

Assignment #1—Simple C++

Parts of this handout were written by Julie Zelenski.

Due: Monday, January 23

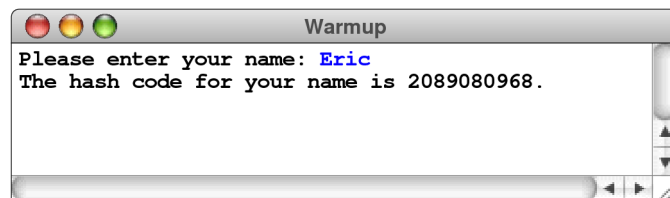
Part 1. Get your C++ compiler working

Your first task is to set up your C++ compiler. If you're using the machines in Stanford's public clusters, you don't need to do anything special to install the software. If you're using your own machine, you should consult one of the following handouts, depending on the type of machine you have:

- Handout #9M. Downloading XCode on the Macintosh
- Handout #9P. Downloading Visual Studio for Windows

Once you have the compiler ready, go to the assignments section of the web site and download the starter file for the type of machine you are using. For either platform, the **Assignment1** folder contains six separate project folders: one for this warmup problem and one for each of the five problems in Part 2 of the assignment. Open the project file in the folder named **0-Warmup**. Your mission in Part 1 of the assignment is simply to get this program running. The source file we give you is a complete C++ program—so complete, in fact, that it comes complete with two bugs. The errors are not difficult to track down (in fact, we'll tell you that one is a missing declaration and the other is a missing `#include` statement). This task is designed to give you experience with the way errors are reported by the compiler and what it takes to fix them.

Once you fix the errors, compile and run the program. When the program executes, it will ask for your name. Enter your name and it will print out a “hash code” (a number) generated for that name. We'll talk later in the class about hash codes and what they are used for, but for now just run the program, enter your name, and record the hash code. You'll email us this number. A sample run of the program is shown below:



```
Warmup
Please enter your name: Eric
The hash code for your name is 2089080968.
```

Once you've gotten your hash code, we want you to e-mail it to your section leader and introduce yourself. You don't yet know your section assignment, but will receive it via email after signups close, so hold on to your e-mail until then. You should also **cc** me on the e-mail (eroberts@stanford.edu) so I can meet you as well.

Here's the information to include in your e-mail:

1. Your name and the hash code that was generated for it by the program
2. Your year and major
3. When you took 106A (or equivalent course) and how you feel it went for you
4. What you are most looking forward to about 106B
5. What you are least looking forward to about 106B
6. Any suggestions that you think might help you learn and master the course material

Part 2. Simple C++ problems

Most of the assignments in this course are single programs of a substantial size. To get you started, however, the first assignment is a series of four short problems that are designed to get you used to using C++ and to introduce the idea of functional recursion. None of these problems require more than a page of code to complete; most can be solved in just a few lines.

Problem 1 (Chapter 2, exercise 15, page 123)

Heads. . . .

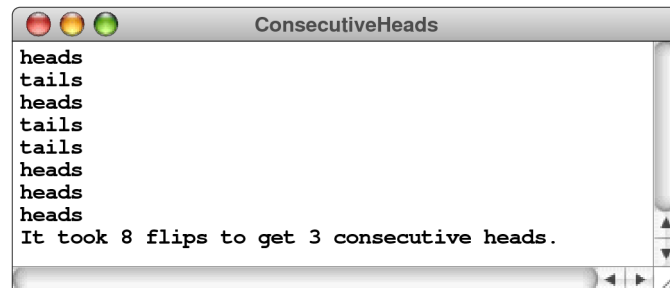
Heads. . . .

Heads. . . .

A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.

—Tom Stoppard, *Rosencrantz and Guildenstern Are Dead*, 1967

Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:



Problem 2 (Chapter 3, exercise 16, page 152)

Most people—at least those in English-speaking countries—have played the Pig Latin game at some point in their lives. There are, however, other invented “languages” in which words are created using some simple transformation of English. One such language is called *Obenglobish*, in which words are created by adding the letters *ob* before the vowels (*a*, *e*, *i*, *o*, and *u*) in an English word. For example, under this rule, the word *english* gets the letters *ob* added before the *e* and the *i* to form *obenglobish*, which is how the language gets its name.

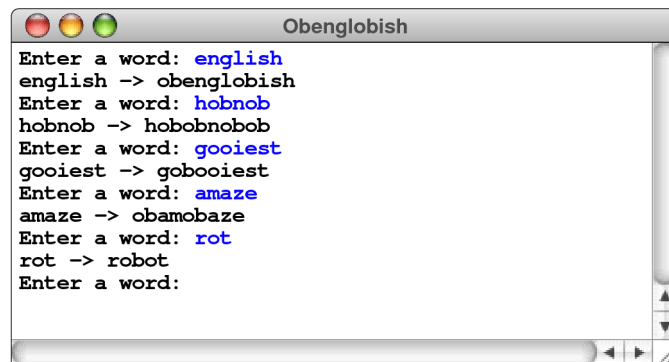
In official Obenglobish, the `ob` characters are added only before vowels that are pronounced, which means that a word like *game* would become *gobame* rather than *gobamobe* because the final *e* is silent. While it is impossible to implement this rule perfectly, you can do a pretty good job by adopting the rule that the *ob* should be added before every vowel in the English word *except*

- Vowels that follow other vowels
- An *e* that occurs at the end of the word

Write a function `obenglobish` that takes an English word and returns its Obenglobish equivalent, using the translation rule given above. For example, if you used your function with the main program

```
int main() {
    while (true) {
        string word = getLine("Enter a word: ");
        if (word == "") break;
        string trans = obenglobish(word);
        cout << word << " -> " << trans << endl;
    }
    return 0;
}
```

you should be able to generate the following sample run:



```
Obenglobish
Enter a word: english
english -> obenglobish
Enter a word: hobnob
hobnob -> hobobnobob
Enter a word: gooiest
gooiest -> gobooiest
Enter a word: amaze
amaze -> obamobaze
Enter a word: rot
rot -> robot
Enter a word:
```

Problem 3 (Chapter 7, exercise 9, page 349)

As you know from Chapter 2, the mathematical combinations function $c(n, k)$ is usually defined in terms of factorials, as follows:

$$c(n, k) = \frac{n!}{k! \times (n - k)!}$$

The values of $c(n, k)$ can also be arranged geometrically to form a triangle in which n increases as you move down the triangle and k increases as you move from left to right. The resulting structure, which is called *Pascal's Triangle* after the French mathematician Blaise Pascal, is arranged like this:

```

c(0, 0)
c(1, 0) c(1, 1)
c(2, 0) c(2, 1) c(2, 2)
c(3, 0) c(3, 1) c(3, 2) c(3, 3)
c(4, 0) c(4, 1) c(4, 2) c(4, 3) c(4, 4)

```

Pascal’s Triangle has the interesting property that every entry is the sum of the two entries above it, except along the left and right edges, where the values are always 1. Consider, for example, the circled entry in the following display of Pascal’s Triangle:

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1

```

This entry, which corresponds to $c(6, 2)$, is the sum of the two entries—5 and 10—that appear above it to either side. Use this fact to write a recursive implementation of the $c(n, k)$ function that uses no loops, no multiplication, and no calls to `Fact`.

Write a simple test program to demonstrate that your combinations function works. If you want an additional challenge, write a program that uses $c(n, k)$ to display the first ten rows of Pascal’s Triangle.

Problem 4 (Implementing numeric conversion)

The `strlib.h` interface exports the following methods for converting between integers and strings:

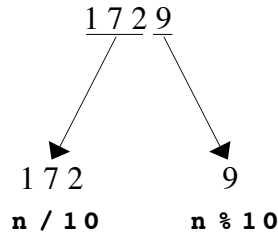
```

string integerToString(int n);
int stringToInteger(string str);

```

The first function converts an integer into its representation as a string of decimal digits, so that, for example, `integerToString(1729)` should return the string "1729". The second converts in the opposite direction so that calling `stringToInteger("-42")` should return the integer `-42`.

Your job in this problem is to write the functions `intToString` and `stringToInt` (the names have been shortened to avoid having your implementation conflict with the library version) that do the same thing as their `strlib.h` counterparts but use a recursive implementation. Fortunately, these functions have a natural recursive structure because it is easy to break an integer down into two components using division by 10. This decomposition is discussed on page 42 in the discussion of the `digitSum` function. The integer 1729, for example, breaks down into two pieces, as follows:



If you use recursion to convert the first part to a **string** and then append the character value corresponding to the final digit, you will get the **string** representing the integer as a whole.

As you work through this problem, you should keep the following points in mind:

- Your solution should operate recursively and should use no iterative constructs such as **for** or **while**. It is also inappropriate to call the provided **integerToString** function or any other library function that does numeric conversion.
- The value that you get when you compute **n % 10** is an integer, and not a character. To convert this integer to its character equivalent you have to add the ASCII code for the character '0' and then cast that value to a **char**. If you then need to convert that character to a one-character string, you can concatenate it with **string()**. (The Java trick of concatenating with "" doesn't work because string constants are C-style strings.)
- You should think carefully about what the simple cases need to be. In particular, you should make sure that calling **intToString(0)** returns "0" and not the empty string. This fact may require you to add special code to handle this case.
- Your implementation should allow **n** to be negative, as illustrated by the earlier example in which **stringToInt("-42")** returns -42. Again, implementing these functions for negative numbers will probably require adding special-case code.