Coding is a difficult and subtle process, fraught with errors and frustration. Most programmers never learn formal methods for proper coding. In this class, we will emphasize and reinforce a coding process known as Incremental and Iterative Design (IID).

Most people compare writing code with writing an essay. It is the use of the word “writing” that causes problems here. Many beginning programmers spend too little time in the planning stages of a programming project and jump right in to writing the actual code (much in the way that students rarely sit down and outline their essays and plan what they are going to say before actually writing it). A much better analogy for programming is creating a blueprint for a building. The result is a series of instructions implementing a complicated algorithm; careful planning is required from the start for it to turn out well. The final part of a project where the code is actually written ends up being a surprisingly small portion of the total time spent.

To continue with the building analogy, the need for planning and preparation is evident when comparing the process of building a dog house versus a four bedroom single family house. The requirements for a dog house are rather simple: A roof, an entrance/exit, and walls. With experience and knowledge of the available tools and materials, a competent builder could easily build a dog house in a day without any plans. The construction methods are simple and the required specifications are shallow—meaning that they are not conflicting—and straight-forward to meet. While the normal engineering process follows the workflow of problem definition → requirements → specifications → implementation, this can easily be skipped with such a simple project with minimal problems.

A house for a family, on the other hand, requires more extensive planning due to the increased requirements and increased flexibility in fulfilling those requirements. First off, the problem definition is broader: design a suitable house for a family of four. Beyond the bare necessities there are additional requirements for creature comforts, some of which can result in conflicting requirements. For example, temperature control is important, so the house should be sealed against the outside in the winter to keep in heat, but allow a substantial breeze in the summer to keep it cool. So while the broad
problem definition is similar between a doghouse and a family house, the specific list of requirements is quite different and these requirements often interact in more interesting ways when being converted to specifications.

Programming projects of sufficient size should be approached carefully using available planning tools before moving to the actual coding phase. Once code has been written or organized in a certain way, it can be difficult and time consuming to alter it (commonly referred to as refactoring), much like it would be expensive to fix a design mistake when construction of the house has already begun. It is worth the upfront effort in generating the projects specifications and construction process before jumping in so that these costly backend modifications can be avoided.

**TECHNIQUES**

Even before we hit the Incremental and Iterative Design process, there are several techniques that are very useful in the pre-planning and software construction stages. While formal versions of these exist, simply keeping these ideas in mind will help a great deal with your coding practices. While most of this falls under the software engineering part of education rather than coding, learning it while you learn to code makes you better at both.

**Decomposition:** When working with the problem definition to generate the list of requirements for the project, a decomposition of the problem into sub-problems can be quite helpful. For example, when considering an entrance to a home, the problem is really several problems combined: how easy should it be to use, how well does it need to keep out the weather, does its requirements change with the season, do animals or pets need to be taken into account, etc. By breaking down the problem into a set of smaller, more specific problems, it can be easier to generate a suitable list of requirements and/or solutions.

Decomposition also holds in the code construction phase, where breaking down the code into simple, testable modules allows you to approach a big problem (difficult) as a series of smaller problems (easy). This will also help to manage the complexity of the problem and confine bugs to smaller and easier to find regions of the code.

**Abstraction/Information Hiding:** Engineering projects are often complicated, involving a number of different parts that can be naturally separated based on functionality. In the example of a house, the most natural separation is by room. Within each room there is the internal structure, power, lighting, exterior/window access, doorways, flooring, heating/cool, etc. By splitting the project's organization into logical modules, each aspect of the planning project becomes smaller and therefore easier to solve.

Separation of the code into modules, which may be organized as separate code libraries, necessitates the definition of a strict interface so that it is clear where one module ends and another begins. Without this interface, there is only a vague definition of how the code is separated. This results in code that appears to be modular because related code
is grouped together, yet the code remains tightly integrated. In practice separating related code into separate files is not enough!

With modules that implement a well-defined interface, it becomes easy to replace modules of code without affecting other modules. This is where the real benefits of modular code become apparent. For example, say during the development of the blueprints for your house you discover that the front door that you were planning on using has been discontinued. While this requires that the blueprints for the front foyer be modified to accommodate a different door, the rest of the house can remain unchanged.

With good modular code, the entire module can be replaced (a new .c file) as long as the interface (defined within the .h file) is respected. This allows you to hide the implementation specifics from the rest of the code. With engineering projects, unforeseen issues are quite common; by splitting the project into well-defined modules, fixing these downstream changes is relatively painless and easy.

**Pseudo-code:** When developing algorithms it is a common practice to use pseudo-code as a means of representing the code. This is a form of code not specific to any language that is very high-level and allows for developing the content of the algorithm without worrying about programming language details. The exact syntax of pseudo-code varies by engineer, but normally it is more akin to staccato English with the intent clear.

For example, the pseudo-code for taking the root sum square (2-norm) of a vector \( y \) of arbitrary length looks like the following:

Set \( x \) to 0
for every element in \( y \):
    Add the square of that element to \( x \)
\( x \leftarrow \) square root of \( x \)

Now in the C programming language it'd look like (assuming \( \text{LEN}_Y \) is the length of \( y \)):

```c
int x = 0;
int i;
for (i = 0; i < \text{LEN}_Y; i++) {
    x += y[i]*y[i];
}
x = \sqrt{x};
```

But in Python it'd look like:

```python
g = 0
for x in y:
    g += x**2
g = \text{math.sqrt}(g)
```

It should be apparent that the algorithm is the same between the languages, but if a specific language is used to write the pseudo-code several disadvantages arise:
• All of your planning is now in a specific language and if a switch is later planned, it may be hard to port your code.

• Your algorithms will be bogged down by language implementation details. It's important to segregate the coding problems from the algorithmic problems. You don't want to be worrying about how to implement the algorithm and what libraries to use at the point when you are just planning the algorithm.

• Finally, it may be harder for members on your team to understand your pseudo-code if they are less familiar with the language chosen for expressing your algorithms.

It is a good practice to include your pseudo-code as the top level block comment for any functions or modules you develop. This way, the pseudo-code guides the actual implementation, and you already have comments added.

**SOFTWARE DESIGN**

At this point, you might have the impression that with careful planning, the various stages of a project look like:

![Flowchart showing planning, coding, and deployment stages]

That is almost never the case, since planning cannot be completely done at the beginning. For example a project’s problem definition may change as a client decided that their needs have changed. It’s possible that materials or tools that were thought to be available are not and suitable replacements must be found. If this project ties in with another project in development, the changing specifications of that project may force changes in yours. In reality the design process will be much more iterative:

![Flowchart with iterative planning, requirements, analysis/design, implementation, testing, evaluation, and deployment stages]
In this case planning is done continually throughout the lifetime of the project as new information is received or existing information changes. This approach is common enough to have a name: *iterative development*. Proper planning is still important and done as early as possible with this approach, but the various stages of planning are expected to be revisited and revised continually throughout the coding stage.

Initial planning should focus on requirements. For example, performance issues must be addressed during the initial planning stages, rather than assuming that the initial design will be fast enough (and defer performance improvements to the final stages of the project as required). Fixing things at the end of the project is expensive and frustrating, as each change can ripple through the previously written code. By planning for the necessary performance from the very beginning, these architectural changes can be made early in the design, or possibly from the start.

With the design requirements for the project clearly defined, the segregation of the project into separate modules follows. Part of this planning process is the separation of the code into modules consolidating related functionality together. Using the house analogy, it is likely that the varying aspects of the house will be grouped depending on function (electrical drawings, plumbing, etc.). Each software module is separated based on related functionality; define the interfaces for each module taking into account the functionality of each module. While this has already been discussed, it bears repeating because the modules and interfaces may require changes over the course of development.

With a proper segregation of the various modules chosen, and their interfaces defined, some thought can now be given to the internal implementation of a module. This includes the various necessary data structures for storing data, the algorithms required to manipulate that data, and any additional internal “helper” functions required for the necessary functionality. Any performance requirements for the module will likely have their greatest impact on the module at this stage, when the necessary data structures and algorithms are chosen. By working through this high-level implementation of the internals of a module, problems may be found that would not be apparent until coding. These problems can now be easily fixed before a single line of code is written and wasted!

**IMPLEMENTATION**

At this point the initial stage of planning is complete. While various planning aspects will likely be revisited for some modules during the implementation stage, the bulk of the work is done. This can sound surprising to new programmers since no code has even been written yet! But after a few times practicing the methods suggested here, it does
become clear that spending significant time in the planning stages of the project really
does pay off.

Just as code organization across modules is important, similar importance should be
placed on following good code organization within a single module or library. With a
language such as C, there is significant flexibility in how code can be written and
organized. So long as the code works and is readable and consistent, the order isn't
terribly important. But by choosing a single consistent method, it makes the code easier
to work with as it is clear where certain things should be. By reducing the things you
need to remember, such as where all those #include statements are, more mental
energy can be spent on the design and coding of your project.

A standard C-source file layout is below. Each section within the source file should start
with a comment indicating the current section. We will provide a pair of template files
(.h and .c) on the class website. In general, public interface items go into the .h file, and
private implementation details go in the .c file. In the .c file you should have:

1. Top level comment indicating copyright license, an explanation of the module
and what it does, example usage, requirements, assumptions made within the
module, etc.
2. #include directives
3. Module-level (private) #define constants and macros
4. Module-level custom (private) datatypes
5. Module-level (private) variables
6. Internal private function prototypes
7. Module code
8. Test harness (conditionally compiled to check code)

With a standard file layout chosen, it is finally time to start writing code. It is here that
incremental development becomes important. As modules consist of a number of
functions, the module should be implemented one function at a time, with testing of
this function being done as soon as it is finished (there will be several iterations of
testing and recoding until it is correct). Typically modules are broken into a group of
separate initialization functions and then a set of functions that do the actual work. You
should first implement the initialization functions and then the other ones in a sensible
order.

Functional testing goes by the common name of unit testing, where the “unit” is
referring to the various units of functionality within the module, generally the individual
functions. Ideally before each function implementation, a unit test is written that can
confirm these functions follow specifications. The general format for unit tests looks as
follows:
• Declare input or initial module state
• Declare expected output or final module state
• Execute sequence of operations configuring the module and executing the code that is being tested
• Compare the output or system state with the expected output or final state declared at the beginning

That is, the unit tests test to be “end-to-end” functionality testing. A robust test suite will also test the range of inputs and error handling. The process of writing tests, implementing functionality, testing, and revising any necessary code repeats until the module is complete. This is then expanded to all necessary modules until the base set of modules are functionally complete.

It is important to have the test harness (usually a main() function) contained within a conditional compilation block (#ifdef … #endif) such that the test harness can be turned off easily without deleting it from the module. It is very important that the test harness remain as part of the module itself, and not a separate file. This becomes even more important when trying to find a bug later on, when retesting a module is required.

At this point the integration process begins, which is again an iterative process much like the unit testing was. Modules are integrated slowly and in small reasonable steps; testing each part as they are combined before moving on to the next step. At the end of each integration step another round of testing should be done. This follows the same concept as the unit testing described above, but tests how modules interact instead.

Integration tests are generally more complicated than unit tests as the interaction between modules can commonly result in unexpected behavior that was not anticipated during the planning and implementation stages. Additionally these tests may rely on hardware components or user interaction which limits the ability to automate them (though, in fact, several companies sell products that specialize in automated software testing). In this case, it may be too much work to write integration tests that persist through the entirety of the development process. In this case integration testing should be done manually or using a separate file for testing the interaction of the code. Again, this can be conditionally included in the project, and returned to as further development takes place if required.

**Conclusion**

Software development, like any other engineering endeavor, benefits a great deal by planning and careful testing. We have introduced Incremental and Iterative Design, and several other software engineering techniques to aide you with this process. Following them will make you a better programmer, and make programming easier. Remember that the old saying remains true: “Failure to plan is planning to fail.”