CSI 201 - Introduction to Computer Science

Chapter 2
C++ Basics

Brian R. King
Instructor

Goals

Introduce basic C++ constructs and concepts including:

- Variables and Assignments
- Simple Input and Output
- Basic Formatting of Output
- Simple Data Types
- Flow Control, including introducing boolean expressions
- Common terminology used in C++ and programming in general.
- Program Style

Programming Language Rules

- Languages need rules in order to be properly interpreted.
  - Written English has rules that must be followed in order to be properly interpreted.
  - Periods, commas, capital letters, etc. all determine how a sentence is interpreted.
- Programming languages also have rules that must be followed for the program to be compiled properly. This is referred to as the **syntax** of the language.
  - A program with legal syntax can be read by the compiler, which will then generate executable code to be run on the computer.
- The **syntax** for a programming language is the set of grammar rules for the language.

Statements

- A **statement** in C++ is the basic unit for building C++ programs.
  - A statement performs an action.
  - A complete program will be a series of C++ statements that direct the computer to follow the statements in your code.
- Statements end with a **semi-colon**
  - Statements do NOT end with the end of a line.
- Example statements we saw in Chapter 1:
  - Variable declarations:
    - `int first_num;`
  - Assignment statements:
    - `result = first_num + second_num;`
  - Output statements
    - `cout << "Your answer is: " << result;`
  - There are many more to be seen!
- A **block-statement** (or sometimes called **statement block**) is a grouping of one or more statements delimited by the curly braces { }
  - The body of the main program in a C++ program is a block statement.

Expressions

- An **expression** is a grouping of variables, constants and/or operators that specifies a computation.
- **An expression results in a value**.
- Examples:
  - **Constant** expression
    - 300
  - **Variable** expression
    - `first_num`
  - **Arithmetic** expression
    - `first_num + second_num`
  - Many more to come!
- An expression is NOT a statement!
  - Statements end in a semicolon
  - A statement may include several expressions.
- Remember:
  - A **statement** performs an action
  - An **expression** produces a value

Variables

- Think of high school algebra. Remember the quadratic formula?
- What are x, a, b and c? **Variables**!
- A variable in algebra is just a place holder where you can substitute different numbers in place of variables on the right hand side of the equation to generate a new value for the variable on the left hand side.
- In programming, a variable gives us a way to tie a name to a place in memory that can store data. (Otherwise, we would have to use addresses to access memory locations!)
- With standard variables:
  - You can read the value in it.
  - You can write a value to it.
  - You can change the value of it.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Declaring Variables in C++

- A variable must be **declared** before it can be used.
- A **declaration** of a variable provides:
  - The **type** of data that will be stored
  - The name of the variable, called an **identifier**
- **Types**
  - Types are IMPORTANT! How a value is stored in memory depends on its type.
    - **`int`** – Integer; Whole numbers (e.g. -1, 10, 255)
    - **`double`** – Floating point number that can hold fractional parts (e.g. 40.345, -10.5, 100.1)
  - More complete list of types discussed later.
- **Identifiers**
  - Choose meaningful names that represent the data to be stored.
  - Variable names have restrictions:
    - First character must be:
      - a letter
      - an underscore character
    - Remaining characters must be:
      - letters
      - numbers
      - an underscore character
    - Can not be a **keyword,** a reserved word that is used in the C++ language (e.g. `if`, `else`, `do`, `using`, …)

Examples of identifiers

- Valid Examples
  - `x`
  - `x_1`
  - `_abc`
  - `A2b`
  - `ThisIsAVeryLongIdentifier`
- Invalid Examples
  - `12`
  - `3X`
  - `%change`
  - `myFirst.c`
  - `data-1`
  - `if`

Variable Declaration Syntax

- The C++ syntax for a declaration is as follows:
  - *type_name variable_name1, variable_name2, … ;*
- Example variable declarations:
  - `int hours_worked;`
  - `double hourly_wage, salary;`

More on Declaring Variables

Two locations for variable declarations

- Immediately prior to use

```
int main()
{
    …
    // Other code
    …
    int sum;
    sum = score1 + score 2;
    …
    return 0;
}
```

- At the beginning.

```
int main()
{
    int sum;
    …
    sum = score1 + score2;
          …
    return 0;
}
```

- Both are acceptable forms of declaration. Do whatever will make your program easier to understand.
- Not sure which to use? If there are only a few statements that use a particular variable, declare the variable prior to use. If a variable is used throughout your program, then declare it at the beginning.
- (Industry standards tend to declare variables at the beginning.)

What happens during a declaration?

- The compiler generates code to set aside memory to store the variable.
  - It associates the name of the variable with the address of the memory used for the variable for you. You need not worry about where the variable resides.
- Giving the variable a type allows your program to interpret the 0's and 1's that reside in the memory set aside for the variable.
- Each type has different memory requirements.
  - `int`
    - Requires 4 bytes of memory
    - Can store all values in the range -2147483647 to 2147483647
  - `double`
    - Requires 8 bytes of memory
    - Can store floating-point values in the range of 10^{-308} to 10^{308} with up to 15 digits of precision. (More on precision later.)
  - Other types to be discussed later.
- IMPORTANT -- After a variable has been declared and memory has been set aside, what value does the variable contain?
  - GRABAGE! Why? Because the variable has not been initialized! (It has not been assigned a value.)

Assignment Statements

- An assignment statement changes the value of a variable
  - `total_weight = one_weight + number_of_bars;`
    - total_weight is set to the sum one_weight + number_of_bars
- Formal syntax:
  - *Variable = Expression;*
- The single variable to be changed is <u>always on the left hand side</u> of the assignment operator '='
  - `one_weight + number_of_bars = total_weight;` is illegal!
  - The left hand side of an assignment must always be an **addressable** expression, sometimes called an lvalue in computer science (which simply stands for the value on the left hand side of an assignment.) Variables are the only lvalues we talk about until the end of the course.
- On the right of the assignment operator is an **expression**
- Valid expressions for the right side of an assignment statement can be:
  - **Constants**
    - `age = 21;`
  - **Variables**
    - `my_cost = your_cost;`
  - or an **arithmetic expression**
    - `circumference = diameter * 3.14159;`

ex1.cpp

```
// ex1.cpp
// Simple program to illustrate some introductory concepts in C++

#include <iostream>
using namespace std;

int main( )
{
  int hours_worked;
  double hourly_wage, week_salary;

  // Input data
  cout << "Enter the number of hours worked:\n";
  cin >> hours_worked;
  cout << "Enter your hourly wage:\n";
  cin >> hourly_wage;

  // Do calculation
  week_salary = hours_worked * hourly_wage;

  // Print output
  cout << "You worked " << hours_worked << " hours.\n";
  cout << "Your hourly wage is $" << hourly_wage << ".\n";
  cout << "You earned $" << week_salary << " this week.\n";

  return 0;
}
```


Initializing a variable

- A common mistake in C++ is using a variable without initializing it to a known value.
- Remember -- a variable has garbage in it after it's declared unless it's been assigned a value.
- How can we initialize a variable to a known value as part of the declaration?
  - Method 1

```
double mpg = 26.3, area = 0.0, volume;
```

  - Method 2

```
double mpg(26.3),  area(0.0), volume;
```

  - Method 1 is the preferred method for variables. (Method 2 is usually reserved for objects, to be covered in Chapter 6.)

Input and Output through streams

- C++ uses **streams** to handle input and output to and from your program
  - A stream in C++ is a sequence of data.
  - An **input stream** handles data being fed into your program.
    - Keyboard
    - File
  - An **output stream** handles data being generated by your program.
    - Monitor / Screen
    - File

Input

- `cin` is an input stream bringing data from the keyboard.
- It is made available to your program because of these lines in your program:

```
#include <iostream>
using namespace std;
```

- The **extraction operator** (`>>`) removes (or extracts) data to be used from the input stream and places it in the specified variable.
- Example:

```
cout << "Enter the number of hours worked\n";
cout << "Then enter the hourly wage\n";
cin >> hours_worked >> hourly_wage;
```

  - This code above prompts the user to enter data and then reads data from cin.
  - The first value read is stored in `hours_worked`. **Must be an integer!**
  - The second value read is stored in `hourly_wage`
  - Multiple variables extracted must be separated by **whitespace** – space, tab, newline (\n).
- Data is not sent to your program until the ENTER (or return) key is pressed.
  - `40` <SPACE> `12.50` <ENTER>
  - `40` <ENTER> `12.50` <ENTER>

  Both will work.

Input Extraction

- When extracting data for a variable, the following rules apply:
  - Leading whitespace characters are skipped.
  - Characters are extracted from the input stream one-by-one, as long as they satisfy the requirements of the input type.
    - integral types – leading sign, then one or more numbers
    - floating point types – leading sign, one or more numbers, a decimal point, then one or more numbers representing the fractional part
- Extraction *stops* on one of the following conditions:
  - Another whitespace character occurs in the input stream.
  - A character that can not satisfy the type being extracted occurs in the input stream. The character is left in the input stream.

Output

- `cout` is an output stream connected to the screen (or monitor).
- As with `cin`, it is made available to your program because of these lines in your program:

```
#include <iostream>
using namespace std;
```

- The **insertion operator** (`<<`) inserts data into the specified output stream.
- Example:

```
week_salary = hours_worked * hourly_wage;
cout << "You earned $" << week_salary << " this
week\n";
```

- This code assigns a value to `week_salary` in the first statement, then inserts the value formatted in a sentence to the monitor.
  - Notice the space before the 't' in " this week\n". Why?
  - The '\n' causes a new line to be started following the 'k' in week.
  - A new insertion operator is needed for each item of output.

Output (continued)

- Instead of:

```
cout << "You earned $" << week_salary << " this week\n";
```

- We could have used three separate C++ statements:

```
cout << "You earned $";
cout << week_salary;
cout << " this week\n";
```

- Or one C++ statement, with three expressions on different lines:

```
cout << "You earned $"
     << week_salary
     << " this week\n";
```

- We could have embedded an arithmetic expression instead:

```
cout << "You earned $"
     << hours_worked * hourly_wage
     << " this week\n";
```

- Why is the last example possible?
  - Each operand to the right of an insertion operator is an **expression**.
  - Remember – each expression produces a result
  - Each expression is evaluated before it is inserted into the output stream

Valid output expressions

- Constants
  - `cout << 21.5;`
- Variables
  - `cout << week_salary;`
- Literal strings
  - A **string** is a sequence of characters. A literal string, also called a **string constant**, is a string enclosed in double quotation marks. More on strings later in the course.
  - `cout << "CSI 201 is too early!\n";`
- Arithmetic expressions
  - `cout << (hours_worked * hourly_wage);`
- Many more to come…

Proper I/O Design

- Prompt the user for input that is desired
  - `cout << "Enter your age: ";`
    `cin >> age;`
  - Don't expect the user to know what your program is asking for.
- Echo the input by displaying what was read
  - Gives the user a chance to verify data
  - `cout << age << " was entered." << endl;`

Escape sequences

- **Escape sequences** tell the compiler to treat characters in a special way
- '\' is the escape character
- So far, we've seen one escape sequence, the newline character: \n.
  - `cout << "Hello, World!\n";`
- Common Escape Sequences
  - \n Newline
  - \t Tab
  - \a Alert (or Bell)
  - \\ Backslash
  - \" Quotation Marks
- An alternative form exists for outputting a newline – **endl**
  - `cout << "Hello, World!\n";`
  - `cout << "Hello, World!" << endl;`

  These are both equivalent.

Formatting numbers

- Sometimes you may want to specify how you want your numbers to be output.
  - Example: when you output money, you often want two significant digits after the decimal (such as `$1265.00`.)
- `cout` is an **object** that has some **member functions** that aid in formatting output. (These OOP concepts will be taught later in the course.)
- `cout.setf(ios::fixed);`
  - Tells `cout` to output floating point values in fixed-point notation.
- `cout.setf(ios::showpoint);`
  - Tells `cout` to output floating-point values always including the decimal point.
- `cout.precision(2);`
  - Specifies the number of digits to output to the right of the decimal point.
    - (Seems to conflict with online reference definition? Go by my definition.)
- The following link has most of the flags you can use as arguments to the `setf` member function:
  - http://www.cplusplus.com/ref/iostream/ios_base/fmtflags.html

ex2.cpp

```
// ex2.cpp
#include <iostream>
using namespace std;

int main( ) {
  int hours_worked;
  double hourly_wage, week_salary;

  cout << "Enter the number of hours worked:\n";
  cin >> hours_worked;
  cout << "Now enter the hourly wage:\n";
  cin >> hourly_wage;

  // Do calculation
  week_salary = hours_worked * hourly_wage;

  // Format output
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(2);

  // Print output
  cout << "You worked " << hours_worked << " hours.\n";
  cout << "Your hourly wage is $" << hourly_wage << ".\n";
  cout << "You earned $" << week_salary << " this week.\n";

  return 0;
}
```

Types - double vs. int

- In C++, `2` and `2.0` are not the same number
  - A whole number such as `2` is of type `int`
  - A real number such as `2.0` is of type `double`
- There are different ways to write double constants in C++:
  - A number with a decimal point:
    - `2.0`
    - `23.0859`
  - Scientific Notation
    - `3.41e1` (means 34.1)
    - `3.67e10` (means 36700000000.0)
    - `5.89e-6` (means 0.00000589)
    - Number to the left of "`e`" must be an integer

Numeric Types

Type	Size in Bytes	Range	Precision
`short`	2	-32767 to 32767	N/A
`int`	4	-2147483647 to 2147483647	N/A
`long`	4	-2147483647 to 2147483647	N/A
`float`	4	Approx 10^{-38} to 10^{38}	7 digits
`double`	8	Approx. 10^{-308} to 10^{308}	15 digits
`long double`	10	Approx. 10^{-4932} to 10^{4932}	19 digits

- `short, int, long` are "integer" types
- `float, double, long double` are "floating-point" types.
- Precision to be explained in class.

Non-numeric types

- `char`
  - Short for character.
  - A variable of type `char` holds a single character.
  - Example character variable declarations:
    - `char symbol;`
    - `char letter = 'a';`
  - Character constants have a single letter surrounded by SINGLE quotation marks.
    - A single character between DOUBLE quotation marks has a string type. A string is different than a character in C++!
    - `"a"` is a string of characters containing one character
    - `'a'` is a single char constant

More on `char`

- Each character has a numeric value.
  - 'A' has the value of 65.
  - 'a' has the value of 97.
- To determine the numeric value of a character, you would use an ASCII table. An ASCII code is the numerical representation of a character such as 'a' or '@'.
- Appendix 3 in the book is an ASCII table containing only printable characters and their numeric equivalent.
- http://www.asciitable.com/
- The importance of the numeric value of characters will become clear when we discuss strings and string comparisons later.

Reading characters

- Consider the following:

```
char letter1, letter2;
cin >> letter1 >> letter2;
```

- If the input was:

  `J D`

  What would `letter1` and `letter2` contain?

- What value is stored in the memory location reserved for these variables?

ex3.cpp

```
// ex3.cpp
#include <iostream>
using namespace std;
int main( )
{
   char first, last, symbol;
   cout << "Enter your first and last initial:\n";
   cin >> first >> last;
   cout << "The two initials are:\n";
   cout << first << last << endl;
   cout << "Once more with a space:\n";
   symbol = ' ';
   cout << first << symbol << last << endl;
   return 0;
}
```


ex3 – Sample Output

```
Enter your first and last initial:
B K
The two initials are:
BK
Once more with a space:
B K
```

- What if we don't use a space when entering characters?

```
Enter your first and last initial:
BK
The two initials are:
BK
Once more with a space:
B K
```

- The extraction operator operating on a char variable reads in one single character. There is no need to skip any white space.

Boolean type

- `bool`
  - Short for boolean
  - Contains the value `true` or `false`
    - `true` and `false` are keywords in C++ that represent boolean constants.
    - C++ introduced these keywords in the1995 Draft ANSI Standard . They are not available in the C language.
  - Example declaration and use:
    - ```
      bool finished;
      finished = false;
      ```
  - The actual value that is stored in memory for a `bool` variable is a 1 for `true` and 0 for `false`.
      ```

Type Compatibilities

- Generally, the types of both sides of an assignment operator must match:

```
int i;
i = 2.99; // Incompatible types
```

- Some compilers will report an error, almost all will at least report a warning.
- In the above example, it will only store a 2 in the variable i, truncating the fractional part.
- Types that are smaller can be assigned to types that are the same size or larger in size without an error.
- Be aware of **truncation** when assigning floating point types to integer types.
- Keep your types the same on both sides of assignment statements. It's not required, but it helps keep your program clean and prevents unnecessary bugs. The compiler usually helps you out with this.

Type Compatibility Examples

- Example:

```
int i;
double d = 21.5;
i = d;
```

Will the compiler report a warning message? Why or why not?

- Example:

```
int i = 21;
double d;
d = i;
```

Will the compiler report an error? Why or why not?

- Example:

```
int i;
short s;
s = i;
```

Will the compiler report an error? Not all compilers do! Should they?

- Based on the previous variable declarations, what about:

```
i = s;
```


General type compatibility rules

- To be discussed in class:
  - assigning `int` to a `double`
  - assigning a `double` to an `int`
  - assigning a `char` to an `int`
  - assigning a `bool` to an `int`
  - assigning `short` to `integer` to `long`
  - assigning `long` to `integer` to `short`
  - assigning `float` to `double`

Arithmetic operators

- Operators include `+,-,*,/`
- Arithmetic operators are forms of **binary operators** because they take **two** *operands* before the expression can be evaluated.
- +,-,* operate the same with both floating point types and integer types
- Using / with floating point types behaves normally
- Using / with integer types **truncates** the fractional part.
  - 31.0 / 5.0 is 6.2
  - 31 / 5 is 6
  - 1.0 / 2.0 is ?
  - 1 / 2 is ?
  - 1 / 2.0 is ? tricky… (answers in class)

Modulus arithmetic operator - %

- The result of a modulus (or mod) operation is the remainder of an integer division.
- Example:
  - 31 / 6 = 5, what is the remainder?
  - Therefore, 31 % 6 =
    - (Answers given in class)

More on Arithmetic Operators

- Arithmetic operators can be used with any numeric type, except the mod operator (%)
  - % operator only operates on integer types
- An **operand** is a constant or variable expression used by the operator
- Result of an operator depends on the types of the operands
  - If both operands are int, the result is int
  - If one or both operands are double, the result is double

Arithmetic Expressions

- Precedence rules for operators are the same as used in your algebra classes
- Use parentheses to alter the order of operations

```
x + y * z      ( y is multiplied by z first)
(x + y) * z    ( x and y are added first)
```

- See book, Display 2.5, page 63 (70 in 4th ed.) for more examples.
- A complete set of C++ precedence rules are given in Appendix 2.

Examples

- What does the following program output?

```
int number;
number = (1/3) * 3;
cout << number;
```

- What does the following program output?

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
cout << f;
```

- If there's something wrong, how would you fix the above?

Shorthand Assignment Statements

- +=, -= , *=, /=, %=
- (Discussed in class)

Increment and Decrement operators

- ++ and --
- Discussed in class….

Introduction of Flow of Control

- There are times when you need to vary the way your program executes based on given input.
- The order in which statements in your program are executed is referred to as **flow of control.**
- Two methods of flow control will be introduced – **branching** and **looping**.

Branching

- When you want your program to execute one of two alternatives, you use the `if - else` statement.
- Example, suppose we were to calculate hourly wages, and overtime was allowed:
  - Regular time (up to 40 hours)
    - `gross_pay = rate * hours;`
  - Overtime (over 40 hours)
    - `gross_pay = rate * 40 + 1.5 * rate * (hours - 40);`
  - The program must choose which of these expressions to use

if - else Syntax

- The general syntax of the if – else statement is:
  - **`if`** (*Boolean_expression*)
    statement1;
    `else`
    statement2;
  - A Boolean_expression is an expression that evaluates to `true` or `false`.
  - If the *Boolean_expression* evaluates to true, then *statement1* is executed, otherwise, *statement2* is executed.
  - *Boolean_expression* can also be a numeric expression. If the expression evaluates to zero, the expression is false, otherwise, the expression is true.
  - The `else` part of the statement is optional.

Block Statements

- **Block statements** are a list of statements enclosed in a pair of braces.
- Required when you want your `if` or `else` to execute more than one statement:
  - **`if`** (*Boolean_expression*)
    {
    yes_statement1;
    …
    yes_statement_last;
    }
    `else`
    {
    no_statement1;
    …
    no_statement_last;
    }
- We often recommend beginning students to always use block statements to prevent unnecessary errors.

Comparison Operators

- Simple boolean expressions use **comparison operators** to compare two operands

Comparison Operators

Math Symbol	English	C++ Notation	C++ Sample	Math Equivalent
=	equal to	==	x + 7 == 2*y	$x+7=2y$
≠	not equal to	!=	ans != 'n'	ans ≠ 'n'
<	less than	<	count < m + 3	count < m + 3
≤	less than or equal to	<=	time <= limit	time ≤ limit
>	greater than	>	time > limit	time > limit
≥	greater than or equal to	>=	age >= 21	age ≥ 21

ex4.cpp

```
// ex4.cpp
// Written by Brian King
// Calculate a speeding fine.
// If the person was with 15 mph, the fine is a flat $100,
// otherwise, the fine is $100 plus $5 for every mph exceeded over 15 mph
#include <iostream>
using namespace std;

int main()
{
  int speed, limit, fine;

  cout << "Enter the speed limit: ";
  cin >> limit;
  cout << "Enter the speed you were driving: ";
  cin >> speed;

  if ((speed - limit) < 15)
    fine = 100;
  else
    fine = 100 + (speed - limit - 15) * 5;

  cout << "Your fine is $" << fine << endl;
  return 0;
}
```

Notice the arithmetic expression to the left of the <

Logical Operators && and ||

- `&&` - Logical "and" operator
- `||` - Logical "or" operator
- Recall boolean logic from high school math
  - AND: If **both** operands evaluate to true, then the expression is true, otherwise it is false.
  - OR: If **either** operand evaluates to true, then the expression is true, otherwise it is false.
- These give you a way to form more elaborate boolean expressions
  - `(`*`Bool_expr_1`*`) && (`*`Bool_expr_2`*`)`
  - `(`*`Bool_expr_1`*`) || (`*`Bool_expr_2`*`)`

Logical NOT Operator

- ! is the "NOT" Operator. It negates the result of a Boolean expression.
- Example:
  - `if (!(x < 5))`
    is another way of saying "if x is NOT less than 5"
  - `if (!(x == 5))`
    is another way of saying "if x is NOT equal to 5"
- Side note:
  - An operator that operates on one operand only is called a **unary** operator. ! is a unary operator.
  - An operator that operates on two operators is called a **binary** operator. && and || are binary operators.

More on Boolean Expressions

- Suppose you wanted to check to see if a integer variable x was greater than 10 and less than 20:
- In math, it's common to think (10 < x < 20). Don't write your boolean expressions this way!
  - `if (10 < x < 20) // this is an ERROR!`
- Your `if` statement would be:
  - `if ((x > 10) && (x < 20))`
- Be careful to use == when testing equality and not =
  - Example: Suppose you have the following code:

```
int x;
x = 0;
if (x = 3)
    cout << "x contains the number 3.\n";
else
    cout << "x does not contain the number 3.\n";
```

  - This will not produce an error:
    - It evaluates the boolean expression for the if statement.
    - This results in storing the value 3 into x. Therefore, the expression itself evaluates to 3.
    - Therefore, the expression returns true, because it is non-zero! Not what was intended.

ex5.cpp

```
// ex5.cpp
// Written by Brian King
// Purpose: Calculate a speeding fine.
#include <iostream>
using namespace std;

int main()
{
  int speed, limit, fine;

  cout << "Enter the speed limit :";
  cin >> limit;
  cout << "Enter the speed you were driving: ";
  cin >> speed;

  if ((speed - limit > 5) && (speed - limit < 15))
    fine = 100;
  else
    fine = 100 + (speed - limit - 15) * 5;

  // Don't output a fine if they weren't speeding!
  if (speed - limit > 5)
    cout << "Your fine is $" << fine << endl;
  else
    cout << "Good driver!" << endl;

  return 0;
}
```

- Illustrates a complex boolean expression.

ex6.cpp

```
#include <iostream>
using namespace std;

int main()
{
  int speed, limit, fine, exceeded;

  cout << "Enter the speed limit :";
  cin >> limit;
  cout << "Enter the speed you were driving: ";
  cin >> speed;

  // Calculate the amount they went over the speed limit
  exceeded = speed - limit;

  if ((exceeded > 5) && (exceeded < 15))
    fine = 100;
  else
    fine = 100 + (exceeded - 15) * 5;

  if (exceeded > 5)
  {
    cout << "Your fine is $" << fine << endl;
    if (exceeded >= 15)
      cout << "Bad driver! Naughty!" << endl;
  }
  else
    cout << "Good driver!" << endl;

  return 0;
}
```

- Illustrates use of block statements.
- Shows how an if statement can be embedded within an if statement.

Simple Loop Mechanism

- Many programs include code that need to be repeated many times.
  - Perhaps you have a grading program. You want to repeat your grading program for each test you collected.
- C++ contains numerous ways to create loops.
- We will introduce the `while` loop and `do-while` loop.

The `while` loop

- Syntax of the while loop:
  `while` (*Boolean_expression*)
  {
  Statement_1;
  …
  Statement_Last;
  }
- The statements between the braces are called the **body** of the loop.
  - (Again, notice that it's just a block statement.)
- Each execution of the body of the loop is called an **iteration** of the loop.
- NOTE: If you only have one statement that you need to loop on, you can omit the braces.

How the `while` loop works

- `while (`*Boolean_expression*`)`
  `{`
  Statement_body;
  `}`
  next_statement;
- First, the boolean expression is evaluated
  - If false, the program skips to the line following the while loop
  - If true, the body of the loop is executed
    - During execution, some item from the boolean expression is changed. This is known as the **altering statement**.
  - After executing the loop body, the boolean expression is checked again repeating the process until the expression becomes false
- The body of a while loop will not execute at all if the boolean expression is false on the first check

ex7.cpp

- Illustrates use of a while loop.

```
#include <iostream>
using namespace std;
int main( )
{
  int count_down;

  cout << "How many greetings do you want? ";
  cin >> count_down;

  while (count_down > 0)
  {
      cout << "Hello ";
      count_down = count_down - 1;
  }

  cout << endl;
  cout << "That's all!\n";

  return 0;
}
```

What is the boolean expression?

What is the body of the loop?

Which statement is the altering statement?

ex8.cpp – A factorial program

- Illustrates the use of a while loop.

```
#include <iostream>
using namespace std;

int main()
{
  int num, fac, i;

  cout << "Enter a number and I'll return the factorial: " << endl;
  cin >> num;

  // 0! is 1, so initialize fac to be 1.
  fac = 1;
  i = num;

  while (i > 0)
  {
    fac = fac * i;
    i = i - 1;
  }

  cout << "The factorial of " << num
       << " is " << fac << endl;

  return 0;
}
```


The `do-while` loop

- Syntax of the do-while loop:

```
do
{
   Statement_1;
   …
   Statement_Last;
} while (Boolean_expression);
```

- Similar to the while loop, except that the boolean expression is checked at the *end* of the loop instead of the beginning.
- The body of a do-while loop is always executed at least once.
- Don't forget the semi-colon at the end!

ex9.cpp – Interactive factorial program

```
int main()
{
    int num, fac, i;
    char ans;

    do
    {
        cout << "Enter a number and I'll return the factorial: " << endl;
        cin >> num;

        // 0! is 1, so initialize fac to be 1.
        fac = 1;
        i = num;

        while (i > 0)
        {
            fac = fac * i;
            i = i - 1;
        }

        cout << "The factorial of " << num
             << " is " << fac << endl;

        // Ask user to try again
        cout << "Try again? [y|n] ";
        cin >> ans;

    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

- Illustrates the use of a **do-while** loop.
- Shows how to interact with the user to repeat an action.

Infinite Loops

- Be careful to be sure your loop can exit properly.
- A loop that runs forever is called an **infinite loop**.
- Example:

```
int x = 1;
while (x != 10)
{
  cout << x << endl;
  x += 2;
}
```

- Why does this loop run forever?

Comments in C++

- Comments start with //
- Everything after // is ignored by the compiler.
- You may also see comments using /* and */.
  - `/* This is a comment */`
- Everything between /* and */ is ignored by the compiler, even if it expands multiple lines.
- A comment should explain code that is not immediately obvious!
- Should document variables where they are declared.
- Don't add obvious comments:
  - `int x;  // This is a variable.` (DUH!)
- See class web site for assignment grading guidelines for information on comments in your program.

#include directives and using namespace

- `#include <library_name>`
  - *library_name* refers to a header file that declares numerous types and identifiers pertinent to the library.
  - This informs the compiler that a specific library called *library_name* is going to be used in your program.
    - Example: #include <iostream>
- using namespace std;
  - This allows your program to use certain constants and identifiers that have been defined in the library.

Constants – The `const` keyword

- There are times when you want to "hard-wire" values in your program.
- Example: In our speeding fine program, we could have hard coded the speed limit variable in the program.
- To do so, you use the `const` keyword.
- Using `const` tells the compiler to do extra checking to ensure you didn't write any code that modifies the variable.
- `const` is a called a **modifier** because it places a restriction on the declaration tied to it.
- Typically, `const` identifiers are all UPPERCASE letters. This is a convention, not a requirement.

ex10.cpp

```
#include <iostream>
using namespace std;
int main()
{
    const int SPEED_LIMIT = 55;
    int speed, fine, exceeded;

    cout << "The speed limit is " << SPEED_LIMIT << ".\n";
    cout << "Enter the speed you were driving: ";
    cin >> speed;

    exceeded = speed - SPEED_LIMIT;

    if ((exceeded > 5) && (exceeded < 15))
        fine = 100;
    else
        fine = 100 + (exceeded - 15) * 5;

    if (exceeded > 5)
    {
        cout << "You were driving " << exceeded
             << " miles over the speed limit." << endl;
        cout << "Your fine is $" << fine << endl;

        if (exceeded >= 15)
           cout << "Bad driver! Naughty!" << endl;
    }
    else
        cout << "Good driver!" << endl;

    return 0;
}
```


Design style

- Indentation makes your program easier to read!
- You will get a point or two deducted if you do not follow appropriate style as outlined on the class website.
  - Indentation
  - Body inside while, do-while, if-else, etc… should be indented
  - Braces delimiting compound statements should be aligned together.
  - Comments
  - Meaningful variable names
  - Consistency of indentation, brace alignment, etc..

The End