# 1.10b — How to design your first programs

BY ALEX ON SEPTEMBER 5TH, 2009 | LAST MODIFIED BY ALEX ON NOVEMBER 17TH, 2017

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 sort of problem that you'd like to solve, or situation that you'd like to simulate. 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.

**Step 1: Define the problem**

The first thing you need to figure out is what problem your program is attempting to solve. Ideally, you should be able to state this in a sentence or two. It can also be useful to express these as an outcome (what outcome are you looking to achieve?). For example:

- I want a better way to keep track of my friends' phone numbers.
- I want to generate randomized dungeons that will produce interesting looking caverns.
- I want recommendations on which stocks I should buy.
- I want to model the height of a ball dropped off of a tower.

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!

**Step 2: Collect 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 have 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 in 7 days.

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

A well-defined set of requirements, given to a competent engineer, should allow that engineer to build the program for you. But that's no fun!

**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 purchased or 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: Email it to yourself, copy it to Dropbox or another cloud service, FTP 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.

**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 write a report on carrots. Our task hierarchy currently looks like this:

- Write report on carrots

Writing a report on carrots is a pretty big task to do in one sitting, so let's break it into subtasks:

- Write report on carrots
    - Do research on carrots
    - Write outline
    - Fill in outline with detailed information about carrots
    - Add table of contents

That's more manageable, as we now have subtasks that we can focus on individually. However, in this case, "Do research on carrots" is somewhat vague, so we can break it down further:

- Write report on carrots
    - Do research on carrots
        - Go to library and get book on carrots
        - Look for information about carrots on internet
        - Take notes on relevant sections from reference material
    - Write outline
        - Information about growing
        - Information about processing
        - Information about nutrition
    - Fill in outline with detailed information about carrots
    - Add table of contents

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 writing a report on carrots.

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 "get from bed 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
- Drive to work
- Brush your teeth
- Get out of bed
- Prepare breakfast
- Get in your car

- 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
    - Get out of bed
    - Pick out clothes
    - Get dressed
  - Bathroom things
    - Take a shower
    - Brush your teeth
  - Breakfast things
    - Prepare breakfast
    - Eat breakfast
  - Transportation things
    - Get in your car
    - Drive 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, “Write a report on carrots” or “Get from bed 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.

**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:

- Get out of bed
- Pick out clothes
- Take a shower
- Get dressed
- Prepare breakfast
- Eat breakfast
- Brush your teeth
- Get in your car
- Drive to work

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

This list essentially defines what will go into your main() function:

```
int main()
{
    getOutOfBed();
    pickOutClothes();
    takeAShower();
    getDressed();
    prepareBreakfast();
    eatBreakfast();
    brushTeeth();
    getInCar();
    driveToWork();
```

```
}
```

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();
}
```

Note that if you're going to use this "outline" method for constructing your programs, it's a good idea to comment each of these out until you actually write them, and then work on them one at a time, testing each as you go. That way the compiler won't complain about them not being defined.

**Step 6: Figure out the data inputs and outputs for each task**

Once you have a hierarchy and a sequence of events, the next thing to do is figure out what input data each task needs to operate, and what data it produces as a result (if any). If you already have the input data from a previous step, that input data will become a parameter. If you are calculating output for use by some other function, that output will generally become a return value.

When we are done, we should have prototypes for each function. In case you've forgotten, a **function prototype** is a declaration of a function that includes the function's name, parameters, and return type, but does not implement the function.

Let's do a couple examples. getUserInput() is pretty straightforward. We're going to get a number from the user and return it back to the caller. Thus, the function prototype would look like this:

```
int getUserInput();
```

In the calculator example, the calculateResult() function will need to take 3 pieces of input: Two numbers and a mathematical operator. We should already have all three of these by the time we get to the point where this function is called, so these three pieces of data will be function parameters. The calculateResult() function will calculate the result value, but it does not display the result itself. Consequently, we need to return that result as a return value so that other functions can use it.

Given that, we could write the function prototype like this:

```
int calculateResult(int input1, int op, int input2);
```

**Step 7: Write the task details**

In this step, for each task, you will write its actual implementation. If you have broken the tasks down into small enough pieces, each task should be fairly simple and straightforward. If a given task still seems overly-complex, perhaps it needs to be broken down into subtasks that can be more easily implemented.

For example:

```
int getMathematicalOperation()
{
    std::cout << "Please enter which operator you want (1 = +, 2 = -, 3 = *, 4 = /): ";

    int op;
    std::cin >> op;

    // What if the user enters an invalid character?
    // We'll ignore this possibility for now

```

```
    return op;
}
```

**Step 8: Connect the data inputs and outputs**

Finally, the last step is to connect up the inputs and outputs of each task in whatever way is appropriate. For example, you might send the output of calculateResult() into an input of printResult(), so it can print the calculated answer. This will often involve the use of intermediary variables to temporarily store the result so it can be passed between functions. For example:

```
// result is a temporary value used to transfer the output of calculateResult()
// into an input of printResult()
int result = calculateResult(input1, op, input2); // temporarily store the calculated result in result
printResult(result);
```

This tends to be much more readable than the alternative condensed version that doesn't use a temporary variable:

```
printResult( calculateResult(input1, op, input2) );
```

This is often the hardest step for new programmers to get the hang of.

A fully completed version of the above calculator sample follows. Note that it uses a few more concepts we haven't covered yet:

- if statements let you execute a line of code if a given condition is true
- the == operator lets you compare two items to see if they are equivalent

You are not expected to understand these at this time (we'll cover these in more detail later). Focus more on the overall flow of the program, and how data moves between the functions.

```
// #include "stdafx.h" // uncomment if using visual studio
#include <iostream>

int getUserInput()
{
    std::cout << "Please enter an integer: ";
    int value;
    std::cin >> value;
    return value;
}

int getMathematicalOperation()
{
    std::cout << "Please enter which operator you want (1 = +, 2 = -, 3 = *, 4 = /): ";

    int op;
    std::cin >> op;

    // What if the user enters an invalid character?
    // We'll ignore this possibility for now

    return op;
}

int calculateResult(int x, int op, int y)
{
    // note: we use the == operator to compare two values to see if they are equal
    // we need to use if statements here because there's no direct way to convert op into the appropria
te operator

    if (op == 1) // if user chose addition (#1)
        return x + y; // execute this line
    if (op == 2) // if user chose subtraction (#2)
        return x - y; // execute this line
    if (op == 3) // if user chose multiplication (#3)
        return x * y; // execute this line
    if (op == 4) // if user chose division (#4)
        return x / y; // execute this line

```

```
    return x + y; // if the user passed in an invalid op, we'll do addition.

    // we discuss better error handling in future chapters
}

void printResult(int result)
{
    std::cout << "Your result is: " << result << std::endl;
}

int main()
{
    // Get first number from user
    int input1 = getUserInput();

    // Get mathematical operation from user
    int op = getMathematicalOperation();

    // Get second number from user
    int input2 = getUserInput();

    // Calculate result and store in temporary variable (for readability/debug-ability)
    int result = calculateResult(input1, op, input2);

    // Print result
    printResult(result);

    return 0;
}
```

**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 2d 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 2d 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, and see it through to completion as much as is possible. It is much better to have one fully 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.

Most new programmers will shortcut many of these steps and suggestions (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 definitely save you a lot of time in the long run. A little planning up front saves a lot of debugging at the end.

The good news is that once you become comfortable with all of these concepts, they will start coming naturally to you without even thinking about it. Eventually you will get to the point where you can write entire functions without any pre-planning at all.