Characters and the SWITCH Statement

http://people.sc.fsu.edu/∼jburkardt/isc/week05
lecture_09.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 07 June 2011

Characters and the SWITCH Statement

INTRO: Midterm Exam

The **Midterm Exam** is scheduled for Thursday, June 30th.

It will be an in-class exam.

It is not an open book exam, but you will be allowed to bring and use two sheets of (normal size) paper on which you have notes.

Detelina will be available on Wednesday, June 29th, during lab hours, to review with you.

I will talk more about the exam on Tuesday, June 28th.

The programming assignment #6, given on June 23rd will not be due on June 30th; it will be due on July 7th instead.

INTRO: Schedule

Read:

- Sections 4.6, 4.7

Next Class:

- The Math Library, Functions and Parameters

Assignment:

- Today: in-class lab exercise #5
- Thursday: Programming Assignment #3 is due.

INTRO: Today's Topics

Today we will quickly go over the **continue** statement, which is related to the **break** statement we saw last time. Instead of completely terminating a loop, the **continue** statement jumps to the end of the current set of statements, allowing the next cycle of the loop to begin.

Then we will introduce the topic of how C++ stores and handles single characters.

We will look at an example of how to read characters, one by one, from a file.

INTRO: Today's Topics

Then we will look at an example, using a combination of a **while** statement and a **break** statement, in which we capitalize the letters read from a file to make an UPPERCASE VERSION.

Then we will look at the **switch** statement, which allows us to describe a situation in which we are going to choose one of many actions, based entirely on the value of some variable.

Our in-class lab exercise will read characters from a file, using a **switch** statement to count characters, and make a histogram of character frequency in the file.

Characters and the SWITCH Statement

CONTINUE: Skip to Next Loop Cycle

The **continue** statement is used inside a looping statement, such as **for**, **while** or **do...while**.

```
while ( condition )
{
   some statements;
   continue;
   more statements;
}
```

Typically, it occurs somewhere in the middle of the statements being repeated. The effect of the **continue** statement is to skip over the statements which follow it in the loop, but to start th next cycle of the loop, rather than to terminate the loop (wh ch what the **break** statement does.)

CONTINUE: Skip to Next Loop Cycle

Of course, if the **continue** statement showed up the way I wrote it in the previous example, the statements after it would never be executed. What's more likely is that we want to skip those statements if some condition is true:

```
while ( condition )
{
  some statements;
  if ( some other condition )
  {
    continue;
  }
  more statements;
}
```

This would make sense if the following statements were only necessary when the condition was not true.

CONTINUE: Skip to Next Loop Cycle

The **continue** statement is another convenience. You can always do the same thing by using an **if** statement in the right way, but sometimes in big programs the **continue** statement makes it easier to see what is going on.

Let us consider a computation in which, for any integer **n** between 1 and 20, we want to compute the product:

product $= (n-1) * (n-2) * ... * (n-20)$.

As we have described it, product will be zero, because **n** is actually equal to 1, or 2 or one of the other integers.

However, let's suppose we actually want to compute this prod[illegible]t by skipping the step where the factor is 0.

CONTINUE: Skip to Next Loop Cycle

We can use a **for** statement to do the loop, and a **continue** statement to jump to the next step in a **for** statement just as it does for a **while**:

```
product = 1;
for ( i = 1; i <= 20; i++ )
{
  if ( i == n )
  {
    continue;
  }
  product = product * ( n - i );
}
```

If **n** is 7, we correctly skip the zero factor:
product = 6 * 5 * 4 * 3 * 2 * 1 * (-1) * (-2) ... * (-13).

Characters and the SWITCH Statement

- Introduction
- The CONTINUE Statement
- **Character Data**
- Reading Characters
- The SWITCH Statement
- ETAION SHRDLU
- Lab Exercise #5

CHAR: a Single Character Data Type

If you're trying to alphabetize a list of names, you don't compare numbers, but rather letters. C++ prefers to use the word **characters** rather than **letters**, because we want to think about messages that include not just 'A', 'B', 'C', but also '&' and '7' and '[' and so on.

C++ provides the variable type **char** for characters.

We want to know some simple things about character data:

- How do we represent a literal character?
- Can we uppercase a character? Compute the next character?
- How do we read and write single characters?
- What about special characters?
- How is the computer storing characters?

CHAR: Literal Values

The literal value of a character can be given using single quotes.

```
char capA = 'A', lowA = 'a', capB = 'B';
char c;

c = 'a';
if ( c == capA ) cout << c << " == " << capA << "\n";
if ( c == lowA ) cout << c << " == " << lowA << "\n";
if ( c == capB ) cout << c << " == " << lowB << "\n";
```

The single quote can be confusing, since text strings use double quotes. And so 'w' is a character, but "w" is a (very short) stri
We're not ready to worry about strings right now!

CHAR: Reading and using a CHAR

You might want the user to type in 'Y' or 'N' for yes or no.

```
char answer;

cout << "Do you want help? (Y/N): ";
cin >> answer;

if ( answer == 'y' || answer == 'Y' )
{
  cout << "Here is some helpful information...\n";
  ...and here we print stuff out.
}
```

CHAR: Capitalizing a CHAR

It's frustrating to have to worry about whether a character is uppercase or lowercase. One thing you can do is use the functions **toupper()** or **tolower()** from <**cctype**>.

No matter whether the character **c** is lowercase or uppercase, for instance, **toupper(c)** returns an uppercase version of it.

```
# include <cstdlib>
# include <iostream>
# include <cctype>     <-- Need this at beginning!

using namespace std;

int main ()
{
  char answer;

  cout << "Do you want help? (Y/N): ";
  cin >> answer;

  if ( toupper ( answer ) == 'Y' )
  {
    cout << "Here is some helpful information...\n";
    ...and here we print stuff out.
  }
  return 0;
}
```

CHAR: How Are CHAR's Ordered?

If we want to alphabetize things, then we can use the comparisons $<$ and $<=$ and $>$ and $>=$ and **==** and **!=**.

```
char capa = 'A', capt = 'T', lowb = 'b', lowt = 't';

cout << "Is A < b?: " << ( capa < lowb ) << "\n";
cout << "Is T < b?: " << ( capt < lowb ) << "\n";
cout << "Is t < b?: " << ( lowt < lowb ) << "\n";
```

Remember, **(capa** $<$ **lowb)** is a logical variable, so you can print it, and it will come out as "0" for false, or "1" for true.

If you insert $<<$ **boolalpha** before the printout, logical variables will print out as "true" or "false" instead.

```
cout << "Is A < b?: " << boolalpha << ( capa < lowb ) << "\n";
```

All capital letters come *before* any lowercase letters, so:
'A' $<$ 'B' $<$ 'C' $<$ ... $<$ 'Z' $<$ 'a' $<$ 'b' $<$ 'c' $<$ ... $<$ 'z'.

CHAR: How Are CHAR's Ordered?

Actually, I left out some information about the ordering of characters. C++ orders all sorts of numbers and characters besides the usual alphabetic letters, so the sequence that includes the alphabet looks like this:

```
... @ A B C D ... Y Z [ \ ] ^ _ ' a b c d ... y z { | }...
```

Luckily, this is still pretty simple. In particular, if we have a character **c**, and we want to know if it represents a letter of the alphabet, we simply check if

```
( 'A' <= c && c <= 'Z' ) || ( 'a' <= c && c <= 'z' )
```

If that's too complicated, you can instead **# include <cctype>** and then call the function

```
isalpha ( c );
```

which returns **true** if **c** is alphabetic.

CHAR: Using a CHAR Variable

average.cpp: The book likes examples that involve letter grades:

```
char grade;
float average;

if ( 90 <= average )
{
  grade = 'A';
}
else if ( 80 <= average )
{
  grade = 'B';
}
else if ( 70 <= average )
{
  grade = 'C';
}
else if ( 60 <= average )
{
  grade = 'D';
}
else
{
  grade = 'F';
}
cout << "Your grade is " << grade << "\n";
```

Why can't we assign a grade of B+ this way?
What if we **really** want plus and minus grades?

Characters and the SWITCH Statement

READ: Reading Character Input

Suppose we wanted to allow the user to type a bunch of letters, and we plan to read them in, one character at a time.

How can we read the data, and how will we be able to stop the process?

Our idea is to do this in a way that makes it easy to use the same program to read "input" from a file instead of from a live user.

I hope you can see that part of the answer will be the requirement that the live user has to enter the EOF (end-of-file) character to terminate the process. Remember that EOF is generated by CTRL-D on Mac OSX, Unix and Linux systems.

On a PC, the EOF character may require CTRL-Z instead.

READ: Reading Character Input

Remember how we handled reading an unknown amount of input from the user:

- We read the data in a loop;
- The user as many items as desired;
- The user entered CTRL-D to terminate;
- The function **cin.eof()** returns **true** on end of file.

It will be convenient to use a **while (condition)** loop, but we want to check for end-of-file in the middle of the loop. We can simply use a **break** there. So the **while()** has nothing to check, and should just loop forever. We can make that happen by putting the word **true** as the condition of the **while()**. You have neve seen this before, so prepare to be surprised a little.

READ: Program Outline

character_count.cpp:

```
char c;
int n = 0;             <-- Count the characters we read.
while ( true )         <-- Loop forever!
{
  cin >> c;
  if ( cin.eof ( ) )   <-- Break at end-of-file!
  {
    break;
  }
  n = n + 1;           <-- We've read another character.
}
cout << "Number of characters was " << n << "\n";
```

READ: Compare with WC

Remember that the **wc** command can be used to count the number of lines, words and characters in a file. Let's compare our program against **wc**, by measuring the length of the Gettysburg address!

```
wc gettysburg.txt

      24      278     1469 gettysburg.txt
g++ character_count.cpp
mv a.out character_count
./character_count < gettysburg.txt

Your input consisted of 1189 characters.
```

READ: The Discrepancy is “White Space”

The **wc** program found more characters than our program. That’s because the **cin** function, by default, doesn’t “see” blanks and new lines and tabs and other separator characters, sometimes called **white space**.

Maybe that’s what we actually want. On the other hand, we may need to know the exact number of characters, including every time we hit the space bar or the return key.

If that’s what we want, we simply replace our friendly **cin** $>>$ **c** call by the slightly less friendly, but more powerful, c = cin.get() function, which returns <u>every</u> character the user types, including spaces, tabs, and so on.

READ: Count ALL the Characters

character_count2.cpp:

```
char c;
int n = 0;             <-- Count the characters we read.
while ( true )         <-- Loop forever!
{
  c = cin.get ( );     <-- Also read control characters!
  if ( cin.eof ( ) )   <-- Break at end-of-file!
  {
    break;
  }
  n = n + 1;           <-- We've read another character.
}
cout << "Number of characters was " << n << "\n";
```

READ: Compare with WC

```
  wc gettysburg.txt

        24      278     1469 gettysburg.txt
--------------------------------------------------
  g++ character_count.cpp
  mv a.out character_count
  ./character_count < gettysburg.txt

  Your input consisted of 1189 characters.
--------------------------------------------------
  g++ character_count2.cpp
  mv a.out character_count2
  ./character_count2 < gettysburg.txt

  Your input consisted of 1469 characters.
```

READ: Longest Word

Suppose we wanted to know the length of the longest word in a file? How would we go about this?

Think of a word as an uninterrupted sequence of characters.

Imagine I encounter a character for the first time. I count it's length as 1. I read another character. If it is also alphabetic, I increase length by 1 again. I continue until I reach a nonalphabetic character.

At that point, I have the length of this word. So I compare the length of this word to the maximum, and update that if necessa

READ: Longest Word

longest_word.cpp:

```
int length = 0;
int length_max = 0;

while ( true )
{
  c = cin.get ( );

  if ( cin.eof ( ) )
  {
    break;
  }
  if ( ( 'A' <= c && c <= 'Z' ) || ( 'a' <= c && c <= 'z' ) )
  {
    length = length + 1;
  }
  else
  {
    if ( length_max < length )
    {
      length_max = length;
    }
    length = 0;
  }
}
cout << "Longest word has length " << length_max << ".\n";
```

Characters and the SWITCH Statement

- Introduction
- The CONTINUE Statement
- Character Data
- Reading Characters
- **The SWITCH Statement**
- ETAION SHRDLU
- Lab Exercise #5

SWITCH: Action Based on Variable Value

The **switch** statement can be thought of as a special kind of **if/else** statement, which allows a user to quickly define a choice of different actions based on the possible values of a variable.

The format of the **switch** statement is unusual, and it is not that commonly used. However, there are some kinds of problems where it can be the best solution.

We will see that, typically, each set of actions terminates with a **break** statement. This is an important part of the structure! We will look at what happens otherwise.

SWITCH: Format of the Statement

The **switch** statement has the following form:

```
switch ( variable )  <-- variable to check
{
  case value 1:          <-- if variable has this value
    actions;                  then do these things.
    break;
  case value 2:          <-- and so on...
    actions;
    break;
  case value 3:
    actions;
    break;
  (...perhaps more cases...)
  default:             <-- if no matches for variable value
    actions;                then do this.
    break;
}
```

SWITCH: Action Based on Variable Value

grade.cpp: Assume the **char** variable **grade** contains a letter grade. Print the corresponding range of averages.

```
switch ( grade )  <-- variable to check
{
  case 'A':
    cout << "Range for A is 90 to 100.\n";
    break;
  case 'B':
    cout << "Range for B is 80 to 89.\n";
    break;
  case 'C':
    cout << "Range for C is 70 to 79.\n";
    break;
  case 'D':
    cout << "Range for D is 60 to 69.\n";
    break;
  case 'F':
    cout << "Range for F is 0 to 59.\n";
    break;
  default:
    cout << grade << " is not a legal grade.\n";
    break;
}
```

SWITCH: Action Based on Variable Value

We can't give a range of values, only a specific value. So we can not use switch to look at the value of the average, and return the letter grade

This is not how switch works!

```
switch ( average )  <-- variable to check
{
  case 90 to 100:       <--No!  You cannot use a range!
    cout << "Grade is A.\n";
    break;
  case 80 to 89:
    cout << "Grade is B.\n";
    break;
    ...
}
```

SWITCH: Action Based on Variable Value

However, if we know that the average was an **int**, then we are allowed to list several cases together, to get the same action.

```
switch ( average )  <-- variable to check
{
  case 100:
  case 99:
  case 98:
  case 97:
  case 96:
  case 95:
  case 94:
  case 93:
  case 92:
  case 91:
  case 90:
    cout << "Grade is A.\n";
    break;
  case 89:
  case 88:
    ...and so on...
    cout << "Grade is B.\n";
    break;
  case 79:
  case 78:
    ...and so on...
}
```

Here, a set of **if/else if** statements would be better!

SWITCH: Action Based on Variable Value

grade.cpp: Returning to our grade example, we can accept upper or lowercase grades this way:

```
switch ( grade )  <-- variable to check
{
  case 'A':
  case 'a':
    cout << "Range for A is 90 to 100.\n";
    break;
  case 'B':
  case 'b':
    cout << "Range for B is 80 to 89.\n";
    break;
  case 'C':
  case 'c'
    cout << "Range for C is 70 to 79.\n";
    break;
  case 'D':
  case 'd'
    cout << "Range for D is 60 to 69.\n";
    break;
  case 'F':
  case 'f'
    cout << "Range for F is 0 to 59.\n";
    break;
  default:
    cout << grade << " is not a legal grade.\n";
    break;
}
```

Characters and the SWITCH Statement

ETAOIN: Letter Frequency

It is known that the letter 'E' tends to occur most frequently in English text. At one time, it was thought that the correct listing for English letters, by frequency, was **ETAOIN SHRDLU**.

More careful and complete counting suggests the full order might be **ETAOIN SRHDLU CMFYWG PBVKXQ JZ**

Now let us suppose that we wanted to examine a piece of text, and count the frequency of occurrence of some of the more common letters.

We might start, simply, with **'E'** and **'T'**.

ETAOIN: Program Design

We can start with the **character_count.cpp** program, but once we have read a character **c**, we want to analyze it before getting the next one.

In particular, in case **c** is an **'E'**, we want to increment the variable **ecount** and if it is a **'T'**, we increment **tcount**.

It is easy to do this with a **switch** statement, and that reminds us that we should also check for **'e'** and **'t'** at the same time!

Once we're done reading, we print the values of **ecount** and **tcount**.

ETAOIN: A Program to Count 'E' and 'T'

frequency.cpp:

```
char c;
int ecount = 0, tcount = 0;

while ( true )
{
  cin >> c;
  if ( cin.eof ( ) )
  {
    break;
  }
  switch ( c )
  {
    case 'E':
    case 'e':
      ecount++;  <-- I got lazy and used the abbreviation!
      break;
    case 'T':
    case 't':
      tcount++;
      break;
    default:
      break;
  }
}
cout << " 1 " << ecount << " E/e\n";  <-- If I write the output this way
cout << " 2 " << tcount << " T/t\n";      I can plot it later with gnuplot.
```

Characters and the SWITCH Statement

EXERCISE: Letter Frequency

Get a copy of **frequency.cpp**, which computes the frequency of the letters 'E' and 'T' in a file, and **gettysburg.txt**.

Modify the program so that both 'E' and 'e' will increase **ecount** and both 'T' and 't' will increase **tcount**.

Compile your program, rename it to **frequency**, and then run it, reading the input from the file:

```
./frequency < gettysburg.txt
```

You should see 165 occurrences of 'E'/'e' and 126 of 'T'/'t'.

Modify the program to check for 'A', 'I', 'O' and 'N' (upper and lowercase) as well. You have to add new variables, new cases to the **switch** statement, and new printout lines at the end.

Once you get your program set up to check all six letters, run it and save the output:

```
./frequency < gettysburg.txt > count.txt
```

EXERCISE: Letter Frequency

If you updated your output statements correctly, your output file **count.txt** should have the form:

```
1 165  E/e
2 126  T/t
3 ???  A/a
4 ???  I/i
5 ???  O/o
6 ???  N/n
```

where the ??? will be replaced by your results.

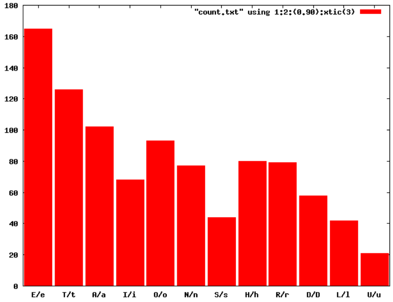
Please show your final table of results to Detelina so you can
credit for the exercise!

EXERCISE: GNUPLOT Can Plot Your Data

```
gnuplot
  plot "count.txt" using 1:2 with boxes
```

...or, for better results...

```
  set yrange [0:180]
  set style fill solid
  plot "count.txt" using 1:2:(0.90) with boxes
```


"count.txt" using 1:2:(0.90):xtic(3)
E/e
T/t
A/a
I/i
O/o
N/n
S/s
H/h
R/r
D/D
L/l
U/u