CS3520 Programming in C++
Control Structures

Ken Baclawski
Fall 2016

Outline

- Review twice program
  - DOS files
- Review frame program
  - wide characters
- Review avg program
  - variables
  - statement blocks
  - string class
  - control constructs
- The graphs program
  - for loop
  - loop invariant
  - if statement
  - compound assignment operators
- Assignment #2
- Safety and robustness

© 2016 Ken Baclawski All Rights Reserved

Review twice program

© 2016 Ken Baclawski All Rights Reserved

DOS Input File

- Try running it with a DOS file:

```
./main < dos.txt

!

!
```

- The CR character has erased the input line.

```
int main() {

  // Read the input line.

  std::string line;
  std::getline(std::cin, line);

  // Concatenate with an exclamation
  // point.

  std::string exclamation = line + "!";

  // Print twice.

  std::cout << exclamation << std::endl;
  std::cout << exclamation << std::endl;

  // Return normal status.

  return 0;
}
```


© 2016 Ken Baclawski All Rights Reserved

Review of frame program

© 2016 Ken Baclawski All Rights Reserved

The frame program

```
int main() {

  // Ask for a name on one line

  cout << "Please enter your name: ";
  string name;
  getline(cin, name);

  // Build the message and its line

  const string message("Welcome, " +
    name + "!");
  const string messageLine("* " +
    message + " *");

  // Build the first (and last)
  // lines of the frame

  const string first(messageLine.size(),
                     '*');

  // Build the second (and second last)
  // lines of the frame

  const string second("* " +
    string(message.size(), ' ') + " *");

  // Write the frame

  cout << endl;
  cout << first << endl;
  cout << second << endl;
  cout << messageLine << endl;
  cout << second << endl;
  cout << first << endl;

  // Return normal status code.

  return 0;
}
```


© 2016 Ken Baclawski All Rights Reserved

Wide String Input

- Try running it with a DOS file:

```
./main < wide.txt
**********************
*                    *
* Welcome, 张伟！ *
*                    *
**********************
```

- C++ uses `char` for the type of a byte. It is **not** a character!
- The wide characters use more than one byte each.
  - In fact, each one in the name above uses three bytes.
  - In the font used above, each wide character uses two positions in the output.

© 2016 Ken Baclawski All Rights Reserved

Review of the avg program

© 2016 Ken Baclawski All Rights Reserved

Variables

- Variables
  - Have a type and a value
  - Memory is allocated to hold the value
- Built-in (primitive) types
  - `int`, `double`, `char`, `bool`, pointer
  - *Must* always be initialized
  - Using an uninitialized variable is dangerous
- Class types
  - string, user-defined classes
  - Will always be initialized

© 2016 Ken Baclawski All Rights Reserved

Statement blocks

- Enclosed in braces as in many languages
- Consists of a sequence of statements
  - Can have just one or no statements at all
- A block is a scope
  - Variables declared in a block are only visible and usable in the block and after they are declared
  - At the end of a block, the memory allocated to the value of a variable is deallocated.
- Variables *must* be declared as late as possible in the scope
- The same variable name *must* never be used for two different purposes in the same file.
- Every variable *must* be in a block: Never declare a global variable.

© 2016 Ken Baclawski All Rights Reserved

The `string` class

- A sequence of bytes
- Can be initialized with a string literal
- If not explicitly initialized, then initialized to empty string
- The sequence can be changed
  - Not like Java `String` which is immutable
  - Closer to Java `StringBuilder`
- Note that the type of a byte is `char`.

© 2016 Ken Baclawski All Rights Reserved

Control constructs

- The control constructs are similar to other languages
  - `if/else` and `switch` are used for conditional execution
  - `for` and `while` are used for looping
- The coding style requirements require:
  - A body of a control construct *must* always be a block
  - Points will be deducted if this is not done
  - IDEs make it very easy to conform to this requirement
- A conditional *must* not be an assignment statement
- *Never* use a `do`, `goto` or comma statement

© 2016 Ken Baclawski All Rights Reserved

The `if` statement

- The `if` statement evaluates a *conditional*
  - The body of the `if` statement is executed if the conditional evaluates to true
- An `if` statement can be followed by an `else` statement
  - The body of the `else` statement is executed if the conditional evaluates to false
- A succession of `if` statements can be used to test a series of conditionals.
- Style requirement: An `if` or `else` statement body must be enclosed in braces.

© 2016 Ken Baclawski All Rights Reserved

The `while` loop

- The loop body is executed as long as a conditional is true
- When the loop completes one of the following must have occurred:
  - The conditional is false or
  - A `break` statement was executed
- Style requirement: A `while` statement body must be enclosed in braces.
- A `do` statement is similar to a `while` statement but with the conditional at the end of the loop body.
  - A `do` statement is less readable because the condition can be far from the start of the expression.
  - Other loop constructs are more readable as well as much more flexible.
  - Avoid using the `do` statement.

© 2016 Ken Baclawski All Rights Reserved

Conditionals

- The C language originally did not have a boolean type
- Many types were allowed in a C conditional
- C++ is more restrictive but still allows some C-style conditionals
- Coding style requirements:
  - *Never* use a numeric expression in a conditional. Explicitly compare with 0.
  - *Never* use an assignment statement in a conditional.

© 2016 Ken Baclawski All Rights Reserved

Standard I/O Streams

- The standard input/output streams
  - `cin` standard input
  - `cout` standard output
  - `cerr` standard error output
- The operator >> reads into a variable (space-delimited)
- The operator << writes a variable (buffered write)
- A stream in a boolean context (such as an `if` or `while` condition) is converted to boolean and tests whether the stream can still be used.
- Examples of ***operator overloading***
- `endl` is a stream manipulator: write a newline and flush the buffer

© 2016 Ken Baclawski All Rights Reserved

Compound Assignment

- The operator += performs an operation and assignment
  - The statement `x += y;` is the same as `x = x + y;`
- Most binary operators have compound versions

© 2016 Ken Baclawski All Rights Reserved

The graphs program

© 2016 Ken Baclawski All Rights Reserved

A simple graph drawing program

- Draw the graphs of two functions:
  - The sum of the integers and
  - The sum of the squares of integers
- The letter `a` denotes the value of the sum of the integers
- The letter `b` denotes the value of the sum of the squares of the integers.
- If the values are the same, then an asterisk is drawn.
- Show values starting at a sum of 0 terms
- Show no more than 50 values
- Stop showing graph lines when a line has more than 80 characters.

© 2016 Ken Baclawski All Rights Reserved

The `for` loop statement

- Designed for the most common loop tasks
- The `for` loop combines:
  - Initialization of a *loop control variable*
  - A `while` conditional
  - Update of the loop control variable
- Usually the loop control variable is declared as well as initialized
  - The scope of the loop control variable is the `for` loop.

```
int sum1 = 0;
int sum2 = 0;

for (int x = 0; x < 50; ++x) {

  // Compute the two sums

  sum1 += x;
  sum2 += x * x;

  // ...
}

// The variable x is no longer
// in scope from this point on.
```


© 2016 Ken Baclawski All Rights Reserved

The `for` loop statement

- Designed for the most common loop tasks
- The `for` loop combines:
  - Initialization of a *loop control variable*
  - A `while` conditional
  - Update of the loop control variable
- Usually the loop control variable is declared as well as initialized
  - The scope of the loop control variable is the `for` loop.

```
int sum1 = 0;
int sum2 = 0;

for (int x = 0; x < 50; ++x) {

  // Compute the two sums

  sum1 += x;
  sum2 += x * x;

  // ...
}

// The variable x is no longer
// in scope from this point on.
```


© 2016 Ken Baclawski All Rights Reserved

Increment and decrement operators

- The increment and decrement operators increase or decrease a numeric variable by `1`
- The prefix version (`++x`) changes the variable and then returns the *new* value
- The postfix version (`x++`) changes the variable and then returns the *old* value
- The prefix version is slightly more efficient
- One can overload these operators

```
int sum1 = 0;
int sum2 = 0;

for (int x = 0; x < 50; ++x) {

  // Compute the two sums

  sum1 += x;
  sum2 += x * x;

  // ...
}

// The variable x is no longer
// in scope from this point on.
```


© 2016 Ken Baclawski All Rights Reserved

Statements and Expressions

- A *statement* is code that does something.
  - A statement can be empty. A semicolon by itself is a statement
  - A statement block groups a sequence of statements in braces. The statements are executed in order.
  - A statement block can be empty: `{ }`
- An *expression* is a sequence of one or more operators and operands that specifies a computation
  - The operands of an operator might not be evaluated in order!
  - If `x` is a variable, then `x` by itself is an expression
  - If `x` and `y` are numeric variables, then `x + y` is an expression
- Any expression can be made into a statement
  - If `x` is a variable, then `x;` is a statement
  - If `x` and `y` are numeric variables, then `x + y;` is a statement

© 2016 Ken Baclawski All Rights Reserved

Comma Operator

- Sequence of expressions evaluated in the specified order
- Value is that of the last expression
- Seldom used
- Easily confused because a comma looks similar to a semicolon
- Also confusing because the comma has many other uses
- Easily replaced with a sequence of ordinary statements
- Mainly used in `for` statements

```
int x = 5;
int y = 3;
int z = x++, x – y,
  2 * x – 3 * y;
```

Style Requirement:
Never use the comma operator

© 2016 Ken Baclawski All Rights Reserved

Precedence and Associativity

- Each operator has a precedence and associativity
- *Precedence* determines which operations are performed first when there are operators with different precedence in an expression
- *Associativity* determines which operations are performed first when there are operators with the same precedence in an expression
- See C++ Operator Precedence and Associativity on the next slide
- Avoid depending on precedence and associativity in complicated expressions: use parentheses liberally

© 2016 Ken Baclawski All Rights Reserved

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<` `<=`	For relational operators < and ≤ respectively	
	`>` `>=`	For relational operators > and ≥ respectively	
9	`==` `!=`	For relational operators = and ≠ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
16	`,`	Comma	Left-to-right

Requirements and Recommendations

- The `for` loop body *must* be enclosed in braces
- There should always be exactly one loop control variable
- One should not declare or initialize variables other than the loop control variable
  - Declare and initialize other variables outside the loop or in the loop body.
- One sometimes sees multiple variables initialized using the comma operator in a `for` loop header. This is confusing and should be avoided.

© 2016 Ken Baclawski All Rights Reserved

Loop Invariants

- A *loop invariant* is a logical expression (predicate) that is true after each iteration of a loop
  - The loop invariant can be false at other times in the body of the loop
  - Sometimes the loop invariant is also required to be true before the start of the loop
    - This only affects whether the loop invariant is true when the for loop starts
  - Handy tool for ensuring that the loop is correct

© 2016 Ken Baclawski All Rights Reserved

Loop Invariant Example

- The for loop in the graphs program satisfies the following loop invariant:

  0 ≤ x < 50 and

  sum1 is the sum 0 + 1 + ... + x and

  sum2 is the sum 0^2 + 1^2 + ... + x^2 and

  graphLine has the character a or * in position sum1 and

  graphLine has the character b or * in position sum2 and

  graphLine has the character * only if sum1 is the same as sum2 and

  graphLine has no more than 80 characters

- The first three are also true at the start of each loop iteration

© 2016 Ken Baclawski All Rights Reserved

Loop Invariant Example

- The loop invariant can be used to prove that the value of sum1 and sum2 is correct
  - This uses mathematical induction
- The loop invariant can also be used to prove that the printed graph is correct
  - This does not use mathematical induction
- Proofs are not required on your assignments or exams

© 2016 Ken Baclawski All Rights Reserved

Loop Invariant Example

- The loop invariant can be used to prove that the value of sum1 and sum2 is correct
  - This uses mathematical induction
- The loop invariant can also be used to prove that the printed graph is correct
  - This does not use mathematical induction
- Proofs are not required on your assignments or exams

© 2016 Ken Baclawski All Rights Reserved

Constructing the graph line

- Use a *nested* if statements to deal with different cases
- The last `if` conditional could have been omitted
- The `graphLine` variable was declared just before its first use.
- The `break` statement terminates the loop
  - It ensures that the loop invariant is true

```
string graphLine("|");
if (sum1 < sum2) {
  graphLine += string(sum1, ' ') + "a";
  graphLine +=
    string(sum2 - sum1 - 1, ' ') + "b";

} else if (sum1 == sum2) {
  graphLine += string(sum1, ' ') + "*";

} else if (sum1 > sum2) {
  graphLine += string(sum2, ' ') + "b";
  graphLine +=
    string(sum1 - sum2 - 1, ' ') + "a";
}

if (graphLine.length() > 80) {
  break;
}

cout << graphLine << endl;
```


© 2016 Ken Baclawski All Rights Reserved

Variations on the wordrev program

© 2016 Ken Baclawski All Rights Reserved

The wordrev requirements

- Read words from the standard input and reverse them.
- The words must consist of ASCII characters.
- The program completes when either the input stream ends or the word “quit” is entered.
- Several variations are discussed
- In later variations, the status returned by the program is:

  0 if the word “quit” is entered

  1 if the input ends without “quit” being entered

© 2016 Ken Baclawski All Rights Reserved

First variation

- The `&&` operator is the *short-circuit* logical and
  - The operands are evaluated in order
  - If the first operand evaluates to false, then later operands are not evaluated
- Caution: If the `&&` operator is overloaded it is not a short-circuit operator.
- The `at` method of the `string` class gets the byte at a specified position.

```
string word;
while (cin >> word &&
       word != "quit") {

  for (int i = word.length()-1;
       i >= 0; --i) {
    cout << word.at(i);
  }

  cout << endl;

}
```


© 2016 Ken Baclawski All Rights Reserved

Second variation

- The `[ ]` operator is the same as the `at` method except for what happens if the index is out of range:
  - The `at` operator will throw an exception which can be caught
  - The behavior of the `[ ]` operator in this case is undefined
- The `at` operator is safer

```
string word;
while (cin >> word &&
       word != "quit") {

  for (int i = word.length()-1;
       i >= 0; --i) {
    cout << word[i];
  }

  cout << endl;

}
```


© 2016 Ken Baclawski All Rights Reserved

Third variation

- Checking for “quit” within the loop body is more readable and versatile
  - It does not depend on the short-circuit property
  - It can distinguish “quit” from the end of input

```
string word;
while (cin >> word) {
  if (word == "quit") {
    return 0;
  }

  for (int i = word.length()-1;
       i >= 0; --i) {
    cout << word.at(i);
  }

  cout << endl;

}
```


© 2016 Ken Baclawski All Rights Reserved

Fourth variation

- Construct the reversed word and then print it.
- One must declare `i` to be `unsigned` in order to compare it with `word.length()`
- This variation will throw an exception. Can you see why?

```
string word;
while (cin >> word) {
  if (word == "quit") {
    return 0;
  }

  string rWord;
  for (unsigned int i = 0;
       i < word.length();
       ++i) {
    rWord.at(i) =
      word.at(word.length()-i-1);
  }
  cout << rWord << end;
}
```


© 2016 Ken Baclawski All Rights Reserved

Fourth variation corrected

- The problem with the last variation is that `rWord` was initialized to an empty word
  - One cannot assign to a byte if there is no byte that can be assigned
- The `resize` method changes the length of the word, initializing added bytes with 0 bytes.

```
string word;
while (cin >> word) {
  if (word == "quit") {
    return 0;
  }

  string rWord;
  rWord.resize(word.length());
  for (unsigned int i = 0;
       i < word.length();
       ++i) {
    rWord.at(i) =
      word.at(word.length()-i-1);
  }
  cout << rWord << end;
}
```


© 2016 Ken Baclawski All Rights Reserved

Fifth variation

- A `stringstream` is a `string` that can be used like a stream
- To use it, one must include the <sstream> library
- A `stringstream` is dynamically increased in size when a new byte is added.
- The `str` method extracts the current string object

```
string word;
while (cin >> word) {
  if (word == "quit") {
    return 0;
  }

  stringstream rWord;
  for (unsigned int i = 0;
       i < word.length();
       ++i) {
    rWord <<
      word.at(word.length()-i-1);
  }
  cout << rWord.str() << end;
}
```


© 2016 Ken Baclawski All Rights Reserved

Assignment #2

© 2016 Ken Baclawski All Rights Reserved

Assignment #2

- Read words from the standard input.
- Print the longest word and the shortest word in the input.
- If there are no words in the input print, then nothing and return status 1.
- If more than one word has the same length as the shortest word, then print the first one.
- Similarly for the longest word.

© 2016 Ken Baclawski All Rights Reserved

Assignment #2 Example Input

```
The quick brown fox jumped over the lazy dog
```


© 2016 Ken Baclawski All Rights Reserved

Assignment #2 Example Output

```
Shortest word is The
Longest word is jumped
```


© 2016 Ken Baclawski All Rights Reserved

Assignment #2 Submitting and Grading

- All code is in the main function
- Submit `Main.cpp`
- No command-line arguments
- Grading:
  - Compile with no errors or warnings (20%)
  - Correct execution on test data (30%)
  - Documentation (20%)
  - Correct style (30%)

© 2016 Ken Baclawski All Rights Reserved

Safety and Robustness

© 2016 Ken Baclawski All Rights Reserved

Safety and Robustness

- C++ inherits many dangerous operations from C
  - The advantage is the close integration with C
  - The disadvantage is that C++ is not a “safe” language
- Unlike languages like Java, the C++ compiler is not completely specified.
  - Some operations are not defined in the C++ standard
  - A compiler may assume that these operations never happen
  - The programmer has the burden of ensuring their program is meaningful, not the compiler.

© 2016 Ken Baclawski All Rights Reserved

Problematic Behavior

- *Undefined* behavior
  - Compiler may assume that it never happens.
  - This is the most dangerous
  - Anything could happen. Your program could reformat your hard drive!
- *Unspecified* behavior
  - Compiler must allow it to happen but need not document what will happen.
  - Almost as dangerous, since you don't know what will happen
- *Implementation-dependent* behavior
  - Compiler must allow it to happen and must document what will happen.
  - This is not as bad, but should still be avoided as it makes your program unportable.

© 2016 Ken Baclawski All Rights Reserved

Examples of Undefined Behavior

- Integer divide by 0
- Using an uninitialized variable
  - Only possible for built-in types
- Modifying a string literal
- Integer overflow
- Using the index operator with an out of range index
- Many others
- This course emphasizes writing code that is safe and robust
  - This is one of the purposes of the coding style requirements

© 2016 Ken Baclawski All Rights Reserved