Chapter 3: More Flow of Control

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Using Boolean Expressions

Relational and Equality Operators

- Relational and equality operators return boolean values (boolean meaning true or false)
- Relational operators
  - `<` less than
  - `>` greater than
  - `<=` less than or equal to
  - `>=` greater than or equal to
- Equality operators
  - `==` equals
  - `!=` not equals

Unary Operators

- Unary means it only acts on a single variable (whereas so far other operators have acted on two variables/values)
- Unary operators
  - `&` address-of (gets the memory address of a variable)
  - `*` contents-of (gets the contents stored in a memory address)
  - `-` negation
  - `+` plus
  - `!` logical negation
  - `(type)` type casting
    - *Type casting* allows us to convert one variable type into another type
    - Notice that this does **not** round, it truncates
    - For instance, common use is to typecase a floating point number into an integer:

```
1 int b = (int)3.5; // 3.5 will be truncated to 3
```

Logical Operators

- `!` logical negation

- && logical AND
- || logical OR

Precedence

- 1 (highest)
  - ++/−−Postfix increment and decrement
  - () Function calls
  - [] Array subscripting
  - . structure/union access
  - −> structure/union member access through pointer
- 2 (second highest)
  - ++/−− Prefix increment and decrement
  - +/− unary plus and minus
  - ! logical not
  - (type) type casting
  - * dereference
  - & address-of
- 3 (third highest)
  - * multiplication
  - / division
  - % remainder division
- 4 (forth highest)
  - + addition
  - − subtraction
- 5 (fifth highest)
  - < less than
  - > greater than
  - <= less than or equal to
  - >= greater than or equal to
- 6 (sixth highest)
  - == equal
  - != not equal
- 7 (seventh highest)
  - && logical AND
- 8 (eighth highest)
  - || logical OR
- 9 (ninth highest)

    - `?:` ternary conditional
- 10 (tenth highest)
    - = assignment (and all other forms of assignment, `+=`, `-=`, `*=`, `/=`, `%=`)
- 11 (eleventh highest)
    - , comma (for creating multiple variables)

Logical Expressions

- A way to evaluate operations over logical values (i.e. `0` for false and anything else for true)
- Gives a way to encode "this AND that" or "this OR that"
- Consider the following:

```
a || b    // 1 when EITHER a OR b is true, 0 otherwise
a && b    // 1 when BOTH a AND b are true, 0 otherwise
!a        // 1 when a is false, 0 otherwise
```

- We can string multiple logical expressions together to create compound logical expressions:

```
((a && b) || (c > d))
```

Multiway Branches

Nesting

If statements can be nested, meaning you have **`if`** statements inside of **`if`** statements

nesting.cpp

```
/* Let's try writing some nested conditional statements... */
#include <iostream>

using namespace std;

int main() {
   int temperature;
   /* Prompt for values */
   cout << "\t\tNested Logic Program\n\n";
   cout << "Please enter today's temperature: ";
   cin >> temperature;

   if (temperature < 50) {
```

```
14          cout << "Gosh, it feels cold...\n";
15          if (temperature < 32) {
16             cout << "And it looks like it's freezing...\n";
17          }
18          else if (temperature < 40) {
19             cout << "And it's nearly freezing...\n";
20          }
21          else {
22             cout << "But atleast it's not freezing cold!\n";
23          }
24      }
25      else if (temperature > 90) {
26          cout << "Gosh, it's hot...\n";
27          if (temperature > 110) {
28             cout << "And it's just boiling... head for air conditioning
                   ...\n";
29          }
30          else if (temperature > 100) {
31             cout << "Atleast it's not boiling...\n";
32          }
33          else {
34             cout << "What a heat wave!!\n";
35          }
36      }
37      else {
38          cout << "Doesn't California have a nice climate!\n";
39      }
40      return( 0 );
41  }
```

- The conditional expression is an if statement that can be assigned to a variable. It is commonly called the *ternary operator*
  - The syntax is the following:

```
1  (/* logical expression goes here */) ? (/* if non-zero (true) */) : (/*
       if 0 (false) */)
```

- If the logical expression is true, the overall condition evaluates to the expression between the `?` and the `:`.
- If the logical expression is false, the overall condition evaluates to the expression after the `:`
- For example, if we want to set `c` to be the larger value of two variables `a` and `b`, we could write the following:

```
c = (a > b) ? a : b;
```

Switch-Case Statement

- The **switch...case** statement enables us to write many "cases" that could be handled by **if...else** in a manner that is sometimes cleaner than **if...else** statements.
  - However, the **switch** statement only switches on an integer or enum type, so if you are branching/selecting on a more complex type, you'll have to use **if...else**
- Basic syntax:

```
switch (/* integer or enum goes here */) {
case /* potential value of the aforementioned int or enum */:
  /* code */
case /* a different potential value */:
  /* different code */
/* insert additional cases as needed */
default:
  /* more code */
}
```

- The switch uses a variable, and integer or enum, to control which case to evaluate. This is a limiation; if you must compare more complicated data, you cannot use a **switch...case**
- This variable is compared against each **case**, one the comparison is true, that particular case will activate (execute)
  - Once a **case** has been activated, no other cases will be evaluated
- Typically, the last statement for each case is a **break** statement. The causes the program to jump to the statement following the closing } of the switch statement.
  - This basically ends the switch statement (and this behavior is probably your intuition behind each case
  - However, if you omit the **break**, the cases "fall throw" until the end of the switch or until a **break** is reached
- If no cases are matched and a **default** case is specified, the default case will execute.
  - Use of **default** is optional.

multiselect.cpp

```
/* Let's try writing a switch statement... */
#include <iostream>
```

```
3
4  using namespace std;
5
6  int main() {
7     char letter;
8     /* Prompt for values */
9     cout << "\t\tCase Statement Program\n\n";
10    cout << "Please enter a letter to inspect: ";
11    cin >> letter;
12
13    /*
14      FYI: you can only switch on a
15      integral value.  The char datatype is just another
16      name for the set of ints between 0 and 255, so you
17      can switch on chars or ints
18     */
19    switch( letter ) {
20     /*
21       Individual letters must be single-quoted.
22       Individual letters map directly to constant
23       integer values based on the ASCII table which
24       we will learn about in upcoming units.  The
25       value of each case must be a constant value,
26       not an expression or variable.  This often
27       makes switch statements not applicable to your
28       situation.
29     */
30     case 'a':
31     case 'e':
32     case 'i':
33     case 'o':
34     case 'u':
35     case 'y':
36      /*
37       Lacking break statements in the upper
38       listed cases, they will all "fall thru"
39       to the set of statements shown here.
40       While at first this may seem very convenient,
41       this is actually the number one programming
42       bug worldwide.  Namely, that folks forget that
43       all the above cases are collapsing down to
44       the code shown below.  So use this form
45       with great caution, as it often leads to
```

```
46        bugs...
47        */
48        cout << "a nice lowercase vowel!\n";
49        break;
50      case 'A':
51      case 'E':
52      case 'I':
53      case 'O':
54      case 'U':
55      case 'Y':
56        cout << "a nice UPPERCASE vowel!\n";
57        break;
58      case '0':
59      case '1':
60      case '2':
61      case '3':
62      case '4':
63      case '5':
64      case '6':
65      case '7':
66      case '8':
67      case '9':
68        cout << "a nice number!\n";
69        break;
70      default:
71        /*
72         The default case is the one selected when
73         no other cases actually match the switched
74         data
75        */
76        cout << "this is not something I recognize...\n";
77        break;
78    }
79    return( 0 );
80  }
```

More C++ Loops

For Loops

- Functionally equivalent to a while loop, but people find them to be more readable/maintainable.
- Typically in a while, you'd put some code to modify the controlling condition as the last statement

to the while loop (increment, decrement, etc)
  - A for loop moves this to the definition of the loop
- Syntax:

```
for (initialization; controlling condition; loop-ending statement) {
  /* code */
}
```

- The *initialization* statement is executed once - at the beginning of the loop
  - Typically, you would assign some variable to be a particular value in this loop section
- The *controlling condition* is the test executed to determine whether or not the loop should run again.
  - It is checked when the loop starts.
- The *loop-ending statement* is typically a form of incrementing/decrementing a value.
  - This statement is executed at the end of every loop statement, but before the controlling condition is checked
  - If you used a **`continue`** statement, this statement is also executed (i.e. it is not skipped because of the use of a **`continue`**).
- Any of these may be omitted.
  - You do not have to run an initialization statement
  - You do not have to provide a controlling condition
    - What must you do to make sure your loop terminates if this is omitted?
  - You do not have to provide a loop ending statement
    - What must you do to make sure your loop terminates if this is omitted?
- Counting example:

```
int i;
for (i = 1; i <= 10; i++) {
  cout << i << " ";
}
```

- A for loop can be given no conditions:

```
for (;;) {
  /* block of statements */
}
```

- This is an infinite loop because it will loop forever unless there is a break statement in the block for the loop
- You may also use the comma operator to add multiple statements inside the loop:

```
int i, j, n = 10;
for (i = 0, j = 0; i <= n; i++, j += 2) {
  cout << "i = " << i << ", j = " << j << endl;
}
```

Exercises

Write a C++ program to find whether a given year is a leap year or not.

```
#include <iostream>

using namespace std;

void main()
{
    int chk_year;

    cout << "Input a year :");
    scanf("%d", &chk_year);
    if ((chk_year % 400) == 0)
        cout << "%d is a leap year.\n", chk_year);
    else if ((chk_year % 100) == 0)
        cout << "%d is a not leap year.\n", chk_year);
    else if ((chk_year % 4) == 0)
        cout << "%d is a leap year.\n", chk_year);
    else
        cout << "%d is not a leap year \n", chk_year);
}
```

Write a C++ program to read any day (7 days of the week) in integer form (as a number) and display day name using the corresponding word 1 - Monday 2 - Tuesday 3 - Wednesday … 6 - Saturday 7 - Sunday

```
#include <stdio.h>
void main()
{
  int dayno;
  cout << "Input Day No : ");
  scanf("%d",&dayno);
  switch(dayno)
  {
    case 1:
```

```
10          cout << "Monday \n");
11          break;
12       case 2:
13          cout << "Tuesday \n");
14          break;
15       case 3:
16          cout << "Wednesday \n");
17          break;
18       case 4:
19          cout << "Thursday \n");
20          break;
21       case 5:
22          cout << "Friday \n");
23          break;
24       case 6:
25          cout << "Saturday \n");
26          break;
27       case 7:
28          cout << "Sunday  \n");
29          break;
30       default:
31          cout << "Invalid day number. \nPlease try again ....\n");
32          break;
33    }
34 }
```