoe size must be in range 5 to 18; try again: 5
You entered 5
```

Note clearly the loop exit condition: you want to do the loop only if the user has entered a **bad** value. So you code "while the size is **bad** - i.e. less than 5 or greater than 15"

If the initial value entered by the user is ok - say 10 - then the loop body won't execute *at all* because *neither* of the conditions is true. 10 is not less than 5; 10 is not greater than 18. So both subconditions are false, so the whole thing is false, so the loop is not executed.

Don't code "while (size < 5 && size > 18)" because an invalid size would pass as valid: Suppose the user types in 20. 20 < 5 is false. 20 > 15 is true. Since both are not true the whole thing is false and so the loop body would not execute and so the invalid size would pass as valid.

Another way to understand it is simply: how could ANY size be *both* less than 5 *and at the same time* greater than 18? It's impossible. So ANY shoe size would create one false condition and so the loop body would never be executed for ANY shoe size, valid or invalid.

Clearly, compound conditions require careful thought.

Version 2:

```
do
  {
  cout << "Enter the shoe size (5 - 18): ";
  cin >> size;

  // Note changes to cond!!
  if (size >= 5 && size <= 18)
    break;
  }
while (1);
cout << "You entered " << size;
```

Note the new condition. Whereas before we wanted to *execute* the loop for an *invalid* value, now we want to *break out* of the loop when we have a *valid* value.

This version will work. But there's only one message. So user sees:

```
Enter the shoe size (5 - 18): 3
Enter the shoe size (5 - 18): 19
Enter the shoe size (5 - 18): 5
You entered 5
```

It's not 100% clear that the first two tries were wrong. We could put an *else* to the *if* and patch it to print out an error message. But Version 1 is probably better.

Combining && and ||

Truth tables:

```
(true   &&  true)  =>  true
(true   &&  false) =>  false
(false  &&  true)  =>  false
(false  &&  false) =>  false
```

So for (c1 **&&** c2) to be true, **both must be true**.

```
(true   ||  true)  =>  true
(true   ||  false) =>  true
(false  ||  true)  =>  true
(false  ||  false) =>  false
```

So for (c1 || c2) to be true, **one or the other or both must be true**.

There are rules of evaluation for these operators:

1. The evaluation looks at one condition at a time (i.e. cond1 op cond2)
2. sub expressions inside () are done first
3. && binds tighter than || (like * and / are done before + and -)
4. otherwise, conditions are evaluated left to right
5. as soon as the truth of the whole is known, evaluation stops
6. but you can provide your own () to change the default rules (or to make the evaluation order clearer to the reader).

Suppose you have:

if (cond1 || cond2 && cond3)

The whole thing has to be true or false.

The parts are considered in pairs.

```
Now will C++ look at this as  (cond1 || cond2) && cond3

or as                          cond1 || (cond2 && cond3)
```

Because && precedes ||, the latter. So

1. Evaluate cond1 (left-to-right rule).

2. If true, done.

3. Else (cond1 is false, so) evaluate (cond2 && cond3). If both are true, the whole condition is true; otherwise it is false.

Example:

We want to judge num as valid if it is between 0 and 100 **or** if it is 2000.

```
if (num == 2000 || num >= 0 && num <= 100)
  cout << "valid";
else
  cout << "not valid";
```

1. Suppose num is 30. Since num is NOT = 2000, we have to evaluate the second half of the condition. Since 30 is >=0 AND is <= 100, the && is satisfied and we have

   ```
   if (false || true)
   ```

   Since this is an OR and for an OR, the whole thing is true if either is true - the whole thing is true.

2. Suppose num is 2000. Since the first half is true, we don't have to evaluate the second half. We have:

   ```
   if (true || anything)
   ```

   Since this is an OR, it can be true if just one part is true, so the whole thing is true.

3. Suppose num is 150. Since num is NOT = 2000, we have to evaluate the second half of the condition. 150 >= 0 BUT 150 is NOT <= 100. So the && is not satisfied and we have:

   ```
   if (false || false)
   ```

   Since both halves are false, the whole thing is false.

Data type bool and flags

C++ defines a data type called *bool*. (Short for "boolean", from the name of a 19th century Irish logician, George Boole)

bool variables can have one of two possible values: *true* and *false*

This data type comes in handy in writing programs that are easier to understand.

For example, consider the following partial example:

```
bool found = false;

while ( !found )
  {
  //some complex code to search for something...
  //somewhere in it is the following:

  if (some condition is true)
    found = true;
  }
```

These kinds of variable that can be true or false are often called *flags*. They typically indicate if some event has happened yet or not, or as a category:

```
bool okToWithdraw;

double bal, request;


//get user's bank balance and store into bal

//get user's withdrawal request and store into request


if (bal - request > 0)
  okToWithdraw = true;
else
  okToWithdraw = false;

...

if (okToWithdraw)
  // give out money
else
  // print error message
```

Scope

The **scope** of a variable is the location within a program where a variable can be used.

With control structures, all of the statements that are contained within a block of code belong to the control structure. Therefore, any variable that is declared within such a block of code only has meaning between the declaration for the variable and the closing curly brace for the block of code.

For example:

```
for( int i = 0; i < 10; i++ )
{
cout << endl << i;            //this line of code will compile
}

cout << i;                //this line of code will NOT compile
```