6.096 Lab 1

Due: 7 January 12:30:00

September 20, 2011

1 Additional Material

1.1 Constants

A constant is an expressions with a fixed value. Kinds of constants:

- Literals: used to express particular values within the source code; can be integers (ex: 1, −0, -17), floating points (ex: 3.1415926535897, 1., 6.096, 3), characters (ex: ‘a’, ‘ ’, ‘\n’), strings (ex: “6.096”, “a”), booleans (ex: true, false)

- Defined constants (`#define`): user-defined named constants that do not require memory-consuming variables. When the preprocessor sees the `#define`, it will replace every subsequent occurrance of the identifier in the source code.

```
#include <iostream>

using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main()
{
    double r = 5.0;
    double circle;

    circle = 2 * PI * r; // circle = 2 * 3.14159 * r;
    cout << circle << NEWLINE; // cout << circle << '\n';

    return 0;
}
```

- Declared constants (`const`): user defined constants with `const` prefix with a specific type that are declared the same way as variables. The value of a `const` variable cannot be modified.

```
const int pathwidth = 100;
// pathwidth = 2; this will cause a compiler error!
const char tabulator = '\t';
cout << "tabulator =" << tabulator << '\n';
```

1.2 L-values, R-values

lvalue is short for "left hand side value" (of an assignment).

Examples of **non**-lvalue expressions:

- `3+3 // you can't assign 3+3 to something`
- `"str" // the literal "str" can't take on another value`
- `const int i = 3 // can't change the value of const variable`

Examples of lvalue expressions:

- `int var // var is an lvalue because we can assign it with some value`
- `float x`

rvalue is short for "right hand side value" because rvalues can appear on the right hand side of an assignment. Anything with a well-defined value can be an rvalue, including an assigment: `(x = 5)` can be used as an rvalue whose value is 5, e.g. `y = (x=5);`.

1.3 Increment, Decrement operators (++, --)

`a++` and `++a` are shorthand for `a = a + 1` with a big warning sign:

- `++a` will increment `a` and then return the value (so it will return one greater than the original value)
- `a++` will return the current value and then increment
- `--a` will decrement `a` and then return the value (so it will return one less than the original value)
- `a--` will return the current value and then decrement

```
// this code outputs 0 to 9
for(int i = 0; i < 10;)
{
    cout << i++ << "\n";
}

```

```
// this code outputs 1 to 10
for(int i = 0; i < 10;)
{
    cout << ++i << "\n";
}
```

1.4 Assignment operators

Assignment ops include `+=`, `-=`, `*=`, `/=`, `%=`, etc.; `a += 5` is equivalent to `a = a + 5`, etc.

1.5 Type Conversions

Used for changing between data types. Also called "casts." Type conversions are implicit when changing from smaller data type to a bigger data type or data type of same size (e.g. `float` to `double` or `int` to `float`). Type conversions usually must be explicitly stated when changing from bigger datatype to smaller datatype or when there could be a loss of accuracy (e.g. `int` to `short` or `float` to `int`), but implicitly converting from a `double` to an `int` will not generate a compiler error (the compiler will give a warning, though).

```
int x = (int)5.0; // float should be explicitly "cast" to int
short s = 3;
long l = s; // does not need explicit cast, but
            // long l = (long)s is also valid
float y = s + 3.4; // compiler implicitly converts s
                   // to float for addition
```

1.6 Operator precedence

Like arithmetic, C++ operators have a set order by which they are evaluated. The table of operators and their precedence is listed in the following table:

1	() [] -> . ::	Grouping, scope, array/member access
2	! ˜ * & sizeof (type cast) ++ −	(most) unary operations, sizeof and typecasts
3	* / %	Multiplication, division, modulo
4	+ -	Addition and subtraction
5	<< >>	Bitwise left and right shift
6	< <= > >=	Comparisons: less than, etc.
7	== !=	Comparisons: equal and not equal
8	&	Bitwise AND
9	ˆ	Bitwise exclusive OR
10	|	Bitwise inclusive (normal) OR
11	&&	Logical AND
12	||	Logical OR
13	?:	Conditional expression (ternary operator)
14	= += -= *= /= %=, etc.	Assignment operators
15	,	Comma operator

1.7 Ternary operator (?:)

An operator that takes three arguments and defines a conditional statement.

```
if(a > b)
    result = x;
else
    result = y;
```

is equivalent to

```
result = a > b ? x : y;
```

1.8 switch statement

A type of selection control statement. Its purpose is to allow the value of a variable or expression to control the flow of program execution via multiple possible branches. Omitting `break` keywords to causes the program execution to "fall through" from one block to the next, a trick used extensively. In the example below, if $n = 2$, the fifth case statement (line 10) will match the value of n, so the next line outputs "n is an even number." Execution then continues through the next three case statements and to the next line, which outputs "n is a prime number." This is a classic example of omitting the `break` line to allow for fall through. The `break` line after a case block causes the switch statement to conclude. The `default` block (line 21) is executed if no other cases match, in this case producing an error message if n has multiple digits or is negative. The `default` block is optional in a `switch` statement.

```
1 switch(n) {
  case 0:
    cout << "You typed zero.\n";
    break;
  case 1:
  case 4:
  case 9:
    cout << "n is a perfect square.\n";
    break;
  case 2:
    cout << "n is an even number.\n";
  case 3:
  case 5:
  case 7:
    cout << "n is a prime number.\n";
    break;
  case 6:
  case 8:
    cout << "n is an even number.\n";
    break;
  default:
    cout << "Only single-digit positive numbers are allowed.\n";
    break;
}
```

1.9 break

Used for breaking out of a loop or switch statement.

```
// outputs first 10 positive integers
int i = 1;
while(true)
{
    if(i > 10)
        break;
    cout << i << "\n";
    ++i;
}
```

1.10 continue

Used for skipping the rest of a loop body and continuing to the next iteration.

```
// print out even numbers in range 1 to 10
for(int i = 0; i <= 10; ++i)
{
    if(i % 2 != 0)
        continue; // skips all odd numbers
    cout << i << "\n";
}
```

1.11 References

- Wikipedia: http://en.wikipedia.org/
- Cplusplus: http://www.cplusplus.com/

2 “Hello, World!”

This section is about writing the canonical “Hello, World!” program and its derivatives. Now is a good time to get used to your development environment. Submit each program in a separate source file named ⟨section⟩.⟨subsection⟩.cpp.

2.1 Hello World I

Write a program that outputs “Hello, World!” by printing a `const char *` with value “Hello, World!”.

2.2 Hello World II

Write a program that outputs “Hello, World!” n times (where n is a nonnegative integer that the user will input) with:

- a `for` loop.
- a `while` loop.
- a `do...while` loop.

3 More Programs

Now that you have had some practice working in your development environment, here are some more challenging tasks. Again, submit your work (source code and answers to questions) in an appropriately named text file.

3.1 Scope

For these questions, you are encouraged to use a computer.

1. Below is a sample program. Use it to answer the following question: What happens if we declare the same name twice within a block, giving it two different meanings?

```cpp
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    int x, y, z;
    char myDouble = '5';
    char arg1 = 'A';
    cout << arg1 << "\n";
    return 0;
}
```

   Hints: Did your program compile? If so, what does it print? If not, what error message do you get?

2. Below is a sample program. Use it to answer the following question: What happens if we declare an identifier in a block, and then redeclare that same identifier in a block nested within that block?

```cpp
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    {
        char arg1 = 'A';
        cout << arg1 << "\n";
    }
    return 0;
}
```

Hints: Did your program compile? If it does, what does the program output? If not, what error message does it produce?

3. Below is a sample program. Use it to answer the following question: Suppose an identifier has two different declarations, one in an outer block and one in a nested inner block. If the name is accessed within the inner block, which declaration is used?

```
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    {
        char arg1 = 'A';
        cout << arg1 << "\n";
    }
    return 0;
}
```

4. Below is a sample program. Use it to answer the following question: Suppose an identifier has two different declarations, one in an outer block and one in a nested inner block. If the name is accessed within the outer block, but after the inner block, which declaration is used?

```
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    {
        char arg1 = 'A';
    }
    cout << arg1 << "\n";
    return 0;
}
```

5. Below is a sample program that will not compile. Why not? By moving which line can we get the code to compile?

```
1 using namespace std;
2
3 int main()
4 {
5     cout << "Hello, World!\n";
6     return 0;
7 }
8
9 #include <iostream>
```

3.2 Basic Statistics

Given a list of N integers, find its mean (as a `double`), maximum value, minimum value, and range. Your program will first ask for N, the number of integers in the list, which the user will input. Then the user will input N more numbers.

Here is a sample input sequence:

```
3 <-- N
2
1
3
```

Three numbers are given: 2, 1, 3. The output should be as follows:

```
Mean: 2
Max: 3
Min: 1
Range: 2
```

3.3 Prime Numbers

Write a program to read a number N from the user and then find the first N primes. A prime number is a number that only has two divisors, one and itself.

3.4 Multiples of numbers

3.4.1 Ternary operator

Write a program that loops indefinitely. In each iteration of the loop, read in an integer N (declared as an `int`) that is inputted by a user, output $\frac{N}{5}$ if N is nonnegative and divisible by 5, and -1 otherwise. Use the ternary operator (`?:`) to accomplish this. (*Hint:* the modulus operator may be useful.)

3.4.2 `continue`

Modify the code from 3.4.1 so that if the condition fails, nothing is printed. Use an `if` and a `continue` command (instead of the ternary operator) to accomplish this.

3.4.3 `break`

Modify the code from 3.4.2 to let the user break out of the loop by entering -1 or any negative number. Before the program exits, output the string “Goodbye!”.

3.5 What does this program do?

Do these problems without the use of a computer!

1. What does this snippet do? Try doing out a few examples with small numbers on paper if you're stuck. (*Hint:* Think about numbers in binary notation – in base 2. How would you express a number as a sum of powers of 2? You may also find it useful to note that multiplying by 2^n is equivalent to multiplying by 2 n times. You should also keep in mind the distributive property of multiplication: $a(x + y) = ax + ay$.)

```
// bob and dole are integers
int accumulator = 0;
while(true)
{
  if(dole == 0) break;
  accumulator += ((dole % 2 == 1) ? bob : 0);
  dole /= 2;
  bob *= 2;
}
cout << accumulator << "\n";
```

2. What does this program do? What would the operating system assume about the program's execution?

```
#define O 1 // That's an oh, not a zero
int main()
{
    return O;
}
```

3. What does this program do?

```
// N is a nonnegative integer
double acc = 0;
for(int i = 1; i <= N; ++i)
{
    double term = (1.0/i);
    acc += term * term;
    for(int j = 1; j < i; ++j)
    {
        acc *= -1;
    }
}
cout << acc << "\n";
```

4 Factorials Gone Wrong

This section focuses on debugging programs. We will start off with a simple factorial program and do a step by step troubleshooting routine.

4.1 Writing the factorial program

Here is the code for a factorial program. Copy it into your IDE and verify that it compiles.

```cpp
#include <iostream>

using namespace std;

int main()
{
    short number;
    cout << "Enter a number: ";
    cin >> number;

    cout << "The factorial of " << number << " is ";
    int accumulator = 1;
    for(; number > 0; accumulator *= number--);
    cout << accumulator << ".\n";

    return 0;
}
```

What do you get when you enter the following values: 0, 1, 2, 9, 10?

4.2 Breaking the program

Run the program and enter -1. What happens? How can you change the code such that it won't exhibit this behavior?

4.3 Breaking the program II

Try entering some larger numbers. What is the minimum number for which your modified program from 4.2 stops working properly? (You shouldn't have to go past about 25. You may find Google's built-in calculator useful in checking factorials.) Can you explain what has happened?

4.4 Rewriting Factorial

Modify the given code such that negative inputs do not break the program and the smallest number that broke the program before now works. Do not hardcode answers.

4.5 Rewriting Factorial II

Since we know that only a small number of inputs produce valid outputs, we can alternatively hardcode the factorials of these inputs. Rewrite the program from the previous part ("Rewriting Factorial") using a switch statement to demonstrate for inputs up to 10 how you would do this. (Of course, the code for inputs above 10 would basically be the same, but you do not need to go through the work of finding all those large factorials.)

4.6 Further testing

Are there any other inputs we have to consider? Why or why not?

Lab 1 Solutions

6.096 Staff

2 “Hello, World” (10 points)

2.1 Hello World I (1 point)

```cpp
#include <iostream>

using namespace std;

int main()
{
    const char* str = "Hello, World!";
    cout << str << "\n";
    return 0;
}
```

2.2 Hello World II (9 points)

1. for loop: (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    for(; N-- > 0;)
    {
        cout << "Hello, World!\n";
    }
    return 0;
}
```

2. `while` loop: (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    while(N-- > 0)
    {
        cout << "Hello, World!\n"
    }
    return 0;
}
```

3. `do...while` loop: (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    do
    {
        cout << "Hello, World!\n";
    }
    while(--N > 0);
    return 0;
}
```

3 More Programs (50 points)

3.1 Scope (10 points; 2 points each)

1. We cannot declare the same name within a block because it will generate a compiler error.

2. The program compiles.

3. The declaration in the inner block is used.

4. The declaration in the outer block is used.

5. The code will not compile because the function `cout` has not yet been defined. If we move `#include <iostream>` to the top, then the code will compile.

3.2 Basic Statistics (10 points)

```
#include <iostream>

using namespace std;

int main()
{
    int N;
    cout << "Enter N: ";
    cin >> N;
    int acc = 0;

    // handle the first number separately
    cin >> acc;
    int minVal = acc;
    int maxVal = acc;

    // then process the rest of the input
    for(int i = 1; i < N; ++i)
    {
        int a;
        cin >> a;
        acc += a;
        if(a < minVal)
        {
            minVal = a;
        }
        if(a > maxVal)
        {
```

```cpp
                maxVal = a;
            }
        }

        cout << "Mean: " << (double)acc/N << "\n";
        cout << "Max: " << maxVal << "\n";
        cout << "Min: " << minVal << "\n";
        cout << "Range: " << (maxVal - minVal) << "\n";

        return 0;
}
```

3.3 Prime Numbers (10 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    for(int i = 2; N > 0; ++i)
    {
        bool isPrime = true;
        for(int j = 2; j < i; ++j)
        {
            if(i % j == 0)
            {
                isPrime = false;
                break;
            }
        }
        if(isPrime)
        {
            --N;
            cout << i << "\n";
        }
    }
    return 0;
}
```

3.4 Multiples of numbers (10 points)

3.4.1 Ternary operator (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    while(1)
    {
        int N;
        cin >> N;
        cout << ((N % 5 == 0 && N >= 0) ? N/5 : -1) << "\n";
    }
    return 0;
}
```

3.4.2 continue (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    while(1)
    {
        int N;
        cin >> N;
        if(N % 5 > 0)
        {
            cout << "-1\n";
            continue;
        }
        cout << N/5 << "\n";
    }
    return 0;
}
```

3.4.3 break (3 points)

```cpp
#include <iostream>

using namespace std;

int main()
{
    while(1)
    {
        int N;
        cin >> N;
        if(N % 5 > 0)
        {
            cout << "-1\n";
            continue;
        }
        if(N == -1)
        {
            break;
        }
        cout << N/5 << "\n";
    }
    cout << "Goodbye!\n";
    return 0;
}
```

1 extra point if all three parts are correct.

3.5 What does this program do? (10 points)

1. Russian peasant multiplication of `bob` and `dole`. (5 points)
2. It returns 1 and exits. The operating system would assume that something went wrong. (2 points)
3. Evaluates the series

   Case $N \equiv 0,1 \mod 4$:
   $$\frac{1}{1^2}+\frac{1}{2^2}-\frac{1}{3^2}-\frac{1}{4^2}+\frac{1}{5^2}+\dots\frac{1}{N^2}$$
   Case $N \equiv 2,3 \mod 4$:
   $$-\frac{1}{1^2}-\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}-\frac{1}{5^2}-\dots\frac{1}{N^2}$$
   The sign of the $\frac{1}{N^2}$ term is positive if $N \equiv 0,1 \mod 4$ and negative if $N \equiv 2,3 \mod 4$. (3 points)

4 Factorials Gone Wrong (40 points)

4.1 Writing the factorial program (5 points)

0: 1; 1: 1; 2: 2; 9: 362880; 10: 3628800

4.2 Breaking the program (5 points)

If -1 is entered, the program will output 1, which is incorrect because the factorial function (or Gamma function) is not defined for negative numbers.

4.3 Breaking the program II (5 points)

The number at which the program fails depends on the system architecture; 64-bit systems often fail at 17. We accepted any answer around that value.

4.4 Rewriting Factorial (10 points)

```cpp
#include <iostream>

using namespace std;

long long accumulator = 1;

int main()
{
    int number;
    cout << "Enter a number: ";
    cin >> number;
    if(number < 0)
    {
        cout << "No negative numbers allowed!\n";
        return 1;
    }
    if(number > 20)
    {
        cout << "Program will not produce correct result!\n";
    }
    for(; number > 0; accumulator *= number--);
    cout << "The factorial of " << number << " is " << accumulator
       << ".\n";
    return 0;
}
```

4.5 Rewriting Factorial II (10 points)

```
#include <iostream>

using namespace std;

int main()
{
    int number;
    cout << "Enter a number: ";
    cin >> number;
    switch(number)
    {
        case 0:
        case 1:
            cout << "1\n";
            break;
        case 2:
            cout << "2\n";
            break;
        case 3:
            cout << "6\n";
            break;
        case 4:
            cout << "24\n";
            break;
        case 5:
            cout << "120\n";
            break;
        case 6:
            cout << "720\n";
            break;
        case 7:
            cout << "5040\n";
            break;
        case 8:
            cout << "40320\n";
            break;
        case 9:
            cout << "362880\n";
            break;
        case 10:
            cout << "3628800\n";
            break;
        default:
            cout << "Input not supported!\n";
            break;
    }
    return 0;
}
```

4.6 Further Testing (5 points)

There are no other inputs we have to consider. Since we are dealing with the short data type for storing our input number, we can separate our analysis into two cases: negative integers and nonnegative integers. We have modified our program to deal with negative numbers in 4.2 and have modified our program to deal with large positive numbers in 4.4. Therefore we do not have to consider any other input cases.

6.096 Problem Set 2

Due: 14 January 12:30:00

For this problem set, you should be able to put all your code for each section into a single source/text file (though you may have to comment out earlier parts of your solution to test later parts). Clearly mark each subsection with comments, and submit a .zip file containing all your source/text files.

1 Additional Material

1.1 Functions

1.1.1 Default Arguments

Say you have a function with 1 argument, but that argument is usually the same. For instance, say we want a function that prints a message n times, but most of the time it will only need to print it once:

```
void printNTimes(char *msg, int n) {
    for( int i = 0; i < n; ++i) {
        cout << msg;
    }
}
```

Rather than writing `printNTimes("Some message", 1);` every time, C++ allows *default arguments* to be defined for a function:

```
void printNTimes(char *msg, int n = 1) {
    for( int i = 0; i < n; ++i) {
        cout << msg;
    }
}
```

Declaring the function argument as `int n = 1` allows us to call the function with `printNTimes("Some message");`. The compiler automatically inserts 1 as the second argument.

You may have multiple default arguments for a function:

```
void printNTimes(char *msg = "\n", int n = 1) {
    for( int i = 0; i < n; ++i) {
        cout << msg;
    }
}
```

Now, to print one newline, we can simply write `printNTimes();`. However, C++ does not allow skipping arguments, so we could not print k newlines by writing `printNTimes(k);`. To do that, we'd need to say `printNTimes("\n", k);`.

1.1.2 Constant Arguments

It's often useful to specify that arguments to a function should be treated as constants. As with regular variables, we can declare function arguments to be `const`:

```cpp
void print(const int n) {
  cout << n;
}
```

This is particularly useful when we are passing values by reference to a function, but don't want to allow the function to make any changes to the original value:

```cpp
void print(const long &x) { // Pass-by-reference avoids overhead
                            // of copying large number
    cout << x;
}

int main() {
    long x = 234923592;
    print(x); // We are guaranteed that x
              // will not be changed by this
    return 0;
}
```

In general, if you know a value shouldn't be changing (particularly a function argument), you should declare it `const`. That way, the compiler can catch you if you messed up and tried to change it somewhere.

1.1.3 Random Number Generation Functions

The C++ standard libraries include the `rand()` function for generating random numbers between 0 and `RAND_MAX` (an integer constant defined by the compiler). These numbers are not truly random; they are a random-seeming but deterministic sequence based on a particular "seed" number. To make sure we don't keep getting the same random-number sequence, we generally use the current time as the seed number. Here is an example of how this is done:

```cpp
#include <iostream>
#include <cstdlib> // C standard library -
                   // defines rand(), srand(), RAND_MAX
```

```
#include <ctime>   // C time functions - defines time()
int main() {
    srand( time(0) ); // Set the seed;
                      // time(0) returns current time as a number
    int randNum = rand();
    std::cout << "A random number: " << randNum << endl;
    return 0;
}
```

1.2 Pointers

1.2.1 Pointers to Pointers

We can have pointers to any type, including pointers to pointers. This is commonly used in C (and less commonly in C++) to allow functions to set the values of pointers in their calling functions. For example:

```
void setString(char **strPtr) {
    int x;
    cin >> x;
    if(x < 0)
        *strPtr = "Negative!";
    else
        *strPtr = "Nonnegative!";
}

int main() {
    char *str;
    setString(&str);
    cout << str; // String has been set by setString
    return 0;
}
```

1.2.2 Returning Pointers

When you declare a local variable within a function, that variable goes out of scope when the function exits: the memory allocated to it is reclaimed by the operating system, and anything that was stored in that memory may be cleared. It therefore usually generates a runtime error to return a pointer to a local variable:

```
int *getRandNumPtr() {
    int x = rand();
    return &x;
}

```

```
 6 int main() {
 7     int *randNumPtr = getRandNumPtr();
 8     cout << *randNumPtr; // ERROR
 9     return 0;
10 }
```

Line 8 will likely crash the program or print a strange value, since it is trying to access memory that is no longer in use – `x` from `getRandNumPtr` has been deallocated.

1.3 Arrays and Pointers

1.3.1 Arrays of Pointers

Arrays can contain any type of value, including pointers. One common application of this is arrays of strings, i.e., arrays of `char *`'s. For instance:

```
const char *suitNames[] = {"Clubs", "Diamonds", "Spades", "Clubs"};
cout << "Enter a suit number (1-4): ";
unsigned int suitNum;
cin >> suitNum;
if(suitNum <= 3)
    cout << suitNames[suitNum - 1];
```

1.3.2 Pointers to Array Elements

It is important to note that arrays in C++ are pointers to continuous regions in memory. Therefore the following code is valid:

```
const int ARRAY_LEN = 100;
int arr[ARRAY_LEN];
int *p = arr;
int *q = &arr[0];
```

Now `p` and `q` point to exactly the same location as `arr` (ie. `arr[0]`), and `p`, `q` and `arr` can be used interchangeably. You can also make a pointer to some element in the middle of an array (similarly to `q`):

```
int *z = &arr[10];
```

1.4 Global Scope

We discussed in lecture how variables can be declared at *global scope* or *file scope* – if a variable is declared outside of any function, it can be used anywhere in the file. For anything besides global constants such as error codes or fixed array sizes, this is usually a bad idea; if you need to access the same variable from multiple functions, most often you should simply pass the variable around as an argument between the functions. Avoid global variables when you can.

2 A Simple Function

What would the following program print out? (Answer without using a computer.)

```
void f(const int a = 5)
{
    std::cout << a*2 << "\n";
}

int a = 123;
int main()
{
    f(1);
    f(a);
    int b = 3;
    f(b);
    int a = 4;
    f(a);
    f();
}
```

3 Fix the Function

Identify the errors in the following programs, and explain how you would correct them to make them do what they were apparently meant to do.

3.1

```
#include <iostream>

int main() {
    printNum(35);
    return 0;
}

void printNum(int number) { std::cout << number; }
```

(Give two ways to fix this code.)

3.2

```cpp
#include <iostream>

void printNum() { std::cout << number; };

int main() {
    int number = 35;
    printNum(number);
    return 0;
}
```

(Give two ways to fix this code. Indicate which is preferable and why.)

3.3

```cpp
#include <iostream>

void doubleNumber(int num) {num = num * 2;}

int main() {
    int num = 35;
    doubleNumber(num);
    std::cout << num; // Should print 70
    return 0;
}
```

(Changing the return type of `doubleNumber` is not a valid solution.)

3.4

```cpp
#include <iostream>
#include <cstdlib> // contains some math functions

int difference(const int x, const int y) {
    int diff = abs(x - y); // abs(n) returns absolute value of n
}

int main() {
    std::cout << difference(24, 1238);
    return 0;
}
```

3.5

```cpp
#include <iostream>

int sum(const int x, const int y) {
    return x + y;
}

int main() {
    std::cout << sum(1, 2, 3); // Should print 6
    return 0;
}
```

3.6

```cpp
#include <iostream>
const int ARRAY_LEN = 10;

int main() {
    int arr[ARRAY_LEN] = {10}; // Note implicit initialization of
                               // other elements
    int *xPtr = arr, yPtr = arr + ARRAY_LEN - 1;
    std::cout << *xPtr << ' ' << *yPtr; // Should output 10 0
    return 0;
}
```

4 Sums

Make sure to use `const` arguments where appropriate throughout this problem (and all the others).

4.1

Write a single `sum` function that returns the sum of two integers. Also write the equivalent function for taking the sum of two `double`s.

4.2

Explain why, given your functions from part 1, `sum(1, 10.0)` is a syntax error. *(Hint:* Think about promotion and demotion – the conversion of arguments between types in a function call. Remember that the compiler converts between numerical types for you if necessary.) [1 point]

4.3

Write 2 more functions such that you can find the sum of anywhere between 2 and 4 integers by writing `sum(num1, num2, ...)`.

4.4

Now write just one function that, using default arguments, allows you to take the sum of anywhere between 2 and 4 integers. What would happen if you put both this definition and your 3-argument function from part 3 into the same file, and called `sum(3, 5, 7)`? Why?

4.5

Write a single `sum` function capable of handling an arbitrary number of integers. It should take two arguments, include a loop, and return an integer. *(Hint: What data types can you use to represent an arbitrarily large set of integers in two arguments?)*

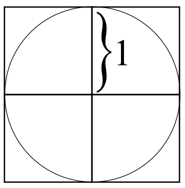
4.6

Now rewrite your function from 4.5 to use recursion instead of a loop. The function signature should not change. Thinking about pointer arithmetic may help you.

5 Calculating π

This problem is a bit tricky, but it's a good exercise in writing a program that actually does something neat. It will also familiarize you with using random numbers.

Using a "Monte Carlo" method – that is, a randomized simulation – we can compute a good approximation of π. Consider a circle of radius 1, centered on the origin and circumscribed by a square, like so:



Imagine that this is a dartboard and that you are tossing darts at it randomly. With enough darts, the ratio of darts in the circle to total darts thrown should be the ratio between the area of the circle (call it a) and the area of the square (4): $\frac{\text{total darts}}{\text{darts in circle}} = \frac{4}{a}$. We can use this ratio to calculate a, from which we can then find $\pi = \frac{a}{r^2}$.

We can simplify the math by only considering the first quadrant, calculating the ratio of the top right square's area to the area of the single quadrant. Thus, we will actually find $\frac{a}{4}$, and then compute $\pi = 4 \times \frac{\frac{a}{4}}{r^2}$.

We'll build a function step by step to do all this.

5.1

Define variables to store the x and y coordinates of a particular dart throw. Initialize them to random `double`s in the range [0, 1] (simulating one dart throw). *(Hint: remember that `rand()` returns a value in the range [0, RAND_MAX]; we just want to convert that value to some value in [0, 1].)*

5.2

Place your x and y declarations in a loop to simulate multiple dart throws. Assume you have a variable `n` indicating how many throws to simulate. Maintain a count (declared outside the loop) of how many darts have ended up inside the circle. (You can check whether a dart is within a given radius with the Euclidean distance formula, $d^2 = x^2 + y^2$; you may find the `sqrt` function from the `<cmath>` header useful.)

5.3

Now use your loop to build a π-calculating function. The function should take one argument specifying the number of dart throws to run (`n` from part 2). It should return the decimal value of pi, using the technique outlined above. Be sure to name your function appropriately. Don't forget to initialize the random number generator with a seed. You should get pretty good results for around 5,000,000 dart throws.

6 Array Operations

6.1

Write a function `printArray` to print the contents of an integer array with the string `", "` between elements (but not after the last element). Your function should return nothing.

6.2

Write a `reverse` function that takes an integer array and its length as arguments. Your function should reverse the contents of the array, leaving the reversed values in the original array, and return nothing.

6.3

Assume the existence of two constants `WIDTH` and `LENGTH`. Write a function with the following signature:

```
void transpose(const int input[][LENGTH], int output[][WIDTH]);
```

Your function should transpose the `WIDTH` × `LENGTH` matrix in `input`, placing the `LENGTH` × `WIDTH` transposed matrix into `output`. (See http://en.wikipedia.org/wiki/Transpose#Examples for examples of what it means to transpose a matrix.)

6.4

What would happen if, instead of having `output` be an "out argument," we simply declared a new array within `transpose` and returned that array?

6.5

Rewrite your function from part 2 to use pointer-offset notation instead of array-subscript notation.

7 Pointers and Strings

7.1

Write a function that returns the length of a string (`char *`), excluding the final NULL character. It should not use any standard-library functions. You may use arithmetic and dereference operators, but not the indexing operator (`[]`).

7.2

Write a function that swaps two integer values using call-by-reference.

7.3

Rewrite your function from part 2 to use pointers instead of references.

7.4

Write a function similar to the one in part 3, but instead of swapping two values, it swaps two pointers to point to each other's values. Your function should work correctly for the following example invocation:

```
int x = 5, y = 6;
int *ptr1 = &x, *ptr2 = &y;
swap(&ptr1, &ptr2);
cout << *ptr1 << ' ' << *ptr2; // Prints "6 5"
```

7.5

Assume that the following variable declaration has already been made:

```
char *oddOrEven = "Never odd or even";
```

Write a single statement to accomplish each of the following tasks (assuming for each one that the previous ones have already been run). Make sure you understand what happens in each of them.

1. Create a pointer to a `char` value named `nthCharPtr` pointing to the 6th character of `oddOrEven` (remember that the first item has index 0). Use the indexing operator.

2. Using pointer arithmetic, update `nthCharPtr` to point to the 4th character in `oddOrEven`.

3. Print the value currently pointed to by `nthCharPtr`.

4. Create a new pointer to a pointer (a `char **`) named `pointerPtr` that points to `nthCharPtr`.

5. Print the value stored in `pointerPtr`.

6. Using `pointerPtr`, print the value pointed to by `nthCharPtr`.

7. Update `nthCharPtr` to point to the next character in `oddOrEven` (i.e. one character past the location it currently points to).

8. Using pointer arithmetic, print out how far away from the character currently pointed to by `nthCharPtr` is from the start of the string.

6.096 Lab 2

Due: 14 January 12:30:00

January 17, 2011

2 A Simple Function [5 points]

The program prints: 2 246 6 8 10

3 Fix the Function [1 point/fix, so 2 points for first 2]

3.1

Either declare a function prototype for `printNum` before `main`, or move the definition of `printNum` to before `main`.

3.2

Either add an `int` argument called `number` to `printNum` (preferable because it avoids use of global variables), or move the `int number` declaration to a global variable.

3.3

Make `num` a pass-by-reference parameter (i.e. add a `&` before its name).

3.4

Add a return statement to `difference` returning `diff` (or just scrap `diff` altogether and make the function `return abs(x-y);`).

3.5

Add a third argument to `sum`.

3.6

Add a `*` to make line 7 say `int *xPtr = arr, *yPtr = ...`.

4 Sums

In this problem and all others, half a point should be deducted for not using a `const` argument where it would have been appropriate.

4.1 [4 points]

```
int sum(const int x, const int y) {
    return x + y;
}

double sum(const double x, const double y) {
    return x + y;
}
```

4.2 [1 point]

Mixing and matching an `int` with a `double` makes it ambiguous which one you want to call. The compiler could either cast 1 to a `double` and call the `double` version of `sum`, or it could cast 10.0 to an `int` and call the `int` version.

4.3 [2+2 points]

```
int sum(const int x, const int y, const int z) {
    return x + y + z;
}

int sum(const int a, const int b, const int c, const int d) {
    return a + b + c + d;
}
```

4.4 [5 + 1 points]

```
int sum(const int a, const int b, const int c = 0, const int d = 0)
   {
    return a + b + c + d;
}
```

If the given definitions were included together, the compiler would give a compile error, since it cannot disambiguate between a call to the 3-argument function and a call to the 4-argument one with a default parameter.

4.5 [5 points]

```c
int sum(const int numbers[], const int numbersLen) {
    int sum = 0;
    for(int i = 0; i < numbersLen; ++i) {
        sum += numbers[i];
    }
    return sum;
}
```

4.6 [8 points]

```c
int sum(const int numbers[], const int numbersLen) {
    return numbersLen == 0 ? 0 : numbers[0] + sum(numbers + 1,
       numbersLen - 1);
}
```

5 Calculating π

5.1 [3 points]

```c
double x = rand() / (double)RAND_MAX, y = rand() / (double)RAND_MAX;
```

5.2 [6 points]

```c
int dartsInCircle = 0;
for(int i = 0; i < n; ++i) {
    double x = rand() / (double)RAND_MAX, y = rand() / (double)
       RAND_MAX;
    if( sqrt(x*x + y*y) < 1 ) {
        ++dartsInCircle;
    }
}
```

5.3 [6 points]

```c
double computePi(const int n) {
    srand( time(0) );

    int dartsInCircle = 0;
    for(int i = 0; i < n; ++i) {
        double x = rand() / (double)RAND_MAX, y = rand() / (double)
            RAND_MAX;
        if( sqrt(x*x + y*y) < 1 ) {
            ++dartsInCircle;
        }
    }

    // r^2 is 1, and a/4 = dartsInCircle/n, yielding this for pi:
    return dartsInCircle / static_cast<double>(n) * 4;
}
```

6 Array Operations

6.1 [4 points]

```
void printArray(const int arr[], const int len) {
    for(int i = 0; i < len; ++i) {
        cout << arr[i];
        if(i < len-1) {
            cout << ", ";
        }
    }
}
```

6.2 [4 points]

```
void reverse(int numbers[], const int numbersLen) {
    for(int i = 0; i < numbersLen / 2; ++i) {
        int tmp = numbers[i];
        int indexFromEnd = numbersLen - i - 1;
        numbers[i] = numbers[indexFromEnd];
        numbers[indexFromEnd] = tmp;
    }
}
```

6.3 [6 points]

```
void transpose(const int input[][LENGTH], int output[][WIDTH]) {
    for(int i = 0; i < WIDTH; ++i) {
        for(int j = 0; j < LENGTH; ++j) {
            output[j][i] = input[i][j];
        }
    }
}
```

6.4 [2 points]

A pointer to the first element in the array would be returned, but the array would have gone out of scope, making the pointer invalid.

6.5 [3 points]

```
void reverse(int numbers[], const int numbersLen) {
    for(int i = 0; i < numbersLen / 2; ++i) {
        int tmp = *(numbers + i);
        int indexFromEnd = numbersLen - i - 1;
        *(numbers + i) = *(numbers + indexFromEnd);
        *(numbers + indexFromEnd) = tmp;
    }
}
```

7 Pointers and Strings

7.1 [5 points]

```
int stringLength(const char *str) {
    int length = 0;
    while(*(str + length) != '\0') {
        ++length;
    }
    return length;
}
```

7.2 [3 points]

```
void swap(int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

7.3 [4 points]

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

7.4 [5 points]

```
void swap(int **x, int **y) {
    int *tmp = *x;
    *x = *y;
    *y = tmp;
}
```

7.5 [8 points]

1. `char *nthCharPtr = &oddOrEven[5];`
2. `nthCharPtr -= 2;` or `nthCharPtr = oddOrEven + 3;`
3. `cout << *nthCharPtr;`
4. `char **pointerPtr = &nthCharPtr;`
5. `cout << pointerPtr;`
6. `cout << **pointerPtr;`
7. `nthCharPtr++;` to point to the next character in `oddOrEven` (i.e. one character past the location it currently points to)
8. `cout << nthCharPtr - oddOrEven;`

6.096 Problem Set 3

1 Additional Material

1.1 Arrays of class objects

An array of class objects is similar to an array of some other data type. To create an array of `Point`s, we write

```
Point parray[4];
```

To access the object at position i of the array, we write

```
parray[i]
```

and to call a method on that object method, we write

```
parray[i].methodName(arg1, arg2, ...);
```

To initialize an array of objects whose values are known at compile time, we can write

```
Point parray[4] = {Point(0,1), Point(1,2), Point(3,5), Point(8,13)};
```

We can also allocate an array of objects dynamically using the `new` operator (this implicitly calls the default constructor of each new `Point`):

```
Point* parray = new Point[4];
```

1.2 Static members and variables

Static data members of a class are also known as "class variables," because there is only one unique value for all the objects of that class. Their content is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of the class that are currently allocated, as in the following example:

```
#include <iostream>

using namespace std;

class CDummy
{
 7 public:
 8     static int n;
 9     CDummy() { ++n; }
10     ~CDummy() { --n; }
11 };
12
13 int CDummy::n = 0;
14
15 int main()
16 {
17     CDummy a;
18     CDummy b[5];
19     CDummy* c = new CDummy;
20     cout << a.n << "\n"; // prints out 7
21     delete c;
22     cout << CDummy::n << "\n"; // prints out 6
23     return 0;
24 }
```

In fact, static members have the same properties as global variables, but they can only be referenced via the class: either in class methods, via a class instance (`someObject.staticVariable`, or via the `className::variable` construct.

Because these variables are global, if we were to initialize them in a header file we could end up with that initialization being compiled multiple times (once per time we include the header). To avoid this, we only include a static member's "prototype" (its declaration) in the class declaration, but not its definition (its initialization). This is why line 13 above is necessary, and why if we were to provide a header file for `CDummy`, we would still need to put line 13 in a separate `.cpp` file. If you get linker errors saying a `static int` is undefined, check to see whether you've included a line like line 13 in a `.cpp` file.

Classes can also have `static` member functions – that is, member functions which are associated with the class but do not operate on a particular class instance. Such member functions may not access non-`static` data members. For instance, we might replace `CDummy` above with the following class definition:

```
class CDummy
{
private:
    static int n;
public:
    CDummy() { ++n; }
    ~CDummy() { --n; }
    static int getN() {return n;}
};
```

`getN` could then be called as `c->getN()` or `CDummy::getN()`.

1.3 const member functions

It is clear what const-ness means for a simple value like an int, but it is not clear what functions should be available on a const object, since functions may allow modifications in subtle ways that ought to be forbidden on const objects. To specify to the compiler that a given member function is safe to call on const objects, you can declare the function with the const keyword. This specifies that the function is a "read-only" function that does not modify the object on which it is called.

To declare a const member function, place the const keyword after the closing parenthesis of the argument list. The const keyword is required in both the prototype and the definition. A const member function cannot modify any data members or call any member functions that aren't also declared const. Generally, const member functions should return const values, since they often return references/pointers to internal data, and we wouldn't want to allow someone to get a modifiable reference to the data of a const object.

```
const string &Person::getName() const {
    return name;     // Doesn't modify anything; trying to modify a
                     // data member from here would be a syntax error
}
```

If an object of class Person would be declared as const Person jesse;, no non-const member functions could be called on it. In other words, the set of const member functions of a class defines the set of operations that can be performed on const objects of that class.

1.4 String objects

Manipulating strings as char * or char[] types tends to be unwieldy. In particular, it is difficult to perform modifications on strings that change their length; this requires reallocating the entire array. It is also very difficult to deal with strings whose maximum length is not known ahead of time. Many C++ classes have been created to solve such problems. The C++ standard library includes one such class, appropriately called string, defined in header file string under namespace std.

The string class allows us to do all sorts of nifty operations:

```
1  #include <string>
2  ...
3  string s = "Hello";
4  s += " world!";
5  if(s == "Hello world!") {
6      cout << "Success!" << endl;
7  }
8  cout << s.substr(6, 6) << endl; // Prints "world!"
9  cout << s.find("world"); // (prints "6")
10 cout << s.find('l', 5); // (prints "9")
```

A line-by-line description of the string features this code demonstrates:

3. We can set strings to normal `char *`'s.
4. We can use the `+` operator to append things to a string. Don't worry about how this works for now; we'll see in Lecture 9 how to allow your classes to do things like this.)
5. We can use the `==` operator to test whether two strings are the same. (If we tried to do this with `char *`'s, we'd just be checking whether they point to the same string in memory, not whether the pointed-to strings have the same contents. To check for string equality with `char *`'s, you need to use the function `strcmp`.)
8. We can get a new `string` object that is a substring of the old one.
9. We can find the index a given string within the `string` object.
10. We can find a character as well, and we can specify a starting location for the search.

Take a few minutes to play around with the string class. Look at the documentation at http://www.cplusplus.com/reference/string/string/. In particular, be sure to understand the behavior of the `substr` function.

1.5 Type Conversions and Constructors

Any time you call a function, the compiler will do its best to match the arguments you provide with some function definition. As a last-ditch strategy, it will even try constructing objects for you.

Say you have a function `f` that takes a `Coordinate` object, and that the `Coordinate` constructor is defined to take one `double`. If you call `f(3.4)`, the compiler will notice that there is no `f` that takes a `double`; however, it will also see that it can match the `f` that it found by converting your argument to a `Coordinate` object. Thus, it will automatically turn your statement into `f(Coordinate(3.4))`.

This applies to constructors, as well. Say you have a `Point` class, whose constructor takes two `Coordinates`. If you write `Point p(2.3, 0.5);`, the compiler will automatically turn your statement into `Point p(Coordinate(2.3), Coordinate(2.5);`.

1.6 Sources

- http://www.cplusplus.com/

2 Catch that bug

In this section, the following snippets will have bugs. Identify them and indicate how to correct them. Do these without the use of a computer!

2.1

```
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
    void doubleVal()
    {
        x *= 2;
        y *= 2;
    }
};

int main()
{
    const Point myPoint(5, 3)
    myPoint.doubleVal();
    cout << myPoint.getX() << " " << myPoint.getY() << "\n";
    return 0;
}
```

2.2

```
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
```

```cpp
    void setX(int newX) const { x = newX; }
};

int main()
{
    Point p(5, 3);
    p.setX(9001);
        cout << p.getX() << ' ' << p.getY();
    return 0;
}
```

2.3

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p(5, 3);
    cout << p.x << " " << p.y << "\n";
    return 0;
}
```

2.4

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    void setX(int newX);
};

void setX(int newX){ x = newX; }

int main()
{
    Point p(5, 3);
    p.setX(0);
    cout << p.getX() << " " << "\n";
    return 0;
}
```

2.5

```
...
int size;
cin >> size;
int *nums = new int[size];
for(int i = 0; i < size; ++i)
{
    cin >> nums[i];
}
... // Calculations with nums omitted
delete nums;
...
```

2.6

```
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point *p = new Point(5, 3);
```

```
        cout << p->getX() << ' ' << p->getY();
    return 0;
}
```

(Hint: this bug is a logic error, not a syntax error.)

3 Point

For the next several problems, you should put your class definitions and function prototypes in a header file called `geometry.h`, and your function definitions in a file called `geometry.cpp`. If your functions are one-liners, you may choose to include them in the header file.

In this section you will implement a class representing a point, appropriately named `Point`.

3.1 Foundation

Create the class with two private `int`s. Name them `x` and `y`.

3.2 Constructors

Implement a single constructor that, if called with 0 arguments, initializes a point to the origin – $(0, 0)$ – but if called with two arguments x and y, creates a point located at (x, y). (*Hint:* You will need to use default arguments.

3.3 Member Functions

Support the following operations using the given function signatures:

- Get the x coordinate

  `int Point::getX() const`

- Get the y coordinate

  `int Point::getY() const`

- Set the x coordinate

  `void Point::setX(const int new_x)`

- Set the y coordinate

  `void Point::setY(const int new_y)`

4 `PointArray`

In this section you will implement a class representing an array of `Points`. It will allow dynamically resizing the array, and it will track its own length so that if you were to pass it to a function, you would not need to pass its length separately.

4.1 Foundation

Create the class with two private members, a pointer to the start of an array of `Points` and an `int` that stores the size (length) of the array.

4.2 Constructors

Implement the default constructor (a constructor with no arguments) with the following signature. It should create an array with size 0.

Implement a constructor that takes a `Point` array called `points` and an `int` called `size` as its arguments. It should initialize a `PointArray` with the specified size, copying the values from `points`. You will need to dynamically allocate the `PointArray`'s internal array to the specified size.

```
PointArray::PointArray(const Point points[], const int size)
```

Finally, implement a constructor that creates a copy of a given `PointArray` (a *copy constructor*).

```
PointArray::PointArray(const PointArray& pv)
```

(*Hint*: Make sure that the two `PointArrays` do not end up using the same memory for their internal arrays. Also make sure that the contents of the original array are copied, as well.)

4.3 Destructors

Define a destructor that deletes the internal array of the `PointArray`.

```
PointArray::~PointArray()
```

4.4 Dealing with an ever-changing array

Since we will allow modifications to our array, you'll find that the internal array grows and shrinks quite often. A simple (though very inefficient) way to deal with this without repetitively writing similar code is to write a member function `PointArray::resize(int n)` that allocates a new array of size n, copies the first $\min(\text{previous array size}, n)$ existing elements into it, and deallocates the old array. If doing so has increased the size, it's fine for `resize` to leave the new spaces uninitialized; whatever member function calls it will be responsible for filling those spaces in. Then every time the array size changes at all (including `clear`), you can call this function.

In some cases, after you call this function, you will have to subsequently shift some of the contents of the array right or left in order to make room for a new value or get rid of an old one. This is of course inefficient; for the purposes of this exercise, however, we won't be worrying about efficiency. If you wanted to do this the "right" way, you'd remember both how long your array is and how much of it is filled, and only reallocate when you reach your current limit or when how much is filled dips below some threshhold.

Add the `PointArray::resize(int n)` function as specified above to your `PointArray` class. Give it an appropriate access modifier, keeping in mind that this is meant for use only by internal functions; the public interface is specified below.

4.5 Member Functions

Implement public functions to perform the following operations:

- Add a `Point` to the end of the array

  ```
  void PointArray::push_back(const Point &p)
  ```

- Insert a `Point` at some arbitrary position (subscript) of the array, shifting the elements past `position` to the right

  ```
  void PointArray::insert(const int position, const Point &p)
  ```

- Remove the `Point` at some arbitrary position (subscript) of the array, shifting the remaining elements to the left

  ```
  void PointArray::remove(const int pos)
  ```

- Get the size of the array

  ```
  const int PointArray::getSize() const
  ```

- Remove everything from the array and sets its size to 0

  ```
  void PointArray::clear()
  ```

- Get a pointer to the element at some arbitrary position in the array, where positions start at 0 as with arrays

  ```
  Point *PointArray::get(const int position)
  ```

  ```
  const Point *PointArray::get(const int position) const
  ```

If `get` is called with an index larger than the array size, there is no `Point` you can return a pointer to, so your function should return a null pointer. Be sure your member functions all behave correctly in the case where you have a 0-length array (i.e., when your `PointArray` contains no points, such as after the default constructor is called).

4.5.1

Why do we need `const` and non-`const` versions of `get`? (Think about what would happen if we only had one or the other, in particular what would happen if we had a `const PointArray` object.)

5 Polygon

In this section you will implement a class for a convex polygon called `Polygon`. A *convex polygon* is a simple polygon whose interior is a convex set; that is, if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.

`Polygon` will be an *abstract class* – that is, it will be a placeholder in the class hierarchy, but only its subclasses may be instantiated. `Polygon` will be an immutable type – that is, once you create the `Polygon`, you will not be able to change it.

Throughout this problem, remember to use the `const` modifier where appropriate.

5.1 Foundation

Create the class with two protected members: a `PointArray` and a `static int` to keep track of the number of `Polygon` instances currently in existence.

5.2 Constructors/Destructors

Implement a constructor that creates a `Polygon` from two arguments: an array of `Point`s and the length of that array. Use member initializer syntax to initialize the internal `PointArray` object of the `Polygon`, passing the `Polygon` constructor arguments to the `PointArray` constructor. You should need just one line of code in the actual constructor body.

Implement a constructor that creates a polygon using the points in an existing `PointArray` that is passed as an argument. (For the purposes of this problem, you may assume that the order of the points in the `PointArray` traces out a convex polygon.) You should make sure your constructor avoids the unnecessary work of copying the entire existing `PointArray` each time it is called.

Will the default "memberwise" copy constructor work here? Explain what happens to the `PointArray` field if we try to copy a `Polygon` and don't define our own copy constructor.

Make sure that your constructors and destructors are set up so that they correctly update the `static int` that tracks the number of `Polygon` instances.

5.3 Member Functions

Implement the following public functions according to the descriptions:

- `area`: Calculates the area of the `Polygon` as a `double`. Make this function pure virtual, so that the subclasses must define it in order to be instantiated. (This makes the class abstract.)
- `getNumPolygons`: Returns the number of `Polygons` currently in existence, and can be called even without referencing a `Polygon` instance. (*Hint:* Use the `static int`.)
- `getNumSides`: Returns the number of sides of the `Polygon`.
- `getPoints`: Returns an unmodifiable pointer to the `PointArray` of the `Polygon`.

5.4 Rectangle

Write a subclass of `Polygon` called `Rectangle` that models a rectangle. Your code should

- Allow constructing a `Rectangle` from two `Points` (the lower left coordinate and the upper right coordinate)
- Allow construct a `Rectangle` from four `ints`
- Override the `Polygon::area`'s behavior such that the rectangle's area is calculated by multiplying its length by its width, but still return the area as a double.

Both of your constructors should use member initializer syntax to call the base-class constructor, and should have nothing else in their bodies. C++ unfortunately does not allow us to define arrays on the fly to pass to base-class constructors. To allow using member initializer syntax, we can implement a little trick where we have a global array that we update each time we want to make a new array of `Points` for constructing a `Polygon`. You may include the following code snippet in your `geometry.cpp` file:

```
Point constructorPoints[4];

Point *updateConstructorPoints(const Point &p1, const Point &p2,
    const Point &p3, const Point &p4 = Point(0,0)) {
    constructorPoints[0] = p1;
    constructorPoints[1] = p2;
    constructorPoints[2] = p3;
    constructorPoints[3] = p4;
    return constructorPoints;
}
```

You can then pass the return value of `updateConstructorPoints(...)` (you'll need to fill in the arguments) as the `Point` array argument of the `Polygon` constructor. (Remember, the name of an array of `T`s is just a `T` pointer.)

5.5 Triangle

Write a subclass of `Polygon` called `Triangle` that models a triangle. Your code should

- Construct a `Triangle` from three `Points`
- Override the area function such that it calculates the area using Heron's formula:

  $$K = \sqrt{s(s-a)(s-b)(s-c)}$$

  where a, b, and c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$.

Use the same trick as above for calling the appropriate base-class constructor. You should not need to include any code in the actual function body.

5.6 Questions

1. In the `Point` class, what would happen if the constructors were private?
2. Describe what happens to the fields of a `Polygon` object when the object is destroyed.
3. Why did we need to make the fields of `Polygon` protected?

For the next question, assume you are writing a function that takes as an argument a `Polygon *` called `polyPtr`.

4. Imagine that we had overridden `getNumSides` in each of `Rectangle` and `Triangle`. Which version of the function would be called if we wrote `polyPtr->getNumSides()`? Why?

5.7 Putting it All Together

Write a small function with signature `void printAttributes(Polygon *)` that prints the area of the polygon and prints the (x, y) coordinates of all of its points.

Finally, write a small program (a `main` function) that does the following:

- Prompts the user for the lower-left and upper-right positions of a `Rectangle` and creates a `Rectangle` object accordingly
- Prompts the user for the point positions of a `Triangle` and creates a `Triangle` object accordingly
- Calls `printAttributes` on the `Rectangle` and `Triangle`, printing an appropriate message first.

6 Strings

In this section you will write a program that turns a given English word into Pig Latin. Pig Latin is a language game of alterations played in English. To form the Pig Latin version of an English word, the onset of the first consonant is transposed to the end of the word an an *ay* is affixed. Here are the rules:

1. In words that begin with consonant sounds, the initial consonant (if the word starts with 'q', then treat 'qu' as the initial consonant) is moved to the end of the word, and an "ay" is added. For example:
   - beast : east-bay
   - dough : ough-day
   - happy : appy-hay
   - question : estion-quay
2. In words that begin with vowel sounds, the syllable "way" is simply added to the end of the word.

Write a function `pigLatinify` that takes a `string` object as an argument. (You may assume that this string contains a single lowercase word.) It should return a new `string` containing the Pig Latin version of the original. (Yes, it is inefficient to copy a whole string in the return statement, but we won't worry about that. Also, your compiler is probably clever enough to do some optimizations.) You may find it useful to define a constant of type `string` or `char*` called `VOWELS`.

Remember that `string` objects allow the use of operators such as `+=` and `+`.

(Your answers for this problem should go in a separate file from `geometry.h` and `geometry.cpp`.)

Lab 3 Solutions

2 Catch that bug

2.1

Line 20 is missing a semicolon. `doubleVal()` cannot be applied to `myPoint` because `myPoint` is `const`.

2.2

Line 11 contains an error because the function is declared `const`, i.e. as not modifying any instance variables, but it assigns `x` to another value.

2.3

`x` and `y` are private members and cannot be accessed outside of the class.

2.4

`setX` is missing the scope; the function should be declared as `void Point::setX(int newX) { x = newX; }`

2.5

Deleting a dynamically allocated array requires `delete[]`, not `delete`.

2.6

`p` is allocated using `new`, but is never deallocated with `delete`. Every piece of memory allocated with `new` must be deallocated somewhere with a corresponding `delete`.

3 Point

3.1 geometry.h

```
class Point {
    int x, y;

public:
    Point(int xx=0, int yy=0) {x = xx; y = yy;}
    int getX() const {return x;}
    int getY() const {return y;}
    void setX(const int xx) {x = xx;}
    void setY(const int yy) {y = yy;}
};
```

Note: the `getX` and `getY` functions should really have been declared as `const`, but we neglected to ask you to do this, so it's fine if you did not.

4 PointArray

4.1 geometry.h

```
class PointArray {
    int size;
    Point *points;

    void resize(int size);

public:
    PointArray();
    PointArray(const Point pts[], const int size);
    PointArray(const PointArray &pv);
    ~PointArray();

    void clear();
    int getSize() const { return size;}
    void push_back(const Point &p);
    void insert(const int pos, const Point &p);
    void remove(const int pos);
    Point *get(const int pos);
    const Point *get(const int pos) const;
};
```

4.2 geometry.cpp

```cpp
#include "geometry.h"

PointArray::PointArray() {
    size = 0;
    points = new Point[0]; // Allows deleting later
}

PointArray::PointArray(const Point ptsToCopy[], const int toCopySize
   ) {
    size = toCopySize;
    points = new Point[toCopySize];
    for(int i = 0; i < toCopySize; ++i) {
        points[i] = ptsToCopy[i];
    }
}

PointArray::PointArray(const PointArray &other) {
    // Any code in the PointArray class has access to
    // private variables like size and points
    size = other.size;
    points = new Point[size];
    for (int i = 0; i < size; i++) {
        points[i] = other.points[i];
    }
}

PointArray::~PointArray() {
    delete[] points;
}

void PointArray::resize(int newSize) {
    Point *pts = new Point[newSize];
    int minSize = (newSize > size ? size : newSize);
    for (int i = 0; i < minSize; i++)
        pts[i] = points[i];
    delete[] points;
    size = newSize;
    points = pts;
}

void PointArray::clear() {
    resize(0);
}
```

```cpp

void PointArray::push_back(const Point &p) {
    resize(size + 1);
    points[size - 1] = p;
    // Could also just use insert(size, p);
}

void PointArray::insert(const int pos, const Point &p) {
    resize(size + 1);

    for (int i = size - 1; i > pos; i--) {
        points[i] = points[i-1];
    }

    points[pos] = p;
}

void PointArray::remove(const int pos) {
    if(pos >= 0 && pos < size) { // pos < size implies size > 0
        // Shift everything over to the left
        for(int i = pos; i < size - 2; i++) {
            points[i] = points[i + 1];
        }
        resize(size - 1);
    }
}

Point *PointArray::get(const int pos) {
    return pos >= 0 && pos < size ? points + pos : NULL;
}

const Point *PointArray::get(const int pos) const {
    return pos >= 0 && pos < size ? points + pos : NULL;
}
```

4.2.1

1. We need the `const` versions so that we can return read-only pointers for `const PointArray` objects. (If the `PointArray` object is read-only, we don't want to allow someone to modify a `Point` it contains just by using these functions.) However, many times we will have a non-`const` `PointArray` object, for which we may want to allow modifying the contained `Point` objects. If we had only `const` accessor functions, then even in such a case we would be returning a `const` pointer. To allow returning a non-`const` pointer in situations where we might want one, we need non-`const` versions of these

functions, as well.

5 Polygon and friends

5.1 Polygon

5.1.1 geometry.h

```cpp
class Polygon {
protected:
    static int numPolygons;
    PointArray points;

public:
    Polygon(const PointArray &pa);
    Polygon(const Point points[], const int numPoints);
    virtual double area() const = 0;
    static int getNumPolygons() {return numPolygons;}
    int getNumSides() const {return points.getSize();}
    const PointArray *getPoints() const {return &points;}
    ~Polygon() {--numPolygons;}
};
```

5.1.2 geometry.cpp

```cpp
int Polygon::n = 0;

Polygon::Polygon(const PointArray &pa) : points(pa) {
    ++numPolygons;
}

Polygon::Polygon(const Point pointArr[], const int numPoints) :
   points(pointArr, numPoints) {
    ++numPolygons;
}
```

5.2 Rectangle

5.2.1 geometry.h

```cpp
class Rectangle : public Polygon {
public:
    Rectangle(const Point &a, const Point &b);
    Rectangle(const int a, const int b, const int c, const int d);
    virtual double area() const;
};
```

5.2.2 geometry.cpp

```

Point constructorPoints[4];

Point *updateConstructorPoints(const Point &p1, const Point &p2,
   const Point &p3, const Point &p4 = Point(0,0)) {
    constructorPoints[0] = p1;
    constructorPoints[1] = p2;
    constructorPoints[2] = p3;
    constructorPoints[3] = p4;
    return constructorPoints;
}

Rectangle::Rectangle(const Point &ll, const Point &ur)
    : Polygon(updateConstructorPoints(ll, Point(ll.getX(), ur.getY()
       ),
                                      ur, Point(ur.getX(), ll.getY()
                                         )), 4) {}

Rectangle::Rectangle(const int llx, const int lly, const int urx,
   const int ury)
    : Polygon(updateConstructorPoints(Point(llx, lly), Point(llx,
       ury),
                                      Point(urx, ury), Point(urx,
                                         lly)), 4) {}

double Rectangle::area() const {
    int length = points.get(1)->getY() - points.get(0)->getY();
    int width  = points.get(2)->getX() - points.get(1)->getX();
    return std::abs((double)length * width);
}
```

(You'll need to add `#include <cmath>` at the top of your file to use the `abs` function.)

5.3 Triangle

5.3.1 geometry.h

```
class Triangle : public Polygon {
```

```cpp
public:
    Triangle(const Point &a, const Point &b, const Point &c);
    virtual double area() const;
};
```

5.4 geometry.cpp

```cpp
Triangle::Triangle(const Point &a, const Point &b, const Point &c)
    : Polygon(updateConstructorPoints(a, b, c), 3) {}

double Triangle::area() const {
    int dx01 = points.get(0)->getX() - points.get(1)->getX(),
        dx12 = points.get(1)->getX() - points.get(2)->getX(),
        dx20 = points.get(2)->getX() - points.get(0)->getX();
    int dy01 = points.get(0)->getY() - points.get(1)->getY(),
        dy12 = points.get(1)->getY() - points.get(2)->getY(),
        dy20 = points.get(2)->getY() - points.get(0)->getY();

    double a = std::sqrt(dx01*dx01 + dy01*dy01),
           b = std::sqrt(dx12*dx12 + dy12*dy12),
           c = std::sqrt(dx20*dx20 + dy20*dy20);

    double s = (a + b + c) / 2;

    return std::sqrt( s * (s-a) * (s-b) * (s-c) );
}
```

5.5 main.cpp

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

#include "geometry.h"

void printAttributes(Polygon *p) {
    cout << "p's area is " << p->area() << ".\n";

    cout << "p's points are:\n";
    const PointArray *pa = p->getPoints();
    for(int i = 0; i < pa->getSize(); ++i) {
        cout << "(" << pa->get(i)->getX() << ", " << pa->get(i)->
            getY() << ")\n";
    }
}

int main(int argc, char *argv[]) {
    cout << "Enter lower left and upper right coords of rectangle as
        four space separated integers: ";
    int llx, lly, urx, ury;
    cin >> llx >> lly >> urx >> ury;
    Point ll(llx, lly), ur(urx, ury);
    Rectangle r(ll, ur);
    printAttributes(&r);

    cout << "Enter three coords of triangle as six space separated
       integers: ";
    int x1, y1, x2, y2, x3, y3;
    cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
    Point a(x1, y1), b(x2, y2), c(x3, y3);
    Triangle t(a, b, c);
    printAttributes(&t);

    return 0;
}
```

5.6 Questions

1. If the constructors were private, then we would not be able to create any `Point` objects.

2. When a `Polygon` is destroyed, the counter for number of `Polygons` created is decremented, and the `PointArray`'s destructor is implicitly called.

3. We had to make the fields of `Polygon` protected so that they could be accessed from `Rectangle` and `Triangle`, but not by arbitrary outside code.

4. The `getNumSides` from `Polygon` would be called, because the function is not `virtual`.

6 Strings

```
const string vowels = "aeiou";

string pigLatinify(const string s) {
    if(s.size() == 0) {
        // oops, empty string
        return s;
    }

    if(s.find("qu") == 0) { // Starts with "qu"
        return s.substr(2, s.size()-2) + "-" + s.substr(0, 2) + "ay"
            ;
    } else if(vowels.find(s[0]) != string::npos) {   // Starts with
       a vowel
        return s + "way";
    } else {
        return s.substr(1, s.size()-1) + "-" + s[0] + "ay";
    }
}
```

6.096 Problem Set 4

1 Additional Material

1.1 Templates and Header Files

The compiler does not compile a templated function until it encounters a use of it – until that function is used with a particular type parameter, at which point it is compiled for that type parameter. Because of this, if you define a templated class in a header file and implement its (templated) functions in a .cpp file, the code in the .cpp file will never get compiled unless you use the templated functions within that .cpp file. To solve this problem, templated classes and functions are generally just implemented in the header file, with no .cpp file.

1.2 Templates and `friend` Functions

There are some syntax oddities when using friend functions with templated classes. In order to declare the function as a friend of the class, you need a function prototype (or the full function definition) to appear before the class definition. This is a problem, because you'll often need to have the class already defined in order for the function prototype to make sense. To get around this issue, when writing friend functions with templated classes, you should include declarations in the following order:

1. A templated *forward declaration* of the class. (A forward declaration is a statement like `class SomeClass;`, with no class body, that just alerts the compiler that the class definition is coming.)

2. A templated function prototype for the friend function (or the entire function definition).

3. The full templated class definition, including the `friend` statement. When you write the name of the function in the `friend` statement, you need to include an extra `<>` after it to indicate that it is a templated function (e.g., `friend MyClass operator+<>(const MyClass &c1, const MyClass &c2;`).

4. The operator function definition, if you didn't include it above.

2 Multi-Type `min`

2.1

Using templates, implement a `min` function which returns the minimum of two elements of any comparable type (i.e., it takes two arguments of some type `T`, and works as long as values of type `T` can be compared with the `<` operator).

(To test this function, you may need to omit your usual `using namespace std;` line, since there is already an `std::min` function.)

2.2

Implement the `min` functionality from part 1 using only preprocessor macros. (*Hint:* You will probably need the ternary operator – the `?:` syntax.)

3 Casting

Assume you implemented Problem 5 from Problem Set 3 correctly. This would mean you would have a working `Polygon` class, and inheriting from that a `Triangle` class and a `Rectangle` class. Now imagine you have a pointer declared as `Rectangle *rect;` that has been properly initialized.

3.1

Write a line of code showing how you would cast `rect` to a `Triangle *` without checking for type correctness (i.e., without checking whether it actually points to a `Triangle`). Do not use C-style casts.

3.2

Now write a line of code that does the same thing, but checks for type correctness and throws an exception or returns a null pointer if `rect` does not actually point to a `Triangle`.

4 Templated Stack

A *stack* data structure stores a set of items, and allows accessing them via the following operations:

- *Push* – add a new item to the stack
- *Pop* – remove the most recently added item that is still in the stack (i.e. that has not yet been popped)

- *Top* – Retrieve

For more explanation, see http://en.wikipedia.org/wiki/Stack_(data_structure).

4.1

Using templates, implement a `Stack` class that can be used to store items of any type. You do not need to implement any constructors or destructors; the default constructor should be sufficient, and you will not need to use `new`. Your class should support the following 3 public functions (assuming `T` is the parameterized type which the `Stack` is storing):

- `bool Stack::empty()` – returns whether the stack is empty
- `void Stack::push(const T &item)` – adds `item` to the stack
- `T &Stack::top()` – returns a reference to the most-recently-added item
- `void Stack::pop()` – removes the most-recently-added item from the stack

Use an STL `vector`, `deque`, or `list` to implement your `Stack`. You may not use the STL `stack` class. Make the member functions `const` where appropriate, and add `const`/non-`const` versions of each function for which it is appropriate. In the case where the `Stack` is empty, `pop` should do nothing, and `top` should behave exactly as normal (this will of course cause an error).

When working with templated classes, you cannot separate the function implementations into a separate .cpp file; put all your code in the class definition.

(*Hint:* You can represent pushing by adding to the end of your `vector`, and popping by removing the last element of the `vector`. This ensures that the popped item is always the most recently inserted one that has not yet been popped.)

4.2 `friend` Functions and Operator Overloading (Optional)

Make a `friend` function that implements a `+` operator for `Stack`s. The behavior of the `+` operator should be such that when you write `a + b`, you get a new stack containing `a`'s items followed by `b`'s items (assuming `a` and `b` are both `Stack`s), in their original order. Thus, in the following example, the contents of `c` would be the same as if 1, 2, 3, and 4 had been pushed onto it in that order. (`c.top()` would therefore return the value 4.)

```
Stack<int> a, b;
a.push(1);
a.push(2);
b.push(3);
b.push(4);
Stack<int> c = a + b;
```

Put your code for this section in the same file as for the previous one.

5 Graph Representation (Optional)

A *graph* is a mathematical data structure consisting of *nodes* and *edges* connecting them. To help you visualize it, you can think of a graph as a map of "cities" (nodes) and "roads" (edges) connecting them. In an *directed graph*, the direction of the edge matters – that is, an edge from A to B is not also an edge from B to A. You can read more at Wikipedia: http://en.wikipedia.org/wiki/Graph_(mathematics).

One way to represent a graph is by assigning each node a unique ID number. Then, for each node ID n, you can store a list of node ID's to which n has an outgoing edge. This list is called an *adjacency list*.

Write a `Graph` class that uses STL containers (`vector`s, `map`s, etc.) to represent a directed graph. Each node should be represented by a unique integer (an `int`). Provide the following member functions:

- `Graph::Graph(const vector &starts, const vector &ends)`
  Constructs a `Graph` with the given set of edges, where `starts` and `ends` represent the ordered list of edges' start and endpoints. For instance, consider the following graph:

  The `vector`s used to initialize a `Graph` object representing this graph would be:

`start:`	1	1	1	5	5	4
`end:`	2	3	4	4	2	2

- `int Graph::numOutgoing(const int nodeID) const`
  Returns the number of outgoing edges from `nodeID` – that is, edges with `nodeID` as the start point

- `const vector<int> &Graph::adjacent(const int nodeID) const`
  Returns a reference to the list of nodes to which `nodeID` has outgoing edges

(*Hint:* Use the following data type to associate adjacency lists with node ID's: `map<int, vector<int> >`. This will also allow for easy lookup of the adjacency list for a given node ID. Note that the `[]` operator for `map`s inserts the item if it isn't already there.)

The constructor should throw an `invalid_argument` exception of the two `vector`s are not the same length. (This standard exception class is defined in header file `stdexcept`.) The other member functions should do likewise if the nodeID provided does not actually exist in the graph. (*Hint:* Use the `map::find`, documented at http://www.cplusplus.com/reference/stl/map/find/.

Lab 4 Solutions

2 Multi-Type `min`

2.1

```
template<typename T>
T min(const T t1, const T t2) {
    return t1 < t2 ? t1 : t2;
}
```

2.2

```
#define min(x, y) (x < y ? x : y)
```

3 Casting

3.1

`static_cast<Triangle *>(p)`

or

`reinterpret_cast<Triangle *>(p)`

3.2

`dynamic_cast<Triangle *>(p)`

4 Templated Stack

4.1

```cpp
template<class T> class Stack;

template<class T>
Stack<T> operator+(const Stack<T> &s1, const Stack<T> &s2);

 {
    Stack<T> result = s1;

    for(unsigned i = 0; i < s1.items.size(); ++i) {
        result.items.push_back(s2.items[i]);
    }

    return result;
}

template<class T>
class Stack {
    friend Stack<T> operator+<>(const Stack<T> &s1, const Stack<T> &
       s2);
    vector<T> items;

public:
    bool empty() const {return items.empty();}
    void push(const T &item) {items.push_back(item);}
    T pop() {
        T last = items.back();
        items.pop_back();
        return last;
    }
};

template<class T>
Stack<T> operator+(const Stack<T> &s1, const Stack<T> &s2)
{
    Stack<T> result = s1;

    for(unsigned i = 0; i < s1.items.size(); ++i) {
        result.items.push_back(s2.items[i]);
    }

    return result;
}
```

5 Graph Representation

```cpp
class Graph {
protected:
    map<int, vector<int> > outgoing;

public:
    Graph(const vector<int> &startPoints, const vector<int> &
       endPoints);
    int numOutgoing(const int nodeID) const;
    const vector<int> &adjacent(const int nodeID) const;
};

// In a .cpp file...

#include <stdexcept>

Graph::Graph(const vector<int> &startPoints, const vector<int> &
    endPoints) {
    if(startPoints.size() != endPoints.size()) {
        throw invalid_argument("Start/end point lists differ in
           length");
    }

    for(unsigned i = 0; i < startPoints.size(); i++ ) {
        int start = startPoints[i], end = endPoints[i];
        outgoing[start].push_back(end);
        outgoing[end]; // Just to indicate this node exists
    }
}

int Graph::numOutgoing(const int nodeID) const {
    return adjacent(nodeID).size();
}

const vector<int> &Graph::adjacent(const int nodeID) const {
    map<int, vector<int> >::const_iterator i = outgoing.find(nodeID)
       ;
    if(i == outgoing.end()) {
        throw invalid_argument("Invalid node ID");
    }
    return i->second;
}
```

6.096 Final Project

1 Project Description

We would like you to demonstrate everything you have learned during this course in your final project. It should be a fully-fledged C++ program, written entirely from the scratch by you, satisfying the requirements specified below. We are not forcing you to work on anything specific – you can come up with any idea that you want for the final project. Possible ideas include:

1. Games (can be multi- or single-player; no AI needed):
   - Card games (e.g. Blackjack, Poker)
   - Reversi
   - Gomoku
   - Connect Four
   - Maxit (very simple – two players just move around a grid from space to space, alternating turns, and each space has a number which gets added to your score; Google for more)
2. An in-memory database (no need to save data to files) in which the user can enter/-modify/view records. For instance, you might make:
   - A database of MIT students, classes they take, etc., allowing adding, removing students, etc.
   - A database of sport results, fixtures, statistics, players, etc.

2 Requirements

1. Your project should be large enough to take about 10-15 hours of coding.
2. Your project proposal needs to be approved by the staff first.
3. Your project must make use of all of the following:
   - Classes (preferably using advanced class features such as inheritance wisely)

- Functions
- STL (Standard Template Library)

4. You may not use any other external libraries (e.g. for graphical interfaces). Generally a good rule of thumb is that if something was not covered in this course, you should probably not use it (if in doubt, email us!). Stick to a text interface.

5. It is very important that you write easily readable, well-designed code.

6. Include a README file, with some basic documentation and instructions on how to use your program. Also include in this README what problems you had with your project, what the challenges were, and what would you have done differently if you could do it again.

3 Deadlines

- 23 January (Sunday) – The project proposal:

  Send us a short project proposal, sketching briefly what your project will do and how it is going to satisfy the requirements. Please limit yourself to at most half a page (the proposal won't count toward your final grade)
- 30 January (Sunday) – The final project: Send us your project, satisfying the requirements. Pack everything into a single .zip file (i.e. your code and the README file). You will get bonus points for the use of more advanced C++ feaures.