Exam I on Thurs.

12-15 questions.

About 1/3 short answer, 1/3 code reading, 1/3 code writing.

Review questions and key are on the website.

Program Development

There are several basic approaches, which can be used in different circumstances, and are often combined in various ways.

Top-down Development

You start from the goal or task that you want to accomplish and break it down into smaller tasks. And keep doing this until to get to tasks that can be done directly (either you already have software to do it, e.g. in a library, or it is built into the programming language).

Especially when you are getting used to programming, this is a good approach: Think about how you would write out instructions for a very dull person to do the task. In other words you can't assume the "execution vehicle" understands anything; you must be explicit about everything.

To do this, you may have to introspect and look and see how you would do it yourself.

Wherever there is a word, or process, or task that needs to be explained, that suggests that that task might be a separate function. It should be a well-defined task, usually something that can be stated in a simple sentence (which also makes good documentation).

One problem: if you apply this process somewhat blindly, you may end up implementing subtasks that are nearly identical, which is usually a waste of effort. E.g., sort ints and sort doubles. This is even more of a problem if there is a team of programmers who are working independently.

The positive aspect of TD development is that you can break a large project into smaller projects that can be implemented independently. The negative is that through lack of coordination the separate teams may duplicate effort.

One way to solve this, is to look ahead (or "look down") and see where you are headed. This is where you can combine TD with bottom-up.

Bottom-up design
We combine what we have already got, assembling it to make other useful things. The problem is, that you have to know what's useful.

In practice we often work TD and BU at the same time, or in successive passes.

Iterative Development:
A useful technique that can be used with either TD or BU design. The idea is that you always have a functioning program, but the early versions don't do everything. So you incrementally add function (capability) to the program, and test as you go. If you discover an error, the problem is most likely in the part you just added. The error may be in the statements or functions you just added, or in their interaction with what was there previously.

The idea is cognitive economy. Our brains aren't so big, so we don't want to have to deal with more than we can hold in our brain at one time. Each part (in TD, BU, or Incr. design) is small enough that you can wrap you brain around it. This will

increase as you get more experience, but not a whole lot, which is why very experienced programmers still say a function should not be longer than about a page.

Scaffolding:
This is stuff you put in your program to help to get it working.
Most common: output statements, so you can see what's going on.
This may require additional variables to hold intermediate results, but that's probably a good idea anyway, because it make your program more readable.

You can deactivate and reactivate output statement by commenting them out. Better than having to retype them.

When you are done debugging a piece of code, you may decide to remove the scaffolding, but think twice about it. You may need to go back and recheck that part of your program. A lot of commented-out code is kind of ugly, and perhaps confusing, but it can be worthwhile to keep in, at least until the code is final.

Which never happens! Programs, if they are used, will be modified and require maintenance. This is why readability and comprehensible design are so important in software engineering.

It's sometimes better to put in "debug flags" that cause debugging code to be executed whenever the flag is set.

```
bool debug = false;
….
if (debug) {
      output some useful information
```

}

This takes a little memory space and a little computer time, but that usually doesn't matter.

Moral: Put in lots of output statements so you can see what your program is doing.

Recursion:
Recursion is when a particular task, working on some problem, needs to do the same task, working on a different problem. E.g., look a word up and summarize the definition.

In programming terms, a recursive function is a function that needs to call itself, but with different input. We have to beware of infinite or unending recursion. The main way to do this is to make sure we getting nearer each time to a stopping condition.

Consider factorial:

E.g., 6! = 6 * 5 * 4 * 3 * 2 * 1

More general: N! = N * (N - 1) * (N - 2) * … * 2 * 1

To be more precise, note how we can reduce N! to a simpler problem, (N - 1)!.

N! = N * (N - 1) * (N - 2) * … * 2 * 1
= N * [(N - 1) * (N - 2) * … * 2 * 1]
= N * (N - 1)!

What is the stopping condition? N=1 is an obvious stopping condition. 1! = 1.

A better stopping condition is 0! = 1, because this makes factorial work for zero (so I don't have to check for 0 as a special case, and for other mathematical reasons).

Recursive definition:

N! = 1, if N = 0

N! = N * (N - 1)!, if N > 0

Easy to translate to C++:

```
int fac (int N) {
    if (N == 0) {
        return 1;
    } else {
        return N * fac(N - 1);
    }
}
```

How do we handle N < 0?

Cheap solution:

Put it in the documentation.

```
// fac(N) works only if N >= 0
```

Better solution:

```
int fac (int N) { // N >= 0
    if (N == 0) {
```

```
        return 1;
    } else if (N > 0) {
        return N * fac(N - 1);
    } else {
        cout << "No factorials of negative numbers!\n";
        return 0; // because fac has to return something
}
```

I could also use an "assert statement".

```
#include <assert.h>
…
int fac (int N) { // N >= 0
    assert (N >= 0); // will stop program if N < 0
    if (N == 0) {
        return 1;
    } else {
        return N * fac(N - 1);
}
```