2.13 — How to design your first programs

Now that you've learned some basics about programs, let's look more closely at *how* to design a program.

When you sit down to write a program, generally you have some kind of idea, which you'd like to write a program for. New programmers often have trouble figuring out how to convert that idea into actual code. But it turns out, you have many of the problem solving skills you need already, acquired from everyday life.

The most important thing to remember (and hardest thing to do) is to design your program *before you start coding*. In many regards, programming is like architecture. What would happen if you tried to build a house without following an architectural plan? Odds are, unless you were very talented, you'd end up with a house that had a lot of problems: walls that weren't straight, a leaky roof, etc… Similarly, if you try to program before you have a good game-plan moving forward, you'll likely find that your code has a lot of problems, and you'll have to spend a lot of time fixing problems that could have been avoided altogether with a little thinking ahead.

A little up-front planning will save you both time and frustration in the long run.

In this lesson, we'll lay out a generalized approach for converting ideas into simple functional programs.

Design step 1: Define your goal

In order to write a successful program, you first need to define what your goal is. Ideally, you should be able to state this in a sentence or two. It is often useful to express this as a user-facing outcome. For example:

- Allow the user to organize a list of names and associated phone numbers.
- Generate randomized dungeons that will produce interesting looking caverns.
- Generate a list of stock recommendations for stocks that have high dividends.
- Model how long it takes for a ball dropped off a tower to hit the ground.

Although this step seems obvious, it's also highly important. The worst thing you can do is write a program that doesn't actually do what you (or your boss) wanted!

Design step 2: Define requirements

While defining your problem helps you determine *what* outcome you want, it's still vague. The next step is to think about requirements.

Requirements is a fancy word for both the constraints that your solution needs to abide by (e.g. budget, timeline, space, memory, etc…), as well as the capabilities that the program must exhibit in order to meet the users' needs. Note that your requirements should similarly be focused on the "what", not the "how".

For example:

- Phone numbers should be saved, so they can be recalled later.
- The randomized dungeon should always contain a way to get from the entrance to an exit.
- The stock recommendations should leverage historical pricing data.
- The user should be able to enter the height of the tower.
- We need a testable version within 7 days.
- The program should produce results within 10 seconds of the user submitting their request.
- The program should crash in less than 0.1% of user sessions.

A single problem may yield many requirements, and the solution isn't "done" until it satisfies all of them.

Design step 3: Define your tools, targets, and backup plan

When you are an experienced programmer, there are many other steps that typically would take place at this point, including:

- Defining what target architecture and/or OS your program will run on.
- Determining what set of tools you will be using.
- Determining whether you will write your program alone or as part of a team.
- Defining your testing/feedback/release strategy.
- Determining how you will back up your code.

However, as a new programmer, the answers to these questions are typically simple: You are writing a program for your own use, alone, on your own system, using an IDE you downloaded, and your code is probably not used by anybody but you. This makes things easy.

That said, if you are going to work on anything of non-trivial complexity, you should have a plan to backup your code. It's not enough to just zip or copy the directory to another location on your machine (though this is better than nothing). If your system crashes, you'll lose everything. A good backup strategy involves getting a copy of the code off of your system altogether. There are lots of easy ways to do this: Zip it up and email it to yourself, upload it to a cloud storage service (e.g. Dropbox), use a file transfer protocol (e.g. SFTP) to upload it to another machine, copy it to another machine on your local network, or use a version

control system residing on another machine or in the cloud (e.g. github). Version control systems have the added advantage of not only being able to restore your files, but also to roll them back to a previous version.

Design step 4: Break hard problems down into easy problems

In real life, we often need to perform tasks that are very complex. Trying to figure out how to do these tasks can be very challenging. In such cases, we often make use of the **top down** method of problem solving. That is, instead of solving a single complex task, we break that task into multiple subtasks, each of which is individually easier to solve. If those subtasks are still too difficult to solve, they can be broken down further. By continuously splitting complex tasks into simpler ones, you can eventually get to a point where each individual task is manageable, if not trivial.

Let's take a look at an example of this. Let's say we want to clean our house. Our task hierarchy currently looks like this:

Clean the house

Cleaning the entire house is a pretty big task to do in one sitting, so let's break it into subtasks:

Clean the house
- Vacuum the carpets
- Clean the bathrooms
- Clean the kitchen

That's more manageable, as we now have subtasks that we can focus on individually. However, we can break some of these down even further:

Clean the house
- Vacuum the carpets
- Clean the bathrooms
  - Scrub the toilet (yuck!)
  - Wash the sink
- Clean the kitchen
  - Clear the countertops
  - Clean the countertops
  - Scrub the sink
  - Take out the trash

Now we have a hierarchy of tasks, none of them particularly hard. By completing each of these relatively manageable sub-items, we can complete the more difficult overall task of cleaning the house.

The other way to create a hierarchy of tasks is to do so from the **bottom up**. In this method, we'll start from a list of easy tasks, and construct the hierarchy by grouping them.

As an example, many people have to go to work or school on weekdays, so let's say we want to solve the problem of "go to work". If you were asked what tasks you did in the morning to get from bed to work, you might come up with the following list:

- Pick out clothes
- Get dressed
- Eat breakfast
- Travel to work
- Brush your teeth
- Get out of bed
- Prepare breakfast
- Get on your bicycle
- Take a shower

Using the bottom up method, we can organize these into a hierarchy of items by looking for ways to group items with similarities together:

Get from bed to work
- Bedroom things
    - Turn off alarm
    - Get out of bed
    - Pick out clothes
- Bathroom things
    - Take a shower
    - Get dressed
    - Brush your teeth
- Breakfast things
    - Make coffee or tea
    - Eat cereal
- Transportation things
    - Get on your bicycle
    - Travel to work

As it turns out, these task hierarchies are extremely useful in programming, because once you have a task hierarchy, you have essentially defined the structure of your overall program. The top level task (in this case, "Clean the house" or "Go to work") becomes main() (because it is the main problem you are trying to solve). The subitems become functions in the program.

If it turns out that one of the items (functions) is too difficult to implement, simply split that item into multiple sub-items/sub-functions. Eventually you should reach a point where each function in your program is trivial to implement.

Design step 5: Figure out the sequence of events

Now that your program has a structure, it's time to determine how to link all the tasks together. The first step is to determine the sequence of events that will be performed. For example, when you get up in the morning, what order do you do the above tasks? It might look like this:

- Bedroom things
- Bathroom things
- Breakfast things
- Transportation things

If we were writing a calculator, we might do things in this order:

- Get first number from user
- Get mathematical operation from user
- Get second number from user
- Calculate result
- Print result

At this point, we're ready for implementation.

Implementation step 1: Outlining your main function

Now we're ready to start implementation. The above sequences can be used to outline your main program. Don't worry about inputs and outputs for the time being.

```
int main()
{
//    doBedroomThings();
//    doBathroomThings();
//    doBreakfastThings();
//    doTransportationThings();

    return 0;
}
```

Or in the case of the calculator:

```
int main()
{
    // Get first number from user
//    getUserInput();

    // Get mathematical operation from user
//    getMathematicalOperation();

    // Get second number from user
//    getUserInput();

    // Calculate result
//    calculateResult();

    // Print result
//    printResult();

    return 0;
}
```

Note that if you're going to use this "outline" method for constructing your programs, your functions won't compile because the definitions don't exist yet. Commenting out the function calls until you're ready to implement the function definitions is one way to address this (and the way we'll show here). Alternatively, you can *stub out* your functions (create placeholder functions with empty bodies) so your program will compile.

Implementation step 2: Implement each function

In this step, for each function, you'll do three things:

1. Define the function prototype (inputs and outputs)
2. Write the function
3. Test the function

If your functions are granular enough, each function should be fairly simple and straightforward. If a given function still seems overly-complex to implement, perhaps it needs to be broken down into subfunctions that can be more easily implemented (or it's possible you did something in the wrong order, and need to revisit your sequencing of events).

Let's do the first function from the calculator example:

```
#include <iostream>

// Full implementation of the getUserInput function
int getUserInput()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}

int main()
{
    // Get first number from user
    int value{ getUserInput() }; // Note we've included code here to test the return
value!
    std::cout << value << '\n'; // debug code to ensure getUserInput() is working,
we'll remove this later

    // Get mathematical operation from user
//    getMathematicalOperation();

    // Get second number from user
//    getUserInput();

    // Calculate result
//    calculateResult();

    // Print result
//    printResult();

    return 0;
}
```

First, we've determined that the *getUserInput* function takes no arguments, and will return an int value back to the caller. That gets reflected in the function prototype having a return value of int and no parameters. Next, we've written the body of the function, which is a straightforward 4 statements. Finally, we've implemented some temporary code in function *main* to test that function *getUserInput* (including its return value) is working correctly.

We can run this program many times with different input values and make sure that the program is behaving as we expect at this point. If we find something that doesn't work, we know the problem is in the code we've just written.

Once we're convinced the program is working as intended up to this point, we can remove the temporary testing code, and proceed to implementation of the next function (function *getMathematicalOperation*). We won't finish the program in this lesson, as we need to cover some additional topics first.

Remember: Don’t implement your entire program in one go. Work on it in steps, testing each step along the way before proceeding.

Related content

We cover testing in more detail in lesson 9.1 -- Introduction to testing your code.

Implementation step 3: Final testing

Once your program is “finished”, the last step is to test the whole program and ensure it works as intended. If it doesn’t work, fix it.

Words of advice when writing programs

Keep your programs simple to start. Often new programmers have a grand vision for all the things they want their program to do. “I want to write a role-playing game with graphics and sound and random monsters and dungeons, with a town you can visit to sell the items that you find in the dungeon”. If you try to write something too complex to start, you will become overwhelmed and discouraged at your lack of progress. Instead, make your first goal as simple as possible, something that is definitely within your reach. For example, “I want to be able to display a 2-dimensional field on the screen”.

Add features over time. Once you have your simple program working and working well, then you can add features to it. For example, once you can display your field, add a character who can walk around. Once you can walk around, add walls that can impede your progress. Once you have walls, build a simple town out of them. Once you have a town, add merchants. By adding each feature incrementally your program will get progressively more complex without overwhelming you in the process.

Focus on one area at a time. Don’t try to code everything at once, and don’t divide your attention across multiple tasks. Focus on one task at a time. It is much better to have one working task and five that haven’t been started yet than six partially-working tasks. If you split your attention, you are more likely to make mistakes and forget important details.

Test each piece of code as you go. New programmers will often write the entire program in one pass. Then when they compile it for the first time, the compiler reports hundreds of errors. This can not only be intimidating, if your code doesn’t work, it may be hard to figure out why. Instead, write a piece of code, and then compile and test it immediately. If it doesn’t work, you’ll know exactly where the problem is, and it will be easy to fix. Once you are sure that the code works, move to the next piece and repeat. It may take longer to finish writing your code, but when you are done the whole thing should work, and you won’t have to spend twice as long trying to figure out why it doesn’t.

Don’t invest in perfecting early code. The first draft of a feature (or program) is rarely good. Furthermore, programs tend to evolve over time, as you add capabilities and find better ways to structure things. If you invest too early in polishing your code (adding lots of documentation, full compliance with best practices, making optimizations), you risk losing all of that investment when a code change is necessary. Instead, get your features minimally working and then move on. As you gain confidence in your solutions, apply successive layers of polish. Don’t aim for perfect -- non-trivial programs are never perfect, and there’s always something more that could be done to improve them. Get to “good enough” and move on.

Optimize for maintainability, not performance. There is a famous quote (by Donald Knuth) that says “premature optimization is the root of all evil”. New programmers often spend far too much time thinking about how to micro-optimize their code (e.g. trying to figure out which of 2 statements is faster). This rarely matters. Most performance benefits come from good program structure, using the right tools and capabilities for the problem at hand, and following best practices. Additional time should be used to improve the maintainability of your code. Find redundancy and remove it. Split up long functions into shorter ones. Replace awkward or hard to use code with something better. The end result will be code that is easier to improve and optimize later (after you’ve determined where optimization is actually needed) and fewer bugs. We offer some additional suggestions in lesson 3.10 -- Finding issues before they become problems.

> A complex system that works is invariably found to have evolved from a simple system that worked
>
> —John Gall, *Systemantics: How Systems Really Work and How They Fail p. 71*

Conclusion

Many new programmers shortcut the design process (because it seems like a lot of work and/or it’s not as much fun as writing the code). However, for any non-trivial project, following these steps will save you a lot of time in the long run. A little planning up front saves a lot of debugging at the end.

Key insight

Spending a little time up front thinking about how to structure your program will lead to better code and less time spent finding and fixing errors.

> I would say this is arguably the most important thing in programming and some of us, like me at first, took it for granted.
>
> —Reader Emeka Daniel, *comment on learncpp.com*

As you become more comfortable with these concepts and tips, they will start coming more naturally to you. Eventually you will get to the point where you can write entire functions (and short programs) with minimal pre-planning.