Planning

Once you get past basic programs, you often run into issues of knowing where to start or knowing how to continue past writer's (or I guess it's coder's) block. This document is meant to give some strategies to try to get you started.

WHAT ARE YOU GOING TO MAKE?

This is the first and most basic question you need to ask. Most of the time for our purposes this question is answered for you, but this goes should go beyond a basic one sentence description like "We need code that makes our shooter work." Here are some example questions you can ask to get yourself started:

What tech are you going to be using?

We know what language we are going to be using (Java) and the main library we are going to be using to interact with the hardware (WPILib), but that doesn't answer every question about tech. What does the physical robot look like? What sensors are being used? What other types of parts? What types of controls would you want to use (a button, a joystick…) These types of questions will help you figure out what types of information your code will have and more specifically what it will have to do. If the original plans for the amount/type of sensors don’t make sense for what you need, ask for more in the right places.

What features do you need?

What are the specific things you need to do in order to accomplish your specific goal? Make a list of the minimum requirements that are needed to accomplish the team’s goals for the project. This should give you guidelines you have to meet. This list of requirements should be reviewed by the programming leads as well as the people working on the project in other departments.

What features do you want?

If you could do everything with the project, what else would you add on to your project? This might include automating things that might otherwise be manual, getting extra sensors working, going for more points or some other type of work that is for whatever reason lower priority.

How are you going to test your code?

This may seem super early to ask this question, but it is important to ask it at this point. How do you know when you are done and how do you know that you did it right? This should be pretty close to your requirements list, but it also involves how you test it. For example, if you want your robot to go 10 m/s, how do you set up the conditions for testing, and how do you know that it is going that fast? You can’t just say “Run the robot and see what the speed is”. Is it going straight? Do all the wheels have to go at that speed? How far does it have to go to get to that speed? Does it have to stay at that speed? For how long? Also, how do you know that it is at that speed: is it a readout on the console, or do you need a piece of equipment that you may need to find in the lab ahead of time? How much time will this test take to run?

What is your schedule?

You know that mechanical is limited to build season, but you should know when things on the particular task you are working on are expected to be done. This can help give you an idea of when testing can happen, but also help you know when and how to work best with other departments.

What sort of test mechanism might help?

What smaller version of the mechanism you are working on would help you work on your code? Can you test on an earlier robot? Can they build the mechanism and attach it to an old robot? Can they just attach things to an e-plate? Do you just need sensors? Do you need motors? Think about what is the most complicated thing for the different departments and try it out with a smaller version of just that piece.

HOW WILL YOU IMPLEMENT IT?

If you know what the general idea is, you can actually get into what you are going to do.

What are ways you (or other people) have solved similar problems?

Once you have a good handle on what the mechanism looks like figure out what people have done before. There are usually standard ways to handle different setups. If you find code for a similar situation, that is probably a good way to move forward for this situation.

Write the steps of what you need to do

Write out the steps of what you want to accomplish in doing the work and you can take them one at a time. The first steps of the project may just be getting signals back from the hardware or building a general structure. You may then choose to get it to move at a slower speed or in a simpler pattern. The final steps may be getting it dialed into the right speed or smoothing out how it works. Start simple.

Flow charts

If your project has more complicated data flow or has lots of parts, you may want to draw a flowchart. A flowchart can serve as a picture outline of how your code will work. Each block may be a function, or a piece of hardware.

Pseudocode

Pseudocode, as you may have guessed from the name, is almost like code, but it is not quite code. It is a way of writing code without being distracted by syntax. It is a way to roughly write out your code without having to worry about it compiling, or even what language you are writing in. Since it is for your own planning purposes, there isn't really a wrong way to do pseudocode, but the most useful pseudocode is going to be close enough to the code you will eventually write that it is clear what code you want to write.

Let's go back to a problem and look at different versions of pseudocode to see what works. Here is the problem we're going to look at:

isPrime: return true if a number is prime, or false otherwise.

Here is some pseudocode:

```
isPrime(n)
    if prime
        return true
    return false
```

Some good things about this: it is simple. It looks code shaped without worrying about syntax. But overall this is a pretty bad example of pseudocode because it tells you nothing about how this code is going to work. It pretty much just restates the problem in a slightly different format. So let's look at a better sample of the same code.

```
isPrime(n)
    for i = 2->n-1
        if n%i = 0
            return false
    return true
```

Now this is much better pseudocode. You can tell exactly how it is going to work. It is going to use a for loop and go through every number from 2 through n-1 and check if it evenly divides n. If you find one, it returns false since this number can't be prime. Since this is pseudocode, you don't have to worry about the fact that the variables don't have types, that there are no semicolons, parentheses, or that the if statement doesn't use a double equals. It gets the point across to a person, which is the main point.

This type of planning let's you start a discussion about how you can make your code better. In this case, you probably don't have to go all the way up to n-1, but can get away with only going as high as the square root. Or that you might want to test two separately and go up by twos and test only odd numbers, or find some other way to try to test numbers more likely to be prime numbers.

That is the power of pseudocode: it is easier, while still being close enough to be able to discuss your plans in detail. You won't always be able to iron out all the issues at this stage, but it can help.

STILL STUCK?

Make lists

It is easy to get caught thinking about hypothetical ideas, so you can just start making lists. List out options, list out things to do. Write things down and then you have something concrete to talk about.

Draw pictures

If text doesn’t work well for you, draw pictures, make analogies, write something out that helps you understand how the code works. The important part is that you know how it works and where you are going. However you go about doing that, do it.

Rubber Duck

Talk to someone. It helps to say things out loud. Even if the person you are talking to doesn’t know much about the project or even coding in general. You can even talk to an inanimate object…like say a rubber duck. Sometimes it helps even more to have it be someone who doesn’t know that much about it because then you end up breaking it down into simple language and may end up finding the problem yourself, even if the other person (or thing) doesn’t say a word.

Work towards the middle

List out all the info that you have and list out the things that you need and find ways to connect the two. If starting from one side doesn’t work, working from the end back to the beginning sometimes works better. Be prepared for some dead ends, but a creative new approach can be fun.

Ask questions

Ask questions of people. Ask questions of yourself. Ask questions of other people who know things. Ask questions of your favorite search engine.

Start somewhere

Even if you don’t have all the answers, start somewhere. It doesn’t have to be the beginning. It doesn’t have to be the most important part. Just start somewhere. Write something.