LECTURE 12
Some Tips on How to Program in C++

When it comes to learning how to program in C++, there's no substitute for experience. But there are some things that are good to keep in mind. Advice on how to construct programs can be found in the last 6 chapters of *More C++ for Dummies* and in chapters 23 and 24 of the 3rd edition of Stroustrup. Here is a brief summary of that advice:

Understand and State the Problem

It's important to have a clear understanding of the problem and how to solve it. Although this may seem obvious, you would be surprised at how many programmers start writing a program without writing down or even clearly stating the problem that their program is supposed to solve.

Finding the Classes

If you're trying to figure out what classes you should have, state the problem and solution. Note the nouns and verbs you use in stating the problem, and use these as candidates for your classes. (Convert the verbs into nouns.) Represent concepts as classes. Each class should stand for something and it should represent just *one* thing.

Understand the Relationships Between Classes

Group the classes according to their relationships. To clarify your thinking, try drawing a diagrams to illustrate how the classes are related. This is helpful for hierarchical relationships. Hierarchies often translate into inherited classes. Use public inheritance to represent `IS_A` relationships. For example, a Manager `IS_A` Employee. In a hierarchy of inherited classes, figure out what properties the classes have in common, and make these properties of the base class. This is called *factoring*. Use membership in classes to represent `HAS_A` relationships, e.g., a Car `HAS_A` Motor. So make motor a member of the class Car. Diagrams can also help illustrate the relationships between different groups of classes.

Class Structure

To be an effective abstraction, a class should demonstrate the following properties:

1. Maximize "cohesion" within a class. Cohesion refers to how well the class describes a single abstraction in a clear, concise manner. A class has good cohesion if it is sufficient, primitive, and complete.

   A class is *sufficient* if it captures enough characteristics of the abstraction to permit efficient interaction between classes. For example, to be an effective representation of an employee, the class `Employee` must represent all the relevant properties of an employee in a way that's useful to the rest of the program. The program shouldn't rely on external functions or on outside data areas to be able to make use of the `Employee` class.

   A class is `primitive` if you can't easily divide it into separate classes. A primitive class represents a single concept. Although dividing a nonprimitive class may increase the number of classes, it generates smaller, simpler, and more useful classes.

Dividing up a primitive class results in subclasses that do not completely define a concept. As a result, the two classes must make frequent reference to each other. The interclass coupling rises dramatically.

Finally a class is *complete* if its public interface provides a full set of operations. Other classes can easily use a complete class because all the operations other classes will need are available.

2. Strive for encapsulation and prefer loosely coupled classes. *Coupling* refers to the level of interaction between different classes. Loosely coupled classes are preferable to tightly coupled classes for the following reasons:

   (a) Loosely coupled classes are more likely to be reusable with other classes because they don't rely heavily on outside classes.

   (b) A loosely coupled class is easier to understand because it can be understood by itself.

   (c) A loosely coupled class tends to be insulated from changes in other classes.

   Public Interfaces

   Some things to keep in mind about the interface between a class and the rest of the world:

   (a) A class with a simple, easy–to–understand public interface tends to be more loosely coupled with its environment.

   (b) Limit the number of friends.

   (c) Keep the interface simple.

3. Keep classes simple; minimize their complexity.

It's generally a good idea to make data members private or protected, not public. Write public access functions for these data members. This gives the class some control over how applications outside the class interact with the class. Member functions tend to be public since they often do tasks that the outside world needs. Try to keep member functions simple.

Make Your Program Easy To Read

You should make your program easy to read. This helps others who are trying to understand your program and also helps you find bugs. You should adopt a style with the following features:

1. Liberal but consistent use of white space, including blank lines, to accentuate the structure of the program. Indent to show where loops and `if` blocks begin and end.

2. Effective use of block comments to explain what's going on. In my opinion, you can never have too many comment lines. The beginning of your program should

state what the program is meant to do. Each class should begin with a description of what it contains and what it does. In the class definition in the `.h` file, each member function and data member should have a brief description describing what it is for and, for a member function, what it does. Description of member functions should explain what the input arguments are, what the function does, and what it returns.

3. A uniform naming convention for classes and objects. Use names that are descriptive of the object. One notation used for naming classes and objects is called Hungarian notation. It goes like this:

   (a) Use all uppercase letters for variables in `#define` statements. Use underscores to delimit words in a multiword name.

   (b) Begin each word in a multiword name with an uppercase letter, e.g., `RealFunction`. Avoid the use of underscores except as noted in the preceding rule.

   (c) Begin class, typedef, and enumerated type names with an uppercase letter.

   (d) Begin function names with a lowercase letter.

   (e) Begin object names with one or more lowercase letters indicating the object type, using the prefixes listed below:

```
      Prefix                      Type
---------------------------------------------------------------------
       c                          char
       n                          integer
       b                          bool
       f                          float
       d                          double
       l                          long
       u                          unsigned
       p                          pointer
       s                          string
       sz                         ascii string
```

   For example, `int *pnMode` is a pointer to an integer.

Debugging

You shouldn't write your whole program and then try to debug it all at once. You should debug it as you go along. Sometimes I test in `main()` some of the functions that I will later make member functions. I usually insert write or `cout` statements to zero in on a bug. You can use the debugger package on your computer. For unix systems there is `ddd` which is a front end for `gdb`. To use `ddd` or `gdb`, compile the program with a `-g` flag. Then run it with `ddd` or `gdb`. You would type something like this:

```
sun1% g++ -g yourprogram.C
sun1% gdb a.out
```

(`sun1%` is the prompt on sun1.ps.uci.edu in the computer lab.) Once you have debugged the program, compile it without the `-g` flag; it will run faster that way. There is also `ups` which is another debugging package you can use. Look up the man pages for documentation. You can also insert error checking code into your program while you are developing it. Later, when things are running smoothly, you can take the error checking lines out. For example, you can make an `isLegal()` member function that checks the range and type of each data member or the input parameters passed to a member function. One can also "throw and catch exceptions".

Program Evolution

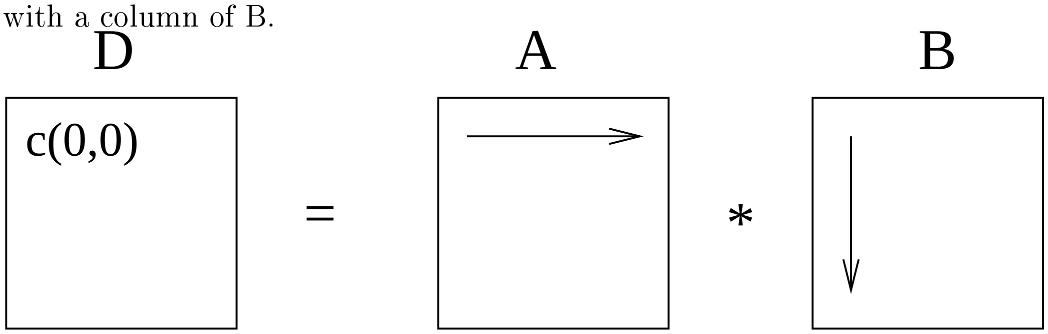
As you write and work with your program, you will probably realize that there are better ways to do things and better ways to organize it. And perhaps, better choices for classes than what you chose initially. This is normal, and you should not be afraid to make changes. If your program is well encapsulated, then you can make changes in one part of your program without seriously affecting other parts of your program.

Efficiency and Optimization

Optimization and efficiency are important goals, especially for scientific programming. If you are trying to optimize your code, i.e., if you are trying to get it to run faster, you shouldn't try to optimize every last line of your program. Rather you should try to improve the code where the computer is spending the most time. Quite often this is in the innermost loop of a nested series of loops. To find out where the most time is spent in running your program, you can use a profiling software. In unix, you compile your program with a `-pg` flag. Then you run it as usual. This produces a file `gmon.out`. Then type `gprof`, and an evaluation of your program will be displayed. So the commands you type look like

```
sun1% g++ -pg myprogram.cc
sun1% a.out
sun1% gprof
```

You should estimate how many times the innermost loop is evaluated when your program is run. You may find that it's millions of times. So in the innermost loop you should have very low level code that's easy and quick to execute; not high level code that involves a lot of manipulations. For example, you don't want `if` statements in the innermost loop if you can help it. You also don't want code in the innermost loop that costs a lot of overhead, e.g., going through a lot of pointers to get to the right place in memory. Efficiently accessing (RAM) memory is important. Sometimes you just have to experiment to see what runs faster on a particular machine with a particular compiler. Different compilers optimize code in different ways for different processors. The order in which you do operations can make all the difference. For example, if you go through the elements of a matrix as in matrix multiplication, you want the fastest moving index in the innermost loop. C and C++ store elements by row, so when going through matrix elements, the column index moves the fastest. But Fortran stores matrix elements by column, so the row index moves the fastest. Accessing adjacent elements in memory is better. The less jumping around in memory, the faster the program. As you go through a row in C or C++, you are going through elements that are stored next to each other. Suppose we multiply matrix A by matrix B to get matrix D. We take a dot product of a row in A with a column of B.



So we can efficiently access the elements of A but not of B. This slows down our program. To make things more efficient, we can transpose matrix B. Let's call the transposed matrix tB. Now matrix multiplication involves the dot product of a row of A and a row of tB. In both cases the elements are stored next to each other. Of course it takes a little work to transpose B, but it is worth it if the matrices are big.

Another reason for a program to run slow is if the executable becomes larger than the available RAM (Random Access Memory). If this is the case, the executable has to be swapped in and out of memory, and this slows things down. So make sure you're program isn't too big. Efficient use of memory is one of the benefits of C++.