CSCI-1200 Computer Science II — Fall 2006
Lecture 1 — Introduction and Background

Instructor

Professor Barb Cutler
309A Materials Research Center (MRC), x3274
cutler@cs.rpi.edu

Today

- Discussion of Website & Syllabus:

  http://www.cs.rpi.edu/academics/courses/fall06/cs2/

  - Instructor, TAs, & office hours
  - Course overview & emphasis
  - Prerequisites, expectations, & grading
  - Calendar
  - Textbooks, lecture notes, & web resources
  - Homework: Electronic Submission & Late Policy
  - Academic Integrity

- Quick Review/Overview of C/C++ Programming:

  - the `main` function
  - iostreams & the standard library
  - C++ vs. Java
  - variables, constants, operators, expressions, and statements
  - if-else conditional
  - arrays
  - `for` & `while` loops
  - functions and parameter passing

Reading Material

The material for today's lecture is covered in the following sections of the two books.

Accelerated C++ (Koenig and Moo):
Chapter 0,
Chapter 2 (except strings, which we will review in Lecture 2)

C++ Programming (Malik, *optional*):
Chapter 2, covering the very basics of C++,
Chapter 3 (pages 96-102 only), covering input and output,
Chapter 4, covering if and switch statements, as well as basic expression logic,
Chapter 5, covering while and for loops,
Chapter 6, covering simple user-defined functions.
Chapter 7 (pages 305-338), covering parameter passing,
Chapter 9 (pages 423-442), covering arrays

This should all be review from CS 1 or a course at an equivalent level. For those of you who started with Java, most of this will be very familiar. Pay most attention to the differences. Overall, you should use the material in Malik as a reference for more information about the topics covered in class.

1.1 Example 1: Hello World

Here is the standard introductory program. We use it to illustrate a number of important points.

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

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

1.2 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 (next week).
- `{ }`: the curly braces indicate to C++ to treat *everything* between them as a unit.

1.3 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. 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`).

1.4 Expressions

- Each *expression* has a *value* and 0 or more *side effects*. Side effects include: printing to the screen, writing to a file, changing the value of a variable, or crashing the computer.
- An expression followed by a semi-colon is a *statement*. The semi-colon tells the computer to "throw away" the value of the expression.
- This line is really two expressions and one statement:

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

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

1.5 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 this declaration in the C++ program: `int main()`

  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 the syntax will be different.

- 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.

1.6 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;
}
```

1.7 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.
- For example, a `float` is an object and each `float` variable is assigned to 4 bytes of memory, and this memory is formatted according floating point standards for what represents the exponent and mantissa. There are many operations defined on floats, including addition, subtraction, etc.
- 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 the `integer` type may be applied to a constant `int`, except operations that change the value.

1.8 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 (Malik, Chapter 2). 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`.

1.9 Conditionals and IF statements

Intuitively, the meaning of this code should be pretty clear:

```
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";
```

- 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++.

1.10 Example 3: Julian Day

```
// Convert a day and month within a given year to the Julian day.
#include <iostream>
using namespace std;

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

// 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);
}

// Returns 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;
}
```

1.11 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. The following code assigns the value 3.14159 to location `i=5` of the array. Here `i` is the *subscript* or *index*.

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

- 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. Next week we will see a standard library generalization of arrays, called *vectors*, which do not have 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.

1.12 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. The function declaration or *prototype* below has a return type of `int` and three parameters, each of type `int`:

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

- 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.

1.13 `for` Loops

- Here is the basic form of a `for` loop:

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

  - `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 expression evaluates to `false` or `0`;
  - `expr3` is evaluated at the very end of each iteration;
  - `statement` is the "loop body"
- The `for` loop below from the `julian_day` function, adds the days in the months 1..`month`-1, and adds an extra day for Februarys that are in leap years.

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

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

1.14 A More Difficult Logic Example

Consider the code in the function `is_leap_year`:

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

- 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 part because `&&` has higher precedence than `||`.

1.15 Example: Julian Date to Month/Day

In the next example we will extract the month & day from the Julian date. Note that in this code:

- 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`.

```
// Convert a Julian day in a given year to the associated month and day.
#include <iostream>
using namespace std;

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

// Function prototypes are listed here so that the main function will
// know what their interfaces look like.
//
// In general, if function A calls function B, function B must be
// defined "before" A or function B's prototype must be specified
// before A.  We often put the prototypes in a header file (.h)
bool is_leap_year(int year);
void month_and_day(int julian_day, int year, int & month, int & day);
void output_month_name(int month);


// The main function handles the I/O.  Defining the main function at
// the top of the function is a stylistic choice.
int main() {
  cout << "Please enter two integers giving the Julian date and the year: ";
  int julian, year;
  cin >> julian >> year;

  int month, day_in_month;
  month_and_day(julian, year, month, day_in_month);
  cout << "The date is ";
  output_month_name(month);
  cout << " " << day_in_month << ", " << year << endl;
  return 0;
}

// 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);
}
```

```
// 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 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;
}

// Output a string giving the name of the month.
void output_month_name(int month) {
  switch (month) {
  case 1: cout << "January"; break;
  case 2: cout << "February"; break;
  case 3: cout << "March"; break;
  case 4: cout << "April"; break;
  case 5: cout << "May"; break;
  case 6: cout << "June"; break;
  case 7: cout << "July"; break;
  case 8: cout << "August"; break;
  case 9: cout << "September"; break;
  case 10: cout << "October"; break;
  case 11: cout << "November"; break;
  case 12: cout << "December"; break;
  default: cout << "Illegal month";
  };
}
```

1.16 Month And Day Function

- 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.

1.17 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.
- In general, the "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.

1.18 Arrays as Function Arguments

- What does the following function do?

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

- Changes made to array `a` are permanent, even though `a` is a value parameter! Why? Because 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.

1.19 Exercises

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 == 2006`? What would be the additional output if we added this statement immediately after the function call in the main function?

   ```
   cout << julian << endl;
   ```

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;
}
```

Important: *It will be assumed that you have read the following statement thoroughly. If you have any questions, contact the instructor or the TAs immediately.*

CSCI-1200 Computer Science II
Academic Integrity Policy

Copying, communicating, or using disallowed materials during an exam is cheating, of course. Students caught cheating on an exam will receive an F in the course and will be reported to the Dean of Students. Students are allowed to assist each other in labs, but must write their own lab solutions.

Academic integrity is a difficult issue for programming assignments. Students naturally want to work together, and it is clear they learn a great deal by doing so. Getting help is often the best way to interpret error messages and find bugs, even for experienced programmers. In response to this, the following rules will be in force for programming assignments:

- Students are allowed to work together in designing algorithms, in interpreting error messages, and in discussing strategies for finding bugs, but NOT in writing code.
- Students may not share code, may not copy code, and may not discuss code in detail (line-by-line or loop-by-loop) while it is being written or afterwards. This extends up to two days after the submission deadline.
- Similarly, students may not receive detailed help on their code from individuals outside the course. This restriction includes tutors.
- Students may not show their code to other students as a means of helping them. Sometimes good students who feel sorry for struggling students are tempted to provide them with "just a peek" at their code. Such "peeks" often turn into extensive copying, despite prior claims of good intentions.
- Students may not leave their code (either electronic versions or printed copies) in publicly accessible areas. Students may not share computers in any way when there is an assignment pending.

We use an automatic code comparison tool to help spot assignments that have been submitted in violation of these rules. The tool takes all assignments from all sections and compares them, highlighting regions of the code that are similar. Code submitted by students who followed the rules produces less than 10% overlap. Code submitted by students who broke the rules produces anywhere from about 30% to 100% overlap.

We (the instructor and the TAs) check flagged pairs of assignments very carefully ourselves, and make our own judgment about which students violated the rules of academic integrity on programming assignments. When we believe an incident of academic dishonesty has occurred, we contact the students involved.

Students caught cheating on programming assignments will be punished. The standard punishment for the first offense is a 0 on the assignment and a 5 percentage point penalty on the semester average. Students whose violations are more flagrant will receive a higher penalty. For example, a student who outright steals another student's code will receive an F in the course immediately. Students caught a second time will receive an immediate F, regardless of circumstances. Each incident will be reported to the Dean of Students.

Name:

Signature: **Date:**