Computer Science II
Lecture 1
Introduction and Reveiw

1 Discussion of Syllabus

- Instructor, TAs, office hours
- The course web site, `http://www.cs.rpi.edu/courses/spring07/cs2`, is up and running!
- Material: textbooks, lecture notes, and examples posted on the web
- Overall expectations
- Prerequisites
- Lectures and lecture notes
- C++ vs. Java
- Requirements:
  - Labs (12%),
  - Homeworks (33%),
  - Quiz and exams (55%)
- Late policy
- Schedule of material
- Academic integrity

2 Overview of Lectures 1 and 2

- Review of CSci I level material, focusing primarily on C++ constructs.
- Background on additional concepts that we will use during the semester.
- Built around examples. Source code is posted on line. Extra practice problems and solutions are also posted.
- In lecture we will only touch on a subset of the material.
- Students who find this material challenging need to work hard to catch up.

3 Example 1: Hello World

Here is the standard introductory program...

```
-----------------------------------------

// a small C++ program
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}

-----------------------------------------
```

Small though it is, it may be used to illustrate a number of important points, as we now discuss.

3.1 Basic Syntax

- Comments are indicated using `//` for single line comments and `/*` and `*/` for multi-line comments.
- `#include` asks the compiler for parts of the standard library that we wish to use (e.g. the input/output stream function `std::cout`).
- `int main()` is a necessary component of all C++ programs; it returns a value (integer in this case) **and** it could have parameters (Lecture 2).
- `{ }`: the curly braces indicate to C++ to treat *everything* between them as a unit.

3.2 The Standard Library

- The standard library is not a part of the core C++ language. Instead it contains types and functions that are important extensions.
  - In our programming we will use the standard library to such a great extent that it will "feel" like part of the C++ core language.
- streams are the first component of the standard library that we see.
- `std` is a *namespace* that contains the standard library
- `std::cout` and `std::endl` are defined in the standard library (in particular, in the standard library header file `iostream`).

3.3 Expressions

- Each **expression** has a **value** and 0 or more **side effects**.
- An expression followed by a semi-colon is a **statement**. The semi-colon tells the computer to "throw away" the value of the expression.
- The line

```
std::cout << "Hello, world!" << std::endl;
```

  is really two expressions and one statement.

  - The first expression is

```
std::cout << "Hello, world!"
```

  - The value of this expression is `std::cout` and the side effect is that `Hello, world!` is output.
  - Using the value of the first expresssion, the second expression is just

```
std::cout << std::endl;
```

  - Perhaps this can be better understood by writing the original statement, equivalently, as

```
(std::cout << "Hello, world!") << std::endl;
```

- `"Hello, world!"` is a *string literal*.

3.4 Aside: C++ vs. Java

The following is provided as additional material for students who have learned Java and are now learning C++.

- In Java, everything is an object and everything "inherits" from `java.lang.Object`. In C++, functions can exist outside of classes. In particular, the `main` function is never part of a class.
- Source code file organization in C++ does not need to be related to class organization as it does in Java. On the other hand, creating one C++ class (when we get to classes) per file is the *preferred* organization, with the *main* function in a separate file on its own or with a few helper function.
- Compare the "hello world" example above in C++ to the same example in Java:

```
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello  World");
  }
}
```

  - The Java object member function declaration

```
public static void main(String  args[])
```

Plays the same role as

```
int main()
```

in the C++ program. A primary difference is that there are no string arguments to the C++ `main` function. Very soon (next week) we will start adding these arguments, although in a somewhat different syntactic form.

– The statement

```
System.out.println("Hello  World");
```

is analogous to

```
std::cout << "Hello, world!" << std::endl;
```

The `std::endl` is required to end the line of output (and move the output to a new line), whereas the Java `println` does this as part of its job.

4 Example 2: Temperature Conversion

Our second introductory example converts a Fahrenheit temperature to a Celsius temperature and decides if the temperature is above the boiling point or below the freezing point:

```
#include <iostream>
using namespace std;   // Eliminates the need for std::

int main()
{
  // Request and input a temperature.
  cout << "Please enter a Fahrenheit temperature:  ";
  float fahrenheit_temp;
  cin >> fahrenheit_temp;

  // Convert it to Celsius and output it.
  float celsius_temp = (fahrenheit_temp - 32) * 5.0 / 9.0;
  cout << "The equivalent Celsuis temperature is " << celsius_temp
       << " degrees.\n";

  // Output a message if the temperature is above boiling or below freezing.
  const int BoilingPointC = 100;
  const int FreezingPointC = 0;
  if ( celsius_temp > BoilingPointC )
    cout << "That is above the boiling point of water.\n";
  else if ( celsius_temp < FreezingPointC )
    cout << "That is below the freezing point of water.\n";

  return 0;
}
```

4.1 Variables and Constants

- A variable is an object with a name (a C++ identifier such as `fahrenheit_temp` or `celsius_temp`).
- An object is computer memory that has a type.
- A type is a structure to memory and a set of operations.
- Too abstract? Think about `float` variables:
  - Each variable has its own 4 bytes of memory, and this memory is formatted according floating point standards for what represents the exponent and mantissa.
  - Many operations are defined on floats, including addition, subtraction, etc.
  - A float is an object

- A constant — such as `BoilingPointC` and `FreezingPointC` — is an object with a name, but a constant object may not be changed once it is defined (and initialized). Any operations on integer types may be applied to a constant int, except operations that change the value.
- Note that the use of capitals and `_` in the identifier is merely a matter of style.

4.2 Expressions, Assignments and Statements

Consider the *statement*

```
float celsius_temp = (fahrenheit_temp - 32) * 5.0 / 9.0;
```

- The calculation on the right hand side of the `=` is an expression.
  - You should review the definition of C++ arithmetic expressions and operator precedence. The rules are pretty much the same in C++ and in Java.
- The value of this expression is assigned — stored in the memory location — of the newly created float variable `celsius_temp`.

4.3 Conditionals and IF statements

Intuitively, the meaning of the code

```
if ( celsius_temp > BoilingPointC )
  cout << "That is above the boiling point of water.\n";
else if ( celsius_temp < FreezingPointC )
  cout << "That is below the freezing point of water.\n";
```

should be pretty clear.

- The general form of an if-else statement is

```
if ( conditional-expression )
   statement;
else
   statement;
```

- Each `statement` may be a single statement, such as the `cout` statement above, a structured statement, or a compound statement delimited by {...}
  - The second `if` is actually a structured statement that is part of the `else` of the first `if`.
- Students should review the rules of logical expressions and conditionals.
- These rules and the meaning of the if - else structure are essentially the same in Java and in C++.

5 Example 3: Julian Day

```
--------------------------------------------------------------------------------

#include <iostream>
using namespace std;

const int DaysInMonth[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

//  The following function returns true if the given year is a leap year
//  and returns false otherwise.
bool
is_leap_year( int year )
{
  return year % 4 == 0 && ( year % 100 != 0 || year % 400 == 0 );
}

//  Calculate and return the Julian day associated with the given
//  month and day of the year.
int
julian_day( int month, int day, int year )
{
  int jday = 0;
  for ( unsigned int m=1; m<month; ++m )
    {
      jday += DaysInMonth[m];
      if ( m == 2 && is_leap_year(year) ) ++jday;  // February 29th
    }
  jday += day;
  return jday;
}

int
main()
{
  cout << "Please enter three integers (a month, a day and a year): ";
  int month, day, year;
  cin >> month >> day >> year;

  cout << "That is Julian day " << julian_day( month, day, year ) << endl;
  return 0;
}

--------------------------------------------------------------------------------
```

5.1 Arrays and Constant Arrays

- An array is a fixed, consecutive sequence of objects all of the same type.

- The following declares an array of 15 double values:

```
double a[15];
```

- The values are accessed through subscripting operations:

```
int i = 5;
a[i] = 3.14159;
```

  This assigns the value 3.14159 to location `i=5` of the array. Here `i` is the *subscript* or *index*

- C++ array indexing starts at 0.
- Arrays are fixed size, and each array knows NOTHING about its own size. The programmer must write code that keeps track of the size of each array.
  - During Lecture 2, we will discuss a standard library generalization of arrays, called *vectors*, which are full-fledged objects and do not have any of these restrictions.
- In the statement:

```
const int DaysInMonth[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

  - `DaysInMonth` is an array of 13 constant integers.
  - The list of values within the braces initializes the 13 values of the array, so that `DaysInMonth[0] == 0`, `DaysInMonth[1]==31`, etc.
  - The array is global, meaning that it is accessible in all functions within the code (after the line in which the array is declared). Global constants such as this array are usually fine, whereas global variables are generally a VERY bad idea.

5.2 Functions and Arguments

- Functions are used to:
  - Break code up into modules for ease of programming and testing, and for ease of reading by other people (never, ever, under-estimate the importance of this!).
  - Create code that is reusable at several places in one program and by several programs.
- Each function has a sequence of parameters and a return type. For example,

```
int julian_day( int month, int day, int year )
```

  has a return type of `int` and three parameters, each of type `int`.
- The order of the parameters in the calling function (the main function in this example) *must match the order of the parameters* in the function prototype.

5.3 for loops

- The basic form of a `for` loop is

  ```
  for ( expr1; expr2; expr3 )
    statement;
  ```

  where

  - `expr1` is the initial expression executed at the start before the loop iterations begin;
  - `expr2` is the test applied before the beginning of each loop iteration, the loop ends when this expresion evaluates to `false` or `0`;
  - `expr3` is evaluated at the very end of each iteration;
  - `statement` is the "loop body"

- The `for` loop from the `julian_day` function,

  ```
  for ( unsigned int m=1; m<month; ++m )
    {
      jday += DaysInMonth[m];
      if ( m == 2 && is_leap_year(year) ) ++jday;  // February 29th
    }
  ```

  adds the days in the months 1..`month`-1, and adds an extra day for Februarys that are in leap years.

- `for` loops are essentially the same in Java and in C++.

5.4 A More Difficult Logic Example

Consider the single statement in the function `is_leap_year`:

```
return year % 4 == 0 && ( year % 100 != 0 || year % 400 == 0 );
```

This is important to understand.

- If the year is not divisible by 4, the function immediately returns `false`, without evaluating the right side of the `&&`.
- For a year that is divisible by 4, if the year is not divisible by 100 or is divisible by 400 (and is obviously divisible by 100 as well), the function returns true.
- Otherwise the function returns false.
- The function will not work properly if the parentheses are removed, in large part because `&&` has higher precedence than `||`.

6 Example 4: Julian Date to Month/Day

- The source code is attached to these notes. We'll examine some of the structure and then focus on the `month_and_day` function
- The `main` function is the first function in the file as opposed to the last. This is a stylistic choice.
- Function prototypes are inserted above the main function in the file.
- The name of the month is output using the function `output_month_name`, which is just a big `switch` statement.
  - Note the use of the `break` statement at the end of each `case`.
  - See Chapter 4 of Malik for detailed discussion of `switch`.

6.1 Month And Day Function

```
//  Compute the month and day corresponding to the Julian day within
//  the given year.
void
month_and_day( int julian_day, int year,
               int & month, int & day )
{
  bool month_found = false;
  month = 1;

  //  Loop through the months, subtracting the days in this month
  //  from the Julian day, until the month is found where the
  //  number of remaining days is less than or equal to the total
  //  days in the month.
  while ( !month_found )
    {
      //  Calculate the days in this month by looking it up in the
      //  array.  Add one if it is a leap year.
      int days_this_month = DaysInMonth[month];
      if ( month == 2 && is_leap_year(year) )
        ++ days_this_month;

      if ( julian_day <= days_this_month )
        month_found = true;   // Done!
      else
        {
          julian_day -= days_this_month;
          ++ month;
        }
    }
  day = julian_day;
}
```

- We'll assume you know the basic structure of a `while` loop and focus on the underlying logic.
- A `bool` variable (which can take on only the values `true` and `false`) called `month_found` is used as a flag to indicate when the loop should end.
- The first part of the loop body calculates the number of days in the current month (starting at one for January), including a special addition of 1 to the number of days for a February (`month == 2`) in a leap year.
- The second half decides if we've found the right month.
- If not, the number of days in the current month is subtracted from the remaining Julian days, and the month is incremented.

Understanding the logic of functions such as this one is important for developing your programming skills.

6.2 Value Parameters and Reference Parameters

Consider the line in the main function that calls `month_and_day`

```
month_and_day( julian, year, month, day_in_month );
```

and consider the function prototype

```
void month_and_day( int julian_day, int year,
                    int & month, int & day )
```

Note in particular the `&` in front of the third and fourth parameters.

- The first two parameters are *value parameters*.
  - These are essentially local variables (in the function) whose initial values are copies of the values of the corresponding argument in the function call.
  - Thus, the value of `julian` from the main function is used to initialize `julian_day` in function `month_and_day`.
  - Changes to value parameters do NOT change the corresponding argument in the calling function (`main` in this example).
- The second two parameters are *reference parameters*, as indicated by the `&`.
  - Reference parameters are just aliases for their corresponding arguments. No new variable are created.
  - As a result, changes to reference parameters are changes to the corresponding variables (arguments) in the calling function.
- "Rules of thumb" for using value and reference parameters:
  - When a function (e.g. `is_leap_year`) needs to provide just one result, make that result the return value of the function and pass other parameters by value.
  - When a function needs to provide more than one result (e.g. `month_and_day`), these results should be returned using multiple reference parameters.

  We will see minor variations on these rules as we proceed this semester.

6.3 Arrays as Function Arguments

Consider the following function

```
void
do_it( double a[], int n )
{
  for ( int i=0; i<n; ++i )
    if ( a[i] < 0 ) a[i] *= -1;
}
```

- What does it do?
- The important point about this function is that the changes made to array `a` are permanent, even though `a` is a value parameter!
- Reason: What's passed by value is the memory location of the start of the array. The entries in the array are not copied, and therefore changes to these entries are permanent.
- The number of locations in the array to work on — the value parameter `n` — must be passed as well because arrays have no idea about their own size.

6.4 Exercises

Here are some practice problems. Solutions will be posted on the course web site, and we will them in class if we have time.

1. What would be the output of the above program if the main program call to `month_and_day` was changed to

   ```
   month_and_day( julian, year, day_in_month, month );
   ```

   and the user provided input that resulted in `julian == 50` and `year == 2007`? What would be the additional output if we added the statement

   ```
   cout << month << endl;
   ```

   immediately after the function call in the main function?

2. What is the output of the following code?

   ```
   void swap( double x, double &y )
   {
     double temp = x;
     x = y;
     y = temp;
   }

   int main()
   {
   ```

```
    double a = 15.0, b=20.0;
    cout << "a = " << a << ", b = " << b << endl;
    swap ( a, b );
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
  }
```

7 Scope

The following code will not compile. We want to understand why not, fix the code (minimally) so that it will, and then determine what will be output.

```
int main()
{
  int a = 5, b = 10;
  int x = 15;
  {
    double a = 1.5;
    b = -2;
    int x = 20;
    int y = 25;
    cout << "a = " << a << ", b = " << b << '\n'
         << "x = " << x << ", y = " << y << endl;
  }
  cout << "a = " << a << ", b = " << b << '\n'
       << "x = " << x << ", y = " << y << endl;
  return 0;
}
```

- The *scope* of a name (identifier) is the part of the program in which it has meaning.
- Curly braces, `{ }`, establish a new scope — this includes functions and compound statements.
  - This means scopes may be nested.
  - Identifiers may be re-used as long as they are in different scopes.
- Identifiers (variables or constants) within a scope hide identifiers within an outer scope having the same name. This does not change the values of hidden variables or constants — they are just not accessible.
- When a `}` is reached, a scope ends. All variables and constants (and other identifiers) declared in the scope are eliminated, and identifiers from an outer scope that were hidden become accessible again in code that follows the end of the scope.
- The operator `::` establishes a scope as well. For example, `std::cout` refers to the identifier `cout` within the scope of namespace `std`.

8 Algorithm Analysis: What, Why, How?

- What?
  - Analyze code to determine the time required, usually as function of the size of the data being worked on.
- Why?
  - We want to do better than just implementing and testing every idea we have.
  - We want to know why one algorithm is better than another.
  - We want to know the best we can do. (This is often quite hard.)
- How? There are several possibilities:
  1. Don't do any analysis; just use the first algorithm you can think of that works.
  2. Implement and time algorithms to choose the best.
  3. Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
  4. Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.
- What happens in practice?
  - 99% of the time: rough count similar to #4 as a function of the size of the data. Use order notation to simplify the resulting function and even to simplify the analysis that leads to the function.
  - 1% of the time: implement and time.

What follows is a quick review of counting and the order notation.

8.1 Exercise: Counting Example

Suppose `foo` is an array of `n` doubles, initialized with a sequence of values.

- Here is a simple algorithm to find the sum of the values in the vector:

```
double sum = 0;
for ( int i=0; i<n; ++i )
  sum += foo[i];
```

- How do you count the total number of operations?
- Go ahead and try. Come up with a function describing the number of operations.
- You are likely to come up with different answers. How do we resolve these differences?

8.2 Order Notation

The following discussion emphasizes intuition. That's all we care about in CS II. For more details and more technical depth, see any textbook on data structures and algorithms.

- Definition

  > Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k \times f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.

- As a result, algorithms requiring $3n+2$, $5n-3$, $14+17n$ operations are all $O(n)$ (i.e. in applying the definition of order notation $f(n) = n$).
- Algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$ (i.e. in applying the definition of order notation $f(n) = n^2$).
- Intuitively (and importantly), we determine the order by finding the asymptotically dominant term (function of n) and throwing out the leading constant. This term could involve logarithmic or exponential functions of n.
- Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

8.3 Common Orders of Magnitude

Here are the most commonly occurring orders of magnitude in algorithm analysis.

- $O(1)$: *Constant time.* The number of operations is independent of the size of the problem.
- $O(\log n)$: *Logarithmic time.*
- $O(n)$: *Linear time*
- $O(n \log n)$
- $O(n^2)$: *Quadratic time.* Also, *Polynomial time*
- $O(n^2 \log n)$
- $O(n^3)$: *Cubic time.* Also, *Polynomial time*
- $O(2^n)$: *Exponential time*

8.4 Significance of Orders of Magnitude

- On a computer that performs 10^8 operations per second:
  - An algorithm that actually requires $15n \log n$ operations requires about 3 seconds on a problem of size $n = 1,000,000$, and 50 minutes on a problem of size $n = 100,000,000$.
  - An algorithm that actually requires n^2 operations requires about 3 hours on a problem of size $n = 1,000,000$, and 115 days on a problem of size $n = 100,000,000$.
- Thus, the leading constant of 15 on the $n \log n$ does not make a substantial difference. What matters is the n^2 vs. the $n \log n$.
- Moreover, in practice the leading constants usually do not vary by a factor of 15.

8.5 Back to Analysis: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`

```
int loc=0;
bool found = false;
while ( !found && loc < n )
  {
    if ( x == foo[loc] )
       found = true;
    else
       loc ++ ;
  }
if ( found ) cout << "It is there!\n";
```

- Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

8.6 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size vector, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

8.7 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem.
  - For arrays and other "container classes" this will generally be the number of values stored.
- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations, such as comparisons.
- Do the count, using order notation to describe the result.

8.8 Examples: Loops

In each case give an order notation estimate as a function of `n` which here does note

- Version A:

```
int count=0;
for ( int i=0; i<n; ++i )
  for ( int j=0; j<n; ++j )
    ++count;
```

- Version B:

```
int count=0;
for ( int i=0; i<n; ++i )
  ++count;

for ( int j=0; j<n; ++j )
  ++count;
```

- Version C:

```
int count=0;
for ( int i=0; i<n; ++i )
  for ( int j=i; j<n; ++j )
    ++count;
```

- How many operations in each?

9 Summary

- Course overview
- Four review examples
- Basic structure of C++ code and the C++ main function

- iostreams and the standard library
- Java vs. C++
- Variables, constants, operators, expressions and statements
- if-else
- arrays
- functions
- for loops
- logic
- `switch` statements and `while` loops
- Value parameters and reference parameters
- Arrays as function parameters
- Scope
- Order notation

All of this material should be review of your C++ knowledge or should be easily transferred from your experience with other programming languages.