Comp 40 Lab 0:
Under the C++

New for Spring 2016. By Mark, Max Bernstein, and Carter.

Transitioning from C++ to C

The purspose of this lab is to help you get comfortable with programming in C rather than C++. To that end, we will explore some aspects of programming in C. We'll talk more about this in class, but you'll mostly be picking things up on your own. This lab gets you started. As you proceed, you may consult with members of the course staff and each other.

If you haven't already, you may want to consult Mark's Whirlwind Tour of C as well as the course C idioms. (Yes, the tour gives away one of the answers below. You must still understand it!)

Here are some things you may be used to in C++ that are not in C:

- There is no `bool` type built in to C. C uses 0 for false and any other value as true. You can `#include <stdbool.h>` and then use the type `bool` and constants `true` and `false`, but you still need to be aware of this fact. Hanson's code often uses `int` for logically boolean values, for example.
- There is no built-in `string` type, and handling string data is much more clumsy in C. For string data, C uses arrays of characters. These arrays do not know how long they are, unlike C++ strings, but like regular arrays. A common way to handle regular strings is to use an array of characters (bytes), and the last character of the string is the special NUL character (written `'\0'` whose value is zero.). This is not the same as a NULL pointer, and confusing the two can lead to strange problems. String constants (written between double quotes) are represented this way. For example, the string `"foo"` represents a sequence of four characters: `'f'`, `'o'`, `'o'`, `'\0'`. If you write a string constant, the terminating `'\0'` is inserted for you by the compiler. But when you manipulate strings, you need to be aware of its existence.

  Since strings are arrays, they are passed around using a pointer to the first character in the sequence, and their type is pointer-to-character (`char *a_string;`).

- NUL-terminated strings have a wide variety of library functions that work on them.
- C++ I/O streams aren't in C. You cannot use >> for program output or << for program input. For printing, you will often use *formatted output* using the functions `printf` and `fprintf`. You'll see examples in the links given above, and you can also use `man` and tutorials on the web. To use the standard Input/Output library, including `printf` you must `#include <stdio.h>`

Getting set up

You will need some files. To make a directory for this lab with the necessary files:

```
git clone /comp/40/git/under-the-c++
```

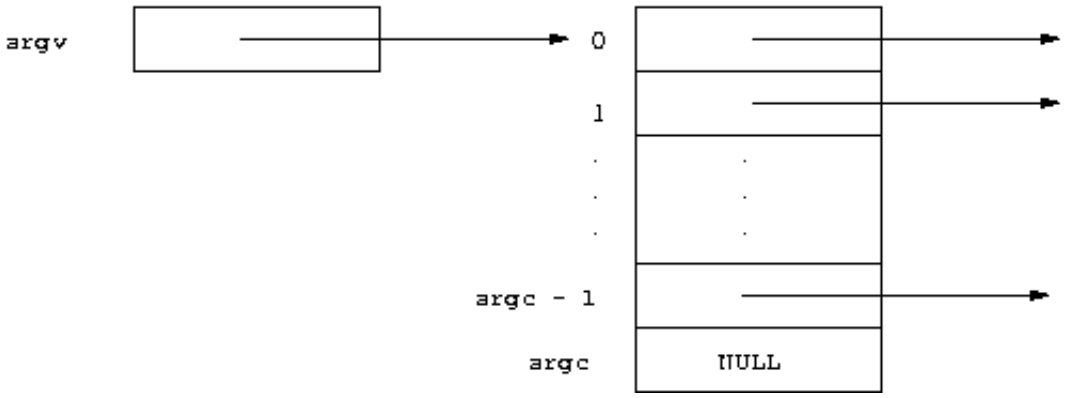
Problem 1: Command-line Arguments (`argc` and `argv`)

This program is very simple in the end, but you'll have to figure out command line arguments as well as some details about C strings.

`main()` takes arguments:

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

where `argc` is the number of arguments specified on the command line that invoked the program, and `argv` is an array (a *vector*) of C (i.e., NUL-terminated) strings, one for each input argument. `argv[0]` is the name of the program being invoked, and `argv[1]` up to `argv[argc - 1]` are the rest. C also guarantees that `argv[argc]` is the `NULL` pointer:



`argv` is an array whose elements are pointers to characters (`char *`).

Your task: Using this information, write a C program in the file print_args.c so that the program prints out all command line arguments in order (excluding the command name). For example, running `./print_args hello world` will print `hello world`.

Problem 2: Exit Codes

In C++ programs, nearly all `main` functions end with the line `return 0;`. This is because, when your program ends, it exits and leaves behind an integer value that can be interpreted by whomever ran the program (e.g., the shell or the command line interpreter). 0 is typically interpreted to mean that the program completed execution successfully (think “no problem”), while non-zero values are interpreted as failures. Different programs have different conventions for exit codes, especially with regard to non-zero values. (The `man` pages will often say how to interpret such codes. Try `man grep` and scroll down to the section called *Exit Codes*.) To make things clearer, in Comp 40 we ask that you use the macros `EXIT_SUCCESS` and `EXIT_FAILURE`. Unless otherwise specified, any program that ends due to error should return or exit with `EXIT_FAILURE`, and any program that executes successfully ends with `EXIT_SUCCESS`.

You must `#include <stdlib.h>` to get these macro definitions.

Note that this convention is for programs in the shell, *not* within C code. In C, 0 is typically interpreted as false, and non-zero as true, so that code enclosed by `if (0) { ... }` will never execute, while code enclosed by `if (1) { ... }` always will.

With this in mind, write a program called `check_args` so that, whenever it is called with an even number of command line arguments (not counting the program name), it exits with `EXIT_SUCCESS`, and whenever it is called with an odd number (again, not counting the program name), it exits with

EXIT_FAILURE.

How do you know whether your program works? Try this:

```
bash-3.2$ ./check_args a
bash-3.2$ echo $?
1
bash-3.2$ ./check_args a b
bash-3.2$ echo $?
0
bash-3.2$
```

In the shell, `$?` means *the exit code of the last program I ran*.

Problem 3: Parsing Arguments

The next two problems use the same application program but a different implementation of one module. In both cases, your application will be in `args.c`, and the application *should not change* for the next problem. Essentially, you're writing the client code for the module in this problem and the implementation code in the next problem.

To build the program for this problem, use `make args_prob3`, which will make a program named `args_prob3`. This will link your application with our solution for `str_to_int()` (see below).

Your task is to write a program named `args` that works as follows:

- It looks at the command line arguments. Unless and until it gets to a command line argument `--print`, it ignores the arguments.
- If it finds `--print`, then the next argument must be an integer numeral, say *n*. The program then prints the next *n* command line arguments, one per line.
- It then continues with command line arguments looking for more `--print` commands.

For example:

```
bash-3.2$ ./args a b --print 2 c d e --print 1 f g
c
d
f
bash-3.2$ ./args a b --print 2 c d e --print 0 f g
c
d
bash-3.2$
```

If `--print` is not followed by a valid integer numeral or if the number specified would be too large (go beyond the last argument), the program must print `Invalid` on `stderr` and exit with `EXIT_FAILURE`. If it succeeds, it should exit with a success indication.

To do this problem, you will have to use

- `fprintf()` to print to `stderr` (which is like `cerr` from C++).
- `strncmp()` to compare strings (command line arguments and `--print`). This function does *not* return a boolean value. See the manual page.
- a `str_to_int()` function to convert a numeral in a string to an integer value. We have provided you with `str_to_int.h` and `str_to_int_solution.o`, which contains our definition of the function. You will have to link with this `.o` file. The contract for `str_to_int()` is in the `.h` file.

Note that `str_to_int()` uses a *call by reference (CBR)* parameter. This is common in C, especially if a function needs to effectively return more than one thing. In this case, the function returns a status value that indicates success or failure. In the case of success, it needs to give the result value back to the caller. One way to do this is with structs, but it's also very common to use a CBR parameter — such parameters are sometimes called *output parameters.* Here, the parmater named `result` is really an output of the function.

C does not have C++-style reference parameters, so we get call by reference by giving the function a pointer to the location where we'd like to put the result value. There is no need do dynamically allocate any memory to call the function.

After you have either succeeded or gotten stuck, draw a stack diagram for this function call. Show your stack diagram to a member of the course staff and explain it.

Problem 4: Converting from Strings to Integers

In the problem above, we gave you a pre-compiled implementation of `str_to_int()`. For this problem, you must implement the version. Write a C module, `str_to_int.c` that contains a definition of this function.

To build the program for this problem, use `make args_prob4`, which will make a program named `args_prob4`. This will link your `args` program with your own solution to `str_to_int()`.

You must **NOT** use `atoi()`, and you should never use that function (at least not in code you give to others or ever expect to use in the future). You should try to figure out why. In particular, it is impossible to meet the specification in the `.h` file if you use it!

There are two excellent, standard methods. One uses `sscanf()` and the other uses `strtol`. Either is acceptable.

Problem 5: Call By Reference (again)

Write a program called `oldest_person` that takes a sequence of one or more name/age pairs on the command line and prints out the olderst person's name and age. For example:

```
bash-3.2$ ./oldest_person Maya 9 Raina 15 Mark 53 January 0
Mark:  53
bash-3.2$
```

If the command line is malformed in any way, print `Invalid` on `stderr` and exit with `EXIT_FAILURE`. Negative ages are considered malformed. Your program must define and use the following function:

```
int oldest_person(char *arglist[], int len, char **oldest_name_p);
```

The function takes the list of command line arguments *excluding the program name* and returns the age of the oldest person and places a pointer to a string giving the oldest person's name in the place pointed to by `oldest_name_p`, that is, `olest_name_p` is a CBR string parameter.

Submitting your work

When you're done, please use `provide` to submit your work. We'll use a slightly different procedure for future assignments, but for now:

```
provide comp40 under-the-c++ *.c *.h Makefile
```