

CSCI-1200 Computer Science II — Fall 2008

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/fall108/cs2/>
 - Instructor, TAs, & office hours
 - Course overview & emphasis
 - Prerequisites, expectations, & grading
 - Calendar
 - Textbooks, lecture notes, ALAC drop-in tutoring, & 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
 - scope
 - algorithm analysis / order notation

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 and other code 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 may have parameters.
- `{ }`: 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. Soon 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 Celsius 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 from any reference textbook. 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-length, 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. Very soon 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

```
// 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. 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.
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);

int main() { // The main function handles the I/O.
    int julian, year, month, day_in_month;
    cout << "Please enter two integers giving the Julian date and the year: ";
    cin >> julian >> year;
    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;
    // Subtracting the days in each month from the Julian day, until the
    // remaining days is less than or equal to the total days in the month.
    while (!month_found) {
        int days_this_month = DaysInMonth[month];
        if (month == 2 && is_leap_year(year)) // Add one if it is a leap year.
            ++days_this_month;
        if (julian_day <= days_this_month)
            month_found = true; // Done!
        else {
            julian_day -= days_this_month;
            ++month;
        }
    }
    day = julian_day;
}

void output_month_name(int month) { // Output a string giving the name of the 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 == 2008`? 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;
}
```

1.20 Scope

- 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.
- 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 :: (namespaces) establishes a scope as well.

1.21 Scope Exercise

The following code will not compile. Why not? Fix it (minimally) & determine the 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 << endl << "x = " << x << ", y = " << y << endl;
    }
    cout << "a = " << a << ", b = " << b << endl << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

1.22 Algorithm Analysis

Why should we bother?

- 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 do we do it? There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.
- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- 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.

1.23 Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the array:

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

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of* n .

1.24 Exercise: Which Algorithm is Best?

An venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the “best” algorithm?

1.25 Order Notation Definition

In *CSII* we will focus on the intuition of order notation. 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 $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- For example, algorithms requiring $3n + 2$, $5n - 3$, and $14 + 17n$ operations are all $O(n)$. This is because we can select values for k and n_0 such that the definition above holds. (What values?) Likewise, algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$.
- Intuitively, 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.

1.26 Common Orders of Magnitude

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC*. e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR*. e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL*. e.g., find closest pair of points.
- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL*. e.g., Fibonacci, playing chess.

1.27 Exercise: A Slightly Harder Example

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

```
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";
```

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

- For a given fixed size array, 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 $O(n)$.

1.29 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. For example, in the previous exercise, we could count the number of comparisons.
- Do the count and use order notation to describe the result.

1.30 Exercises: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of `n`:

- | | | | | | |
|----|--|----|---|----|--|
| 1. | <pre>int count=0; for (int i=0; i<n; ++i) for (int j=0; j<n; ++j) ++count;</pre> | 2. | <pre>int count=0; for (int i=0; i<n; ++i) ++count; for (int j=0; j<n; ++j) ++count;</pre> | 3. | <pre>int count=0; for (int i=0; i<n; ++i) for (int j=i; j<n; ++j) ++count;</pre> |
|----|--|----|---|----|--|

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. Please sign this form and give it to your TA during your first lab section.*

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 from prior terms, and internet resources.
- 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 all prior terms 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. All students caught cheating on a programming assignment (both the copier and the provider) will be punished. The standard punishment for the first offense is a 0 on the assignment and a letter grade reduction on the final semester grade. 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.

Refer to the *Rensselaer Handbook of Student Rights and Responsibilities* for further discussion of academic dishonesty. Note that: “Students found in violation of the academic dishonesty policy are prohibited from dropping the course in order to avoid the academic penalty.”

In Fall 2007, Fall 2006, and Spring 2005, a total of 29 students taking CSCI 1200 were found in violation of the academic integrity policy and were punished.

Name:

Signature:

Date: