How to approach a computational problem

A lot of people find computer programming difficult, especially when they first get started with it. Sometimes the problems are problems specifically related to writing programs – difficulties in dealing with the exact syntax the computer demands from the programmer. A lot of the time, though, this is not the case, and the problem is more basic. One major source of difficulty that beginning programmers will often have is that they try to write the computer program before they have properly thought about what they are trying to do. This sort of impatience is human nature, especially if you're learning to program in a computer lab, while sitting in front of the computer terminal. It's also very bad practice and can lead to a lot of frustration.

As you first get started with programming, you should follow a few steps before you start typing on the computer. As you become more experienced, you may decide to take a more relaxed approach, and for simple tasks, that more relaxed approach may lead to your getting the job done faster. For complicated tasks, though, you should always follow certain elements of "good practice".

Subroutines and pseudocode

We often refer to a finished computer program as a piece of "code." The essence of a computer program is that it is a means of encoding an algorithm – a step-by-step procedure for performing some function, usually a calculation – into a language that a computer can "understand."[1] Very often, what will happen for beginning (and even experience) programmmers is that the computer code is a correct implementation of an incorrect algorithm.

We need to remember that, without programming, computers can do only a small handful of things – some basic input and output tasks, some shifting around of data from one location in the computer's memory to another, and some basic arithmetic. Once we remember that the computer can do only these things, but can do them perfectly,[2] we understand the level of detail in which we need to develop our algorithms in order to get them to work out.

What you need to do, then is to take the larger task you wish to accomplish, and break it down into a set of smaller tasks. Eventually you will wish to get to tasks so small that the computer can understand them. You will have the most success in achieving your goals if you do this *before* you sit down at the computer and start typing in a program.

There are a few key "buzzwords" for this sort of algorithm development – *structured code*, *subroutines*, and *pseudocode*.

First, let's discuss "structured code". The nature of a structured code is that it is built up from a main body with a set of subprograms, called subroutines. The main body exists almost exclusively for calling the subroutines to be run – and sometimes for determining whether enough of the subroutines have been run. For very simple programs, one can get away with doing a bit more in the

[1] As we will discuss later, we actually need to take our codes written in a "high level" programming language and convert them into machine language before the computer can really "understand" them.

[2] We will discuss round-off errors in computer arithmetic later in the course – computers actually don't do certain types of math perfectly

main body, but as programs get more complicated, the main body really needs to be kept as simple as possible, with the complexity moved to the subroutines.

The modularity of structured codes leads to several key advantages. During the programming process, it becomes possible to split tasks up among several programmers, if a code is especially complicated. Additionally, certain subroutines are useful for a variety of purposes, and one can often find these subroutines already written (e.g. the book *Numerical Recipes* contains a lot of computer codes that can be applied to various tasks, mostly relevant to scientific computing). There is also a package called the GNU Scientific Library that contains libraries of routines written for use with C/C++ programs.

Consider the program `compute_sines.cc`. You can look at this code and see immediately that it first calls a subroutine `readinvalues` and then calls a subroutine `printoutsines`. You do not need to be an expert in computer programming to figure out what the basics of this program are, especially since I have put comments in the subroutines. You can also see that if we wanted to allow the user to decide whether to take a sine or a cosine wave, we could have put in another subroutine, almost like the `printoutsines` routine, but with a cosine function in it and called it `printoutcosines`[3]. We then could have included sine or cosine as an option in the routine `readinvalues`, and a "control structure" in the main body to use the user's choice to decide which of the subroutines. The main body of the code would get only slightly longer, even as the code got much longer.

The other main advantage is that two programmers could work simultaneously on this program – one could write the input subroutine and one could write the output subroutine, and then the full program could be put together later on, and a third programmer could be added to write the cosine routine, if that were deemed valuable. This program, and indeed, most of what you are likely to do as an undergraduate, is simple enough that the amount of time spent discussing exactly how to separate the work and how to make sure that the pieces fit together at the end might not be justified by the additional in person-power to the task[4] – but you can at least see how a nice structured program makes this much easier to do.

Pseudocode

So: how do we get to the point of writing a nice structured program, and coding it in correctly? This, plus some basic understanding of some numerical methods, is the whole point of our course. A lot of it will just take practice. People who spend hours and hours writing computer programs tend to become good at it. So, think about this once in a while when you are doing your homework for other classes, or are working on lab reports, or anything else that requires some mathematical analysis – and maybe perform a computational

[3] We could call it `Fred` too, if we wanted to, but we generally like to give subroutines names that help describe what they do. But the program would still work with `Fred`

[4] There is an influential book in software engineering called *The Mythical Man-Month* which suggests that sofware projects should generally be done with the fewest people possible because having too many people working on the project means that too much time is spent communicating about the project and not enough time is spent doing it.

solution to something, or use a computer program to model your data.

Let's also remember the words of NFL Hall of Famer Vince Lombardi, who said, "Practice does not make perfect. Only perfect practice makes perfect." There are some ways to approach writing a computer program that will allow you to do a better job, right from the start.

The most important of these is writing a "pseudocode" before you start programming. A pseudocode is a description of what your program will do that is written out in plain English (with some mathematical notation where that's easier to understand than plain English). A good pseudocoding approach will start from a top-down approach – give a list of fairly vague tasks that are "black boxes". Each of those black boxes is then specified further later on. The top page then becomes the main body of the program, and each of the black boxes becomes a subroutine. Some of the subroutines may need further subroutines, of course.

In class, we will discuss the example of writing a pseudocode to make a peanut butter and jelly sandwich.

Debugging

Writing computer programs is not easy for most people at first, but it's something that people generally become better at as they get more experience. A lot of what you have to do is "trial and error" type work, but there are some strategies that can make it go more quickly and less painfully.

There are two results of making a mistake in writing your program. The first result is that the program fails to be compiled. This is usually the easy kind of problem to fix, because the compiler usually gives you some advice in the form of an error message, telling you what went wrong and in which line of the program it went wrong. Often, in C, the problem will be as simple as a missing semi-colon. Occasionally, the real problem will be elsewhere in the code and will show up when it causes a fatal error somewhere else.

The other problem which can arise, which is harder to fix, is when your program compiles and runs, but gives the wrong answer. In this case, there are three main classes of mistakes:

1. Mistakes in designing the algorithm – that is, your plan for how the computer should follow simple steps and get the answer you're looking for is incorrect.

2. Mistakes in coding the algorithm into the computer – that is, the idea you have about how to break the problem up into simple steps is correct, but you have made some error in terms of translating that idea from pseudocode into actual computer code.

3. Typographical errors – these can include typographical errors in writing the computer code, and also sometimes errors in the input data set you send the computer.

When in doubt, print it out!

When you are trying to figure out where you have made a mistake in your program, you should print out the results of the program to the screen, or to

a file that you can look at, or even send to a printer to get a paper copy. Not only that, you should print out a lot of things that you wouldn't normally need to see – e.g. the intermediate results of calculations; lines that say you entered a loop, and that you exited a loop; etc. Print out as much stuff as possible, and then follow the problem through a simple example that you can calculate by hand. Find where what you're doing by hand differs from what the computer is giving you. If at all possible, print out so much stuff that you locate the mistake to within a few lines of computer code.

Getting started

We will now write the first C program that most people write, "Hello World", called `helloworld.cc` in the course web page.

```
/* Hello World program */

#include<iostream>
using namespace std;
main()
{
cout << "Hello World\n";

}
```

There are several key syntax things to notice about this program. First, I have set apart the first line as a comment that tells me something about what the program does. Second, I have included the standard input-output library. Third, I have a section starting with `main()` – this is the main body of the program. Fourth, I have printed something out within the main body of the program. Note that there is a `\n` in the `cout` statement. That signifies to go to a new lines afterwards.

Next, let's try to move to a structured programming approach. Let's make that printout go inside a function.

```
/* Hello World program -- with a function */

#include<iostream>
using namespace std;

void helloworld()
{
cout << "Hello World\n";
return;
}


main()
{
helloworld();
}
```

Now we have `void helloworld()` before the main body of the program. This is a function. The function is of "type" void.

An aside: data types

Functions and variables have "types" in C. The basic types are `char, int, float`, and `double`, for characters (i.e. things which may not be numbers), in-

tegers, floating point numbers (i.e. real numbers), and double precision floating point numbers.[5] It is fine to use a `float` to store an integer, but if only integer operations will be done, that will waste memory and CPU time.

Back to functions

A void function is a function that doesn't return a value. Other functions sometimes will return values and store them in variables in the main program. Functions can also have variables that are passed to the function. There aren't any of those for the `helloworld` function, either, but if there were some, they would go in the parentheses.

Next, we see that the main body of the program calls `helloworld`. Then the program ends.

Reading in input from the user

OK - now let's say we want to print out "Hello World" a bunch of times, and we want to let the user decide how many times. Then we need to figure out how to take input from the user. We use the `cin` command here.

Take a look at the program `helloworld3.cc` now:

```
/* Hello World program -- with a function */

#include <iostream>
using namespace std;
void helloworld()
{
cout << "Hello World\n";
return;
}


main()
{
int i,numtimes;
cout << "How many times do you want to print out a message?\n";
cin >> i;
for (i=1;i<=numtimes;i=i+1)
{
helloworld();
}
}
```

We have to declare the type of the variable `numtimes`[6] before we use it.

[5] We'll get to the idea of precision later on, but double precision variables use twice as much of the computer's memory, and sometimes take longer to be used in computations, but they are less susceptible to round-off errors.

[6] We also declare a type for `i` which we will use just a bit later.

Control structures

The next thing that we want to do in programs is to be able to have some process where the computer can use some criteria to decide whether to keep repeating a statement or not to. There are three of these in C: the `for` statement, the `while` statement and the `do while` statement.

When we have a case where we want to do something a specific number of times the `for` statement is the easiest way to go. We have a first line with the starting point for a variable, the ending point for the variable, and the way we change the variable as we go through the loop.

`for` *(initial;final;increment)*
{
set of tasks
}

Then we have a set of curly braces which include all the tasks that we complete. At the end of the set of curly braces, we go back to the beginning of the loop, after incrementing the variable by the amount specified in the *increment* statement.

Sometimes we wish to have conditions that aren't easily specified in a `for` loop. We then have a couple other options: the `while` loop and the `do while` loop. These are almost identical to one another – the only difference is that the `do while` loop runs through one time before checking whether the condition to continue running is met. The syntaxes are:

`do`
{
set of tasks
}
`while` *(condition is true)*;
and

`while` *(condition is true)*
{
Set of tasks
}

There are many reasons for physics-related tasks why you might prefer to use a `while` type loop rather than a `for` loop. For example, suppose you were computing the trajectory of a projectile, and you wanted to stop your calculation when the projectile hit the ground. If you already knew ahead of time how long the projectile would be in the air (e.g. if you were neglecting air resistance) then you could do this with a for loop – but then you probably wouldn't need a computer to solve the problem. If you don't know how long the projectile will be in the air, then you wouldn't know what to make the maximum value of the

loop.

Conditional statements

The other kind of control structures we'll want to work with are "conditionals" – i.e. tasks that we only execute one time, and only if certain conditions are met. In principle, we can always do this with a `while` loop, and the other ways of coding things I will show you here just make things easier.

The workhorse command is the `if` command. The `if` command allows also for `else` and `else if` statements, which, again, are unnecessary, but often quite convenient. The following snippet of code could be used in a computer blackjack game:

```
   if (total==21)
{cout << ‘‘Blackjack!’’}
else if (total>21)
{cout << ’’Busted!’’}
else if (total<21)
newcard(deck[]);
```

There is also a syntax involving the commands `switch` and `case`, which can make the coding for some tasks a bit tidier and easier to read (but really not any easier to program), but which is not valuable enough to be worth the time to teach in this class. If you have a menu of tasks you wish to run through, you should feel free to google the syntax for using these commands and decide if you wish to use them.