C++ Lab 02 - Command Line Arguments and Strings

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Ten Short CPP Labs

IAP 2024

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Lab Two Overview and Objectives **3**

2 The Meat and Potatoes: Variables, Operators, and Control Structures **3**
2.1 Exercise 1: A Program to Compute 10! 3
2.2 Exercise 2: A Program to Compute BIG! 4

3 Command Line Arguments **4**
3.1 Exercise 3: A Program to Compute N! with N Given on the Command Line 5

4 Strings in C++ **6**
4.1 C-Style Strings 6
4.2 C++ Style Strings 8
4.3 Exercise 4: A Program to Query String Command Line Args 10

5 Solutions to Exercises **11**
5.1 Solution to Exercise 1 11
5.2 Solution to Exercise 2 12
5.3 Solution to Exercise 3 13
5.4 Solution to Exercise 4 14

1 Lab Two Overview and Objectives

This lab addresses much of the core functionality of the C/C++ language, including variables, operators, control structures. If you have programmed before in a similar language like Java, you will find reading a quick-read. Otherwise newcomers should take some time to absorb the on-line tutorials recommended here. We will use this knowledge to expand our abilities to handle C++ command line arguments with a few exercises. The final topic covered will introduce you to C++ strings, a very useful component of the language.

- Variables, Operators and Control Structures
- Handling C++ Command Line Arguments
- Strings and Basic String Functions

2 The Meat and Potatoes: Variables, Operators, and Control Structures

If you've programmed in another language previously the issues of C/C++ variables, literals, and operators are not terribly different in syntax structure. Same with for-loops, while-loops and if-then blocks. Take a few minutes to get familiar with them using the below on-line material, and return to a simple exercise to confirm your knowledge.

- Variables and Types: `http://www.cplusplus.com/doc/tutorial/variables`
- Constants: `http://www.cplusplus.com/doc/tutorial/constants`
- Operators: `http://www.cplusplus.com/doc/tutorial/operators`
- Statements and Flow Control: `http://www.cplusplus.com/doc/tutorial/control`

2.1 Exercise 1: A Program to Compute 10!

Write a program to calculate 10! by first setting a variable to 10 and then using a for-loop to calculate the result. Before your function returns, produce the answer. Call your file `factorial.cpp` and build it to the executable `factorial`. When your program runs, it should be invocable from the command line with:

```
$ ./factorial
10! is equal to: 3628800
```

The solution to this exercise is in Section 5.1.
Comments on Exercise 1:
In Exercise 1 you likely kept a running total as you were building up to the 10! calculation. If this running total was an integer variable (type `int`), things would be fine up to about 12!. Go back and try it, changing to use an `int` if you didn't already. Compare the results of 12! and 13!. At first glance it may be easy to miss that something is wrong, afterall the answer for 13! is bigger than 12!. But take a closer look and draw your own conclusion from the following three facts:

```
479001600   // The reported value of 12!
4790016000  // 12! * 10 (by just tacking a 0 to 12!)
1932053504  // The reported value of 13! (which is 13 * 12!, and must be > 10*12!)
```

The problem comes from the fact that a regular `int` variable has only 4 bytes, or 32 bits. It should be able to represent 4,294,967,296 distinct values, but it uses half of these for representing negative numbers, so the true range from -2,147,483,648 to 2,147,483,647. And note where this number lies w.r.t. 13! and 14!:

```
479001600    // 12!
2147483648   // (2**32 / 2) the max number in a 32 bit integer
6227020800   // 13!
```

We point this out here since C++ is a strongly typed language - you have to declare the variable type right up front. And some consideration should be made in that choice. You could also use a `double` here which allows for representation of much larger numbers, using 8 bytes, but you lose precision as the numbers get higher. If you think this is an esoteric issue, and you're a fan of the Korean pop star PSY, check out the following slashdot article:
http://beta.slashdot.org/story/210605
(and then check out the MIT version of this video if you haven't seen it)

2.2 Exercise 2: A Program to Compute BIG!

Write a program to calculate 13! by first setting a variable to 13 and then using a for-loop to calculate the result. Use two variables to store the value of N!, one with type `int` and one with type `long int`. Call your file `factorial_longint.cpp` and build it to the executable `factorial_longint`. Before your function returns, produce both answers. When your program runs, it should be invocable from the command line with:

```
$ ./factorial_longint
13!  is: 1932053504 (using int)
13!  is: 6227020800 (using long int)
```

Comments on Exercise 2:
If you think it's a bit silly to hard-code a variable value like 13 in this example, you're right. We should be able to pass this in from the command line, and that's what we'll do next.

The solution to this exercise is in Section 5.2.

3 Command Line Arguments

A very common part of C++ programs is the ability to handle *command line arguments*. Instead of hard coding a number like 13 in the previous exercise to calculate 13!, it should be possible to pass this on the command line:

```
$ ./factorial 12
479001600
$ ./factorial 13
6227020800
```

The ability to create programs that handle command line arguments will be super convenient for the rest of our labs. It is also an absolutely essential part of writing good code in MIT 2.680 and in our marine robotics lab in general as we will see. So in this section we will try our hand with command line arguments to build a better "factorial" program that takes the N in N! from the command line. To so we'll need a bit of background on parsing command line parameters found here:

- How to Parse Command Line Parameters: http://www.cplusplus.com/articles/DEN36Up4/

From the above tutorial page we see that command line parameters are implemented using C-style string of type `char*`. Modern C++ has a `string` class to better handle this, but command line arguments in the `main()` function still use C-style strings. So we need to know a little bit about C-style strings and how to convert them to a numerical value. To convert a C-style string (of type `char*`) to a number we'll use the following function:

```
int atoi(char*);
```

To use this function, you will need to use the library in which it is implemented by invoking `#include <cstdlib>`. Thus if you have a string variable (of type `char*`), e.g., `sval="23"`, you should be able to use the function like this.

```
...
#include <cstdlib>
...
int numerical_val = atoi(sval);
...
```

We now have what we need to write the improved factorial program.

3.1 Exercise 3: A Program to Compute N! with N Given on the Command Line

Write a program to calculate N! by reading N from the command line. If no argument or more than one argument is provided, the program should exit with a usage message guiding the user how to use the program. Call your file `factorial_cmdargs.cpp` and build it to the executable `factorial_cmdargs`. When your program runs, it should be invocable from the command line with:

```
$ ./factorial_cmdargs 12
479001600
$ ./factorial_cmdargs 13
6227020800
$ ./factorial_cmdargs
Usage: factorial_cmdargs NUMBER
$ ./factorial_cmdargs 14 23 24
Usage: factorial_cmdargs NUMBER
```

The solution to this exercise is in Section 5.3.

Comments on Exercise 3:

- In this exercise we used the `atoi(char*)` function. You can find out more about this function right from the command line using the `man` program very likely already installed and invocable in your shell by typing:

  ```
  $ man atoi
  ```

  This `man` utility will be useful to you in the future. Try to remember that it's there. You also don't need a network connection for it to work, so it's even better than going into a web browser and searching on "C++ atoi".

- If you wanted to convert a C-style string to a floating point value like the type `double`, you could use the `atof(char*)` function instead. How could you possibly know to look for such a function if you don't already know the function name? Notice that when you pull up the man page for `atoi`, there is a `SEE ALSO` section at the bottom, common on many man pages. So when you need a function similar to one you're already familiar with, this is not a bad way to proceed.

  ```
  $ man atoi
  ...
  SEE ALSO
    atof(3), atol(3), strtod(3), strtol(3), strtoul(3), xlocale(3)
  ...
  ```

4 Strings in C++

Strings in C++ are generally handled by using the `string` class. This class was not part of C which uses a different approach often referred to as C-Style strings. Since C++ is largely backward compatible to support C code, both kinds of strings are common in C++ code. There are a ton of utilities for both, and we aim here to (a) introduce the two and describe some of their differences, and (b) introduce a few common utilities for manipulating strings.

4.1 C-Style Strings

C-Style strings are nothing more than array of characters followed by a special (NULL) character to indicate the end of a string. A C-Style string may be declared by:

```
char  mystring[] = "Hello";
or
char *mystring = "Hello";
```

The above two styles are equivalent under the hood, although newer compilers may warn that the latter style of declaration and initialization is deprecated. But both essentially declare that `mystring`

is an array of characters and that `mystring` is a *pointer* to the first character in the array. Here's a way of thinking about `mystring` in memory.

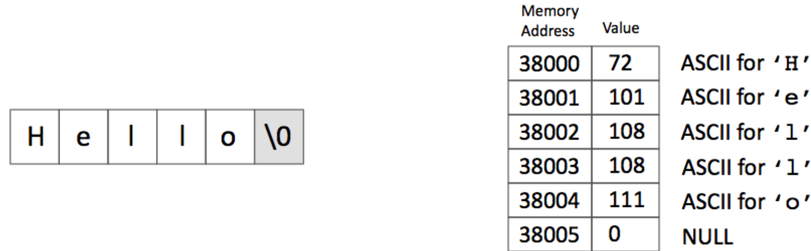

Figure 1: A C-Style string is an array of characters terminated by a NULL character.

To compare, a variable of type `char` containing a simple character `'H'`, would simply hold the value `72`, the ASCII value for `'H'`. By contrast, the `mystring` variable would hold the address in memory, e.g., 38000 in this example, where the array begins. The convention of using one extra NULL character indicating the end of the array means `mystring` doesn't need to somehow also know about the array length. Clever, right? And this type of string is still used all throughout perfectly modern C++ code wherever you see any reference to variables or arguments of type `char*`.

A further discussion on "Null terminated strings" can be found here: http://www.cplusplus.com/doc/tutorial/ntcs/.

Arrays of C-Style Strings

Sometimes we are interested in reasoning about *arrays of strings*. Again, C++ has more modern ways of handling this, but C has it's own way too which is still supported in C++. The C way of handling arrays of strings normally wouldn't warrant early attention in a C++ tutorial structure, but since they are right there in our face at the beginning of every darn C++ main() function, it's worth discussing. So consider an array of two strings, `"Hello"` and `"World"`. The second string is handled the same way as our first example, with no guarantee of where it may reside in memory relative to the first string:

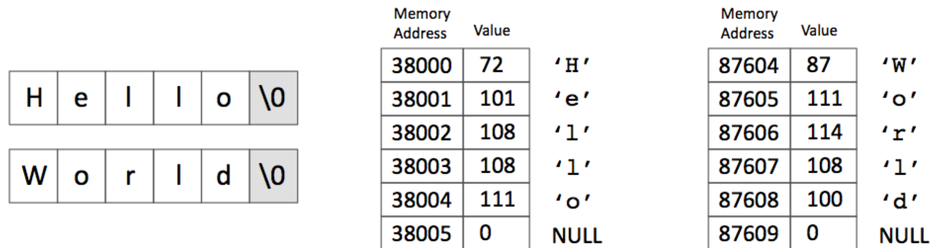

Figure 2: Two C-Style strings in different locations in memory.

So an array of C-Style strings can be thought of as any of the following three equally correct statements: "an array of arrays", or "an array of pointers", or "pointer to an array of pointers",

and could be declared with either of the below syntax.

```
char **my_words;
or
char *my_words[];
```

By our two word example, holding "Hello" and "World", my_words is an array of pointers (memory addresses) where the two strings reside, an my_strings would hold the value 92100:

Memory Address	Value	
92100	38000	Hello
92101	87604	World

Figure 3: An array of C-Style strings is array of pointers to strings.

In the case of character arrays (C-Style strings), there is this nice convention that the array is terminated by a NULL character. So the length of the string is embedded in the string representation itself. In the case of an *array of strings*, there is no such convention. Therefore when we pass in an array of strings on the command line, we need two arguments:

```
main(int argc, char** argv)
```

The argc argument tells us the length of the array of strings, given by argv.

4.2 C++ Style Strings

In C++, most uses of a string no longer use the C-Style string but rather the string class implemented in the STL (standard template library), now an official part of the C++ language. We haven't covered C++ classes yet, but that shouldn't stop us from introducing strings here.

```
string a = "Hello";
string b = "World";
string sentence = a + " " + b + "!";
cout << "Here is a sentence: " << sentence << endl;
```

```
Here is a sentence: Hello World!
```

There is a ton of material on C++ strings on the web, reflecting the dozens of functions and operations supported with strings. Here we try to introduce the top handful of utilities that cover most initial situations where you will want to use strings. These are:

- **Length or size checking:** The two really useful functions are

  ```
  unsigned int length();
  unsigned int size();
  ```

They both do the same thing. It's one way to check for an empty string or get the string length before iterating through with a for-loop over each of the characters. If you want to read more about these and few related functions:

http://www.learncpp.com/cpp-tutorial/17-3-stdstring-length-and-capacity/

- **Concatenation:** Concatenating or appending strings is easy with a bunch of choices. This page is a good introduction:

  http://www.learncpp.com/cpp-programming/17-6-stdstring-appending/

- **Substring checking:** Often we want to know if a string contains, begins with or ends with another given string. And sometimes we want to create a new string from a portion of another string. We'll need two of these functions (the `substr()` and `find()` functions) in our last exercise. They are discussed here:

  http://www.cplusplus.com/reference/string/string/substr/, and

  http://www.cplusplus.com/reference/string/string/find/

- **Conversion to/from C-Style strings:** Often we need to convert between C-Style strings and C++ strings. From C-Style to C++ strings is easy, just assign one to the other:

```
char *c_string = "hello";
string cppstring = c_string;
```

  In the other direction you can use the `c_str()` function defined on the C++ string class:

```
string cpp_string = "hello";
char *c_string = cpp_string.c_str();
```

- **Accessing individual characters:** A C++ string is still an array of characters underneath and thus can be accessed by an index.

```
string cpp_string = "hello";

char a = cpp_string[0];      // 'h'
char b = cpp_string.at(4);  // 'o' (another way of accessing)
cpp_string[0] = 'H';        // Change from 'h' to 'H'
cpp_string.at(4) = 'O';     // Change from 'o' to 'O'
```

- **Conversion to numerical values:** To convert a C++ string to a numerical value we can used the old C methods. Recall that the `c_str()` function on a string returns a C-Style (`char*`) string.

```
string str_flt = "123.456";
string str_int = "100";

double val_flt = atof(str_flt.c_str());
int    val_int = atoi(str_flt.c_str());
```

- **Conversion from numerical values:** To convert a number to a C++ string we can use the `stringstream` utility:

```
#include <sstream>
...
double dval = 123.456;
stringstream ss;
ss << dval;
string sval = ss.str();
```

4.3 Exercise 4: A Program to Query String Command Line Args

Write a program to accept any amount of strings on the command line, and determine if a given string is among those provided. The query string should be provided with the `--search=` argument, for example `--search=apples`. Your program should accept the `--search=` argument regardless of where it is provided among the command line arguments.

Call your file `str_search.cpp` and build it to the executable `str_search`. When your program runs, it should be invocable from the command line with:

```
$ ./str_search --search=apples pears grapes mangos apples bananas
The pattern was found.
$ ./str_search pears grapes --search=apples mangos apples bananas
The pattern was found.
$ ./str_search pears grapes --search=apples mangos bananas
The pattern was not found.

$ ./str_search pears grapes apples bananas
Usage: str_search PATTERN str str ... str
```

The solution to this exercise is in Section 5.4.

Comments on Exercise 4:

In this exercise you will likely need to make two passes on the given command line arguments. You also will likely need to know about the `find()` and `substr()` functions defined on the string class. In this respect your job will be easier if you convert the command line arguments into C++ strings as you process them. The links to more info on the `find()` and `substr()` functions are provided again below.

http://www.cplusplus.com/reference/string/string/substr/, and
http://www.cplusplus.com/reference/string/string/find/

5 Solutions to Exercises

5.1 Solution to Exercise 1

```
/*--------------------------------------------------------------------*/
/*  FILE:  factorial.cpp  (Second C++ Lab Exercise 1)                 */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/factorial.cpp          */
/*  BUILD: g++ -o factorial factorial.cpp                             */
/*  RUN:   ./factorial                                                */
/*--------------------------------------------------------------------*/

#include <iostream>

using namespace std;

int main()
{
  int total = 1;
  int nfact = 10;

  for(int i=1; i<=nfact; i++) {
    total *= i;
  }

  cout << nfact << "!  is: " << total << endl;
  return(0);
}
```

Comments on the Solution to Exercise 1:

- Recall the two lines:

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

  allow us to use the `cout` function later on to put output to the screen. `cout` is an output *stream* defined in `iostream` library we #include here. The namespace declaration of `std` allows us to use `cout` rather than the fully specified name `std::out`.
- The for-loop in this example has an open and close-brace. With only one line, the brace is optional. Usually when we have only one line we omit the braces, but in this case we leave them in.
- The program finishes by returning zero. If this line were omitted, zero would have been returned anyway. We leave it here to show good practice. In general, programs should return zero when they end normally, and with a non-zero value otherwise.

5.2 Solution to Exercise 2

```
/*----------------------------------------------------------------------*/
/*  FILE:  factorial_longint.cpp  (Second C++ Lab Exercise 2)           */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/factorial_longint.cpp    */
/*  BUILD: g++ -o factorial_longint factorial_longint.cpp               */
/*  RUN:   ./factorial_longint                                          */
/*----------------------------------------------------------------------*/

#include <iostream>

using namespace std;

int main()
{
  long int total_long = 1;

  int total = 1;
  int nfact = 13;

  for(int i=1; i<=nfact; i++) {
    total *= i;
    total_long *= i;
  }

  cout << nfact << "!  is: " << total << " (using int)" << endl;
  cout << nfact << "!  is: " << total_long << " (using long int)"<< endl;
  return(0);
}
```

5.3 Solution to Exercise 3

```
/*----------------------------------------------------------------------*/
/*  FILE:  factorial_cmdargs.cpp  (Second C++ Lab Exercise 3)           */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/factorial_cmdargs.cpp    */
/*  BUILD: g++ -o factorial_cmdargs factorial_cmdargs.cpp               */
/*  RUN:   ./factorial_cmdargs 12                                       */
/*----------------------------------------------------------------------*/

#include <iostream>   // For terminal I/O using cout
#include <cstdlib>    // For converting C-Style string to number with atoi

using namespace std;

int main(int argc, char **argv)
{
  if(argc != 2) {
    cout << "Usage: factorial_cmdargs NUMBER" << endl;
    exit(1);
  }

  int nfact = atof(argv[1]);

  long int total = 1;
  for(int i=1; i<=nfact; i++)
    total *= i;

  cout << nfact << "!  is: " << total << endl;
  return(0);
}
```

5.4 Solution to Exercise 4

```
/*----------------------------------------------------------------------*/
/*  FILE:  str_search.cpp  (Second C++ Lab Exercise 4)                  */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/str_search.cpp            */
/*  BUILD: g++ -o str_search str_search.cpp                             */
/*  RUN:   ./str_search --search=apples pears apples grapes             */
/*----------------------------------------------------------------------*/

#include <iostream>   // For terminal I/O using cout

using namespace std;

int main(int argc, char **argv)
{
  string search_pattern;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--search=") == 0)
      search_pattern = argi.substr(9);
  }

  if(search_pattern == "") {
    cout << "Usage: str_search PATTERN str str ... str" << endl;
    exit(1);
  }

  bool found = false;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--search=") != 0)
      if(argi == search_pattern)
        found = true;
  }

  if(found)
    cout << "The pattern was found." << endl;
  else
    cout << "The pattern was not found." << endl;

  return(0);
}
```