

CSCI-1200 Data Structures — Spring 2013

Lecture 1 — Introduction to C++, STL, & Strings

Instructor

Professor Barb Cutler
331A Materials Research Center (MRC), x3274
cutler@cs.rpi.edu

Today

- Discussion of Website & Syllabus:
<http://www.cs.rpi.edu/academics/courses/spring13/ds/>
- Getting Started in C++ & STL, C++ Syntax, STL Strings

1.1 Transitioning from Python to C++

- Python is a great language to learn the power and flexibility of programming and computational problem solving. This semester we will work in C++ and study lower level programming concepts, focusing on details including efficiency and memory usage.
- Outside of this class, when working on large programming projects, you will find it is not uncommon to use a mix of programming languages and libraries. The individual advantages of Python and C++ (and Java, and Perl, and C, and UNIX bash scripts, and ...) can be combined into an elegant (or terrifyingly complex) masterpiece.
- Here are a few excellent references Prof. Stewart recommends to help you transition from Python to C++:
<http://www.cse.msu.edu/~cse231/python2Cpp.html>
<http://dehn.slu.edu/courses/spring08/180/transition.pdf>
<http://www4.wittenberg.edu/academics/mathcomp/shelburne/comp255/notes/Python2Cpp.pdf>

1.2 A Short Example: Temperature Conversion

```
/* This program 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>

int main() {
    // Request and input a temperature.
    std::cout << "Please enter a Fahrenheit temperature: ";
    float fahrenheit_temp;
    std::cin >> fahrenheit_temp;
    // Convert it to Celsius and output it.
    float celsius_temp = (fahrenheit_temp - 32) * 5.0 / 9.0;
    std::cout << "The equivalent Celsius temperature is " << celsius_temp << " degrees." << std::endl;
    // Output a message if the temperature is above boiling or below freezing.
    const int BoilingPointC = 100;
    const int FreezingPointC = 0;
    if (celsius_temp > BoilingPointC)
        std::cout << "That is above the boiling point of water.\n";
    else if (celsius_temp < FreezingPointC)
        std::cout << "That is below the freezing point of water.\n";
    return 0; /* normal non-error exit code */
}
```

1.3 Some Basic C++ 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.

1.4 The C++ Standard Library, “STL”

- The standard library contains types and functions that are important extensions to the core C++ language. We will use the standard library to such a great extent that it will feel like part of the C++ core language.
- I/O streams are the first component of the standard library that we see.
- `std` is a *namespace* that contains the standard library.
- `std::cout` (“console output”) and `std::endl` (“end line”) are defined in the standard library header file, `iostream`

1.5 Compiled Languages vs. Interpreted Languages

- C/C++ is a *compiled language*, which means your code is processed (compiled & linked) to produce a low-level machine language executable that can be run on your specific hardware. You must re-compile & re-link after you edit any of the files – although a smart development environment or `Makefile` will figure out what portions need to be recompiled and save some time. Also, if you move your code to a different computer you will usually need to recompile. Generally the extra work of compilation produces an efficient and optimized executable that will run fast.
- In contrast, many newer languages including Python, Java, & Perl are *interpreted languages*, that favor incremental development where you can make changes to your code and immediately run all or some of your code without waiting for compilation. However, an interpreted program will almost always run slower than a compiled program.
- These days, the process of compilation is almost instantaneous for simple programs, and in this course we encourage you to follow the same incremental editing & frequent testing development strategy that is employed with interpreted languages.
- Finally, many interpreted languages have a Just-In-Time-Compiler (JIT) that can run an interpreted programming language and perform optimization on-the-fly resulting in program performance that rivals optimized compiled code. Thus, the differences between compiled and interpreted languages are somewhat blurry.
- You will practice the cycle of coding & compilation & testing during Lab 1. You are encouraged to find a development environment (code editor & compiler) that allow you to be most productive. Ask the your lab TAs & mentors about their favorite programming environments! The website includes many helpful links as well.
- As you see in today’s handout, C++ has more required punctuation than Python, and the syntax is more restrictive. The compiler will proofread your code in detail and complain about any mistakes you make. Even long-time C++ programmers make mistakes in syntax, and with practice you will become familiar with the compiler’s error messages and how to correct your code.

1.6 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. We’ll see several other uses of the `const` keyword in the next few weeks.

1.7 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 Java and Python.
- The value of this expression is assigned (stored in the memory location) of the newly created float variable `celsius_temp`.

1.8 Conditionals and IF statements

- 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 `{...}`. Note that in the earlier example, the second `if` is actually part of the `else` clause of the first `if`.

1.9 Functions and Arguments

- Each function has a sequence of parameters and a return type. The function declaration below has a return type of `int` and three parameters, each of type `int`:

```
// 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;
}
```

- The order and types of the parameters in the calling function (the main function in this example) must match the order and types of the parameters in the function prototype.

1.10 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 example above adds the days in the months `1..month-1`, and adds an extra day for February’s that are in leap years.

1.11 Larger Complete Example: Julian Date to Month/Day

```
// Convert a Julian day in a given year to the associated month and day.
#include <iostream>
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.
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;
    std::cout << "Please enter two integers giving the Julian date and the year: ";
    std::cin >> julian >> year;
    month_and_day(julian, year, month, day_in_month);
    std::cout << "The date is ";
    output_month_name(month);
    std::cout << " " << day_in_month << ", " << year << std::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: std::cout << "January"; break;
        case 2: std::cout << "February"; break;
        case 3: std::cout << "March"; break;
        case 4: std::cout << "April"; break;
        case 5: std::cout << "May"; break;
        case 6: std::cout << "June"; break;
        case 7: std::cout << "July"; break;
        case 8: std::cout << "August"; break;
        case 9: std::cout << "September"; break;
        case 10: std::cout << "October"; break;
        case 11: std::cout << "November"; break;
        case 12: std::cout << "December"; break;
        default: std::cout << "Illegal month";
    };
}
```

1.12 C-style 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;
```

- In C/C++, array indexing starts at 0.
- Arrays are fixed size, and each array knows *NOTHING* about its own size. The programmer must keep track of the size of each array. (Note: C++ STL has generalization of C-style arrays, called *vectors*, which do not have these restrictions.)
- In the statement:

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

1.13 Month And Day Function

- 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 parts of the *while* 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.

1.14 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).
 - We'll see more examples of the importance of value vs. reference parameters as the semester continues.

- 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.

1.15 Python Strings vs. C chars vs. C-style Strings vs. C++ STL Strings

- Strings in Python are immutable, and there is no difference between a string and a char in Python. Thus,

```
s1 = 'a'
s2 = "a"
```

are both strings, not individual characters.

- A “C-style” string is an array of `chars` that ends with the special char `'\0'`. C-style strings (char arrays) can be edited, and there are a number of helper functions to help with common operations. However...
- The “C++-style” STL `string` type has a wider array of operations and functions, which are more convenient and more powerful.

1.16 About STL String Objects

- A `string` is an object type defined in the standard library to contain a sequence of characters.
- The `string` type, like all types (including `int`, `double`, `char`, `float`), defines an interface, which includes construction (initialization), operations, functions (methods), and even other types(!).
- When an object is created, a special function is run called a “constructor”, whose job it is to initialize the object. There are several ways of constructing string objects:
 - By default to create an empty string: `std::string my_string_var;`
 - With a specified number of instances of a single char: `std::string my_string_var2(10, ' ');`
 - From another string: `std::string my_string_var3(my_string_var2);`
- The notation `my_string_var.size()` is a call to a function `size` that is defined as a **member function** of the `string` class. There is an equivalent member function called `length`.
- Input to string objects through streams (e.g. reading from the keyboard or a file) includes the following steps:
 1. The computer inputs and discards white-space characters, one at a time, until a non-white-space character is found.
 2. A sequence of non-white-space characters is input and stored in the string. This overwrites anything that was already in the string.
 3. Reading stops either at the end of the input or upon reaching the next white-space character (without reading it in).
- The (overloaded) operator `'+'` is defined on strings. It concatenates two strings to create a third string, without changing either of the original two strings.
- The assignment operation `'='` on strings overwrites the current contents of the string.
- The individual characters of a string can be accessed using the subscript operator `[]` (similar to arrays).
 - Subscript 0 corresponds to the first character.
 - For example, given `std::string a = "Susan";`
Then `a[0] == 'S'` and `a[1] == 'u'` and `a[4] == 'n'`.

- Strings define a special type `string::size_type`, which is the type returned by the string function `size()` (and `length()`).
 - The `::` notation means that `size_type` is defined within the scope of the `string` type.
 - `string::size_type` is generally equivalent to `unsigned int`.
 - You will have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.

This seems like a lot to remember. Do I need to memorize this? Where can I find all the details on `string` objects?

1.17 Problem: Writing a Name Along a Diagonal

- Let's write a simple program to read in a name using `std::cin` and then output a fancier version to `std::cout`, written along a diagonal. Here's how the program should behave:

```
What is your first name? Sally
S
 a
 1
 1
 y
```

Hint: You will need to use nested for loops OR a single loop and exploit properties of the `string` class.

```
#include <iostream>
#include <string>

int main() {
    std::cout << "What is your first name? ";
    std::string first;
    std::cin >> first;
    for (int i = 0; i < first.size(); i++) {
        for (int j = 0; j < i; j++) {
            std::cout << ' ';
        }
        std::cout << first[i] << std::endl;
    }
    return 0;
}
```

1.18 Putting a Frame around the Output

- Next, output the name within a frame of asterisks:

```
What is your first name? Bob

*****
*   *
* B *
* o *
*  b *
*   *
*****
```

- For this version, there are two main difficulties:
 - Making sure that we can put the characters in the right places on the right lines.
 - Getting the asterisks in the right positions and getting the right number of blanks on each line.
- There are often many ways to solve a programming problem. Sometimes you can think of several, while sometimes you struggle to come up with one.
- When you have finished a problem or when you are thinking about programming examples, it is useful to think about the core ideas used. If you can abstract and understand these ideas, you can later apply them to other problems.

1.19 Implementation: Framed Diagonal Name

- Let's solve this problem by thinking about what changes from one line to the next. If we have a "blank line" string, containing only the beginning and ending asterisks and the spaces between. We will overwrite the appropriate blank character, output the string, and then replace the blank character (so we are ready for the next line).

```
#include <iostream>
#include <string>

int main() {
    std::cout << "What is your first name? ";
    std::string first;
    std::cin >> first;

    const std::string star_line(first.size()+4, '*');
    const std::string blanks(first.size()+2, ' ');
    const std::string empty_line = '*' + blanks + '*';
    std::string one_line = empty_line;

    std::cout << '\n' << star_line << '\n' << empty_line << std::endl;

    // Output the interior of the greeting, one line at a time.
    for (unsigned int i = 0; i < first.size(); ++i) {
        // Create the output line by overwriting a single character from the
        // first name in location i+2. After printing it restore the blank.
        one_line[ i+2 ] = first[i];
        std::cout << one_line << '\n';
        one_line[ i+2 ] = ' ';
    }

    std::cout << empty_line << '\n' << star_line << std::endl;

    return 0;
}
```

1.20 L-Values and R-Values

- Consider the simple code below. String `a` becomes "Tim". No big deal, right? Wrong!

```
string a = "Kim";
string b = "Tom";
a[0] = b[0];
```

- Let's look closely at the line: `a[0] = b[0];` and think about what happens.

In particular, what is the difference between the use of `a[0]` on the left hand side of the assignment statement and `b[0]` on the right hand side?

- Syntactically, they look the same. But,
 - The expression `b[0]` gets the char value, 'T', from string location 0 in `b`. This is an *r-value*.
 - The expression `a[0]` gets a reference to the memory location associated with string location 0 in `a`. This is an *l-value*.
 - The assignment operator stores the value in the referenced memory location.

The difference between an *r-value* and an *l-value* will be especially significant when we get to writing our own operators later in the semester

- What's wrong with this code?

```
std::string foo = "hello";
foo[2] = 'X';
cout << foo;
'X' = foo[3];
cout << foo;
```

Your C++ compiler will complain with something like: "non-lvalue in assignment"

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 Data Structures 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 and ask the TAs for help, but each student must write up their own lab solutions.

Academic integrity is a more complicated issue for individual programming assignments, but one we take very seriously. 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 high level algorithms, in interpreting error messages, and in discussing strategies for testing and finding bugs, but NOT in writing code or pseudocode. Detailed algorithms and implementation must be done individually.
- Students may not jointly write code or pseudocode, may not share code or pseudocode, may not copy code or pseudocode, and may not discuss code or pseudocode 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, etc.
- 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.

The instructor checks flagged pairs of assignments very carefully, to determine which students may have violated the rules of academic integrity on programming assignments. When it is believed that an incident of academic dishonesty has occurred, the involved students are contacted and a meeting is scheduled. 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.”

Historically, each semester, 5-10 students are found to be in violation of the academic dishonesty policy.

Name:

Section #:

Signature:

Date: