All about flow control

Prof. Zhang

March 11, 2014

1 Read Coding Style Guideline

Please go to the class website, find the resource part, click on the "how labs are graded?" link. Read the guideline, and then follow the "Quick Hints" to fix the indentation of your lab4.cpp (before you submit the labs).

2 Basics

In order to do something interesting, we need to be able to

- branch off in our code, i.e., execute different codes depending on whether some condition is met or not.
- repeat some codes for a certain number of times, or until some condition becomes true

C++'s answer to the above two needs are

- **if-else** statement, **if** statement, and **switch** statement
- **for** loop, **while** loop, and **do-while** loop.

3 Branch off

3.1 if-else statements

The basic format (or syntax) or a if-else statement is as follows

```
if (condition)
   yes_statement;
else
   no_statement;
```

The semantics (or meaning) of the above statement is

1. first check *condition*'s value
2. if condition's value is true, execute *yes_statement*
3. if condition's value is false, execute *no_statement*
4. execute next statement.

3.2 if statement

The basic format (or syntax) or a if statement is as follows

```
if (condition)
   yes_statement;
```

The semantics (or meaning) of the above statement is

1. first check *condition*'s value
2. if condition's value is true, execute *yes_statement*
3. execute next statement .

3.3 Nested if-else and if statements

Nested if-else and if statements: the yes_statement or no_statement itself can be a if-else or if statement. For example:

```
if (x==1)
  if (y==2) //this if-else statement is the yes_statement for the if (x==1)
    cout <<"x==1 and y==2\n";
  else
    cout <<"x==1 and y!=2\n";
else
  if (y==3) //this if statement is the no_statement for the if (x==1)
cout <<"x!=1, y==3\n";
```

Note that the above code is not hard to understand due to the indentation style. Now compare that with the following:

```
if (x==1) if (y==2) //this if-else statement is the yes_statement for the if (x==1)
cout <<"x==1 and y==2\n";
else cout <<"x==1 and y!=2\n";
else
  if (y==3) //this if statement is the no_statement for the if (x==1)
cout <<"x!=1, y==3\n";
```

Both reads the same to the compiler, but the second one gives everyone a big headache to figure out. So often the first step to debug your program, or even just to understand it, is to fix its indentation.

3.4 Multi-way branch

Multi-way branch We often use nested if-else statement to express a multi-way branch, for example,

```
if (coin=="quarter")
   value=25;
else
  if (coin=="dime")
     value=10;
  else
    if (coin=="nickle")
       value=5;
    else
       if (coin=="penny")
         value=1;
       else
         cout <<"Invalid coin name" << coin <<"\n";
```

A conventional way to format a multi-way branch such as above is follows:

```
if (coin=="quarter")
   value=25;
else if (coin=="dime")
   value=10;
else if (coin=="nickle")
   value=5;
else if (coin=="penny")
   value=1;
else
   cout <<"Invalid coin name" << coin <<"\n";
```

Compare the above code segment with the following (where 5 consecutive if statements are used):

```
if (coin=="quarter")
   value=25;
if (coin=="dime")
   value=10;
```

```
if (coin=="nickle")
   value=5;
if (coin=="penny")
   value=1;
if (coin!="quarter" && coin!="dime" && coin!="nickle" && coin!=penny")
   cout <<"Invalid coin name" << coin <<"\n";
```

3.5 Dangling else problem

Dangling else problem is a problem in computer programming in which an optional *else clause* in an **if(else)** statement results in nested conditionals being ambiguous.

```
if (condition1)
   if (condition2)
      yes_statement;
else  //is this else for the first if, or the second if ?
   no_statement;
```

In C++, the compiler will associate the **else** with the **nearest** unmatched if statement. So in the above example, the **else** is associated with **if** condition2. If you want the **else** to be associated with the first **if** condition, do the following:

```
if (condition1)
{ //use {} to demarcate this if statement
    if (condition2)
       yes_statement;
}
else
   no_statement;
```

3.6 switch statement

switch statement has the following syntax:

```
switch (expression)
{
   case constant1:
      statement11;
      ...
      break;
   case constant2:
      statement21;
      ...
```

```
      break;
   ...
   default:
       statement31;
       ...
}
```

There are some constraints about **switch** statement:

- The expression that we "switch" on needs to be **integetral** type, including **int, long, char, bool**.
- The switch "case" needs to be constants (named constants or literal constants)

The meaning (or semantics) of the switch statement can be explained in terms of the equivalent **if-else** statement:

```
if (expression==constant1)
{
   statemt11;
   ...
}
else if (expression==constant2)
{
   statemnet21;
   ...
}
...
else {
   statement31;
   ...
}
```

Note: Pay special attention to the **break** statements at the end of each cases. As beginners, use **break** always. Without break statement, the flow of execution cascade down to next case, and the case after (if the next case also does not have break statement).

```
char lettergrade;
...
switch (lettergrade)
{
   case 'A':
     cout <<"you got 90-100\n";
```

```
   case 'B':
    cout <<"You are above average\n";
    break;
   case 'C':
    cout <<"Try harder!\n";
    break;
   case 'D':
   case 'F':
    cout <<"Danger of failing!\n";
    break;
   defalut:
    cout <<"Invalid grade\n";
}
```

Here are the outputs of the program depending on lettergrade's value:

- A: you got 90-100 you are above average
- B: You are above average
- C: try harder!
- D: Danger of failing!
- F: Danger of failing!
- other values: invalid grade

4 Understand and Build Loop

4.1 for loop statement

The basic format (or syntax) or a **for** loop statement is as follows

```
for (initialiation; condition; update)
   statement_to_repeat;
```

The semantics (or meaning) of the above statement is

1. first execute **initialization**
2. **condition** is checked,
3. if **condition** is true, executes *statement*, then *update*, and then goes back to step 2.

4. if **condition** is false, for loop ends, execution continues from the next statement (after the for loop statement)

A **for** loop can be converted to a **while** loop as follows:

```
initialization;

while (condition)
{
   statement_to_repeat;
   update;
}
```

For example, the following code prints out integers from 0 to 99:

```
for (int i=0;i<100;i++)
   cout << i <<"\n";
```

First, variable i is declared and initialized to 0 (in the initialization of the loop). As long as i is less than 100, the loop body prints out i's value, and then i is incremented by 1. This continues until *i==100*.

Note that the above *for* loop is the most commonly used form. Variable i is called *counter* variable, it counts up from 0 to 100 by 1. One can count up, or count down, or by other increment other than 1.

```
//this for loop calculate the sum of all even numbers between 0 and 98
sum=0;
for (int i=0;i<100;i=i+2)
   sum = sum+i;
```

Exercises:

- Write a for loop to print out 30 asterisk signs on the same line We can count up:

  ```
  for (int i=0;i<30;i++)
     cout <<"*";
  ```

  or count donw:

  ```
  for (int i=30;i>0;i--)
     cout <<"*";
  ```

- Write a for loop to print power of 2's, i.e., 2^0, 2^1, 2^2,..., up to 2^{31}. We first initilaizes our **power** variable that keeps track the current power, and then keeps doubling its value:

```
int power=1;
for (int i=0;i<32;i++)
{
  cout <<"2^" << i <<"= "<<power;
  power=power*2;
}
```

for loop can be nested, for example, the following code prints multiplication table:

```
for (int i=1;i<10;i++) //we call this outer loop
{
    for (int j=1;j<10;j++) //we call this inner loop
        cout << " " << i*j << " ";
    cout <<"\n";
}
```

Let's **follow (trace)** this loop to see how it works...
Exercises:

- Write a program to print a rectangle made up of asterisks, the length and height of the rectangle are given by the user. For example, if the length is 5, and the height is 3, the output should be:

```
*****
*****
*****
```

- Write a program to print a right triangle made up of asterisks of a given side length. For example, if the side length is 5, the output should be:

```
*
**
***
****
*****
```

- Write a program to print a triangle of the following shape with a given side length. For example, if the side length is 4, the output should be:

```
   *
  ***
 *****
*******
```

4.2 while loop statement

The basic format (or syntax) or a **while** loop statement is as follows

```
while (condition)
   statement;
```

The semantics (or meaning) of the above statement is

1. **condition** is checked,
2. if **condition** is true, executes *statement*, and then goes back to step 1.
3. if **condition** is false, while loop ends, execution continues from the next statement (after the loop statement)

The most simple **while** loop is as follows:

```
int i=0;
while (true)
{
   i++;
   cout <<"i=" <<i <<"\n";
}
```

Exercise

- What will happen when the above while loop is executed?
- How to make the loop ends when i reaches the maximum int value? Use **break** statement

```
// To read any number of values, end with a -1
int num;
while (true)
{
   cout <<"Enter next value:";
   cin >> num;

   if (num==-1)
break;
}
```

Preferred way to write such as loop is to avoid using *break* statement. What do you think are the downsides of using break statement (one or multiple?)?

We prefer to express the loop condition in the **while** header only, so that it's easy to see when the loop continues, as the loop can be very long and complicated.

```
// To read any number of values, end with a -1
int num=0;
while (num!=-1) // as long as value is not -1, continue...
{
   cout <<"Enter next value:";
   cin >> num;
}
```

4.3 do-while loop

The **do-while** loop has the following syntax:

```
do
  statement;
while (condition);
```

The semantics (or meaning) of the statement is as follows:

1. execute *s*tatement first
2. evaluate *c*ondition, if it's true, go back to 1

The difference between **while** loop and **do-while** loop is that **while** loop might iterate for zero time (if the condition is false the first time), while **do-while** loop always iterate the first time.