2.0 Modularization

Sue is frustrated! She is working on a large project with a couple classmates where there must be a thousand lines of code and three dozen functions. Some functions are huge consisting of a hundred lines of code. Some functions are tiny and don't seem to *do* anything. How can she ever make sense of this mess? If only there was a way to map all the functions in a program and judge how large a function should be.

Objectives

By the end of this class, you will be able to:

- Measure the Cohesion level of a function.
- Measure the degree of Coupling between functions.
- Create a map of a program using a Structure Chart.
- Design programs that exhibit high degrees of modularization.

Prerequisites

Before reading this section, please make sure you are able to:

- Create a function in C++ (Chapter 1.4).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).

Overview

A large part of program design is knowing how to split a large program into smaller ones. We call this process **modularization.** There are three tools we use to help us with this process: Cohesion or the measure of the internal strength of a module, Coupling or the measure of the information interchange between modules, and a Structure Chart which is a tool allowing us to visualize the relationship between modules.

Cohesion

A well designed function performs a single task. When this is the case, it becomes easy to use one module for many purposes. Some of these purposes may not have been visualized by the module author. To better understand why Cohesion is important and how to measure it, please view the following videos:

- Modularization: This video will provide a definition of modularization and help us see examples in the world around us.
- Reusability: Why we want to reuse code and how we can make our code more reusable.
- Managing Complexity: The purpose of this video is to illustrate the dangers of allowing complexity to overwhelm a project, and to discuss some tools that will help manage this potential problem.
- Evaluating Modularization: What is Cohesion and why it is an important metric of modularization.
- Levels of Cohesion: One of the most important topics of this semester, the seven levels of Cohesion allows us to characterize the quality of modularization in a function.

The seven levels of Cohesion (from highest to lowest) are Functional, Sequential, Communicational, Procedural, Temporal, Logical, and Coincidental.

Functional

Every item in a function or module is related to a single task

The strongest form of cohesion is Functional. Here all the items in the module are focused on performing a single well-defined task. Observe that it does not matter how simple or complex the task is; it is Functionally cohesive as long as only that task is being performed.

```
float computePay(float hours, float wage)
{
   float pay;

   if (hours < 40.0)
      pay = hours * wage;
   else
      pay = (wage * 40.0) +
            (wage * 1.5 * (hours - 40.0));

   return pay;
}
```

Sequential

Operations in a module must occur in a certain order.
Here operations depend on results generated from preceding operations.

A strong form of cohesion is when operations in a module must occur in a pre-specified order and each operation depends on results generated from preceding modules. The difference between Sequential and Procedural cohesion is that there is no dependence between the operations in procedural. With Sequential, than each operation depends on the one before. In this example, `computeLetter()` depends on results generated from `getGrade()` through the `numberGrade` variable. Similarly `displayLetter()` depends on results generated from `computeLetter()` through the `letterGrade` variable. Because we often need to enforce the sequence of operations like this, it is often necessary to develop modules exhibiting Sequential cohesion.

```
void processGrade()
{
   int  numberGrade;
   char letterGrade;

   // 1. Get the number grade from the user
   numberGrade = getGrade();

   // 2. Convert it from number to letter
   letterGrade = computeLetter(numberGrade);

   // 3. Display the results
   displayLetter(letterGrade);
}
```

Communicational

All elements work on the same piece of data.

Frequently we need to perform many unrelated operations on a single piece of data. If these operations were put in a single function, it would exhibit Communicational cohesion. In the function `processGPA()`, both validation and honor-student adjustments are being performed. These operations are only related in that they work on the same piece of data.

```
float processGPA(float gpa, bool isHonorsStudent)
{
   if (gpa > 4.0)
      cout << "You must be an over-achiever\n";
   if (gpa < 0.0)
      cout << "Talk about a bad grade!\n";

   if (isHonorsStudent)
      gpa += 0.12;

   return gpa;
}
```


Procedural

All related items must be performed in a certain order.

It is often necessary to perform a set of tasks in a certain order. It is certainly important that you take a shower before you put on your socks, and your socks go on before you put on your shoes. If this order is not enforced, then your morning routine will be, well, less effective.

```
void sendLetter()
{
   // 1. Insert the content of the letter
   writeBody();

   // 2. write salutation
   writeSalutation();

   // 3. send letter to recipient
   send();

   return;
}
```

Temporal

Items are grouped in a module because the items need to occur at nearly the same time. What they do or how they do it is not important.

Occasionally we have a collection of task that need to be performed at nearly the same time. Typical examples are those activities that prepare for a task, or those that conclude or clean up a task. Consider your morning routine. You get dressed, you eat, and read your scriptures. It doesn't matter what order they occur in nor are the items particularly related. The only reason why they are put in a module is because they need to be completed before you leave the house. This is an example of Temporal cohesion.

```
void initialize()
{
  float value = 0;
  float pi = 3.14;

  openConfigurationFile();
}
```

Logical

Items are grouped in a module because they do the same kinds of things. What they operate on, however, is totally different.

Another weak form of cohesion is Logical. Here, items are grouped in a module because they do the same kinds of things. What they operate on, however, is totally different. If I were to build a car with logical cohesion, I might put all the motors in one location. This may seem logical; after all, they all do the same thing. They all provide power. However, it really does not make sense to group the engine, the windshield wiper motor, and the power door locks motor in the same location. I doubt anyone will buy my car.

```
void openStuff()
{
  readRecords();
  readNames();

  cin >> value;

  openFile();

  return;
}
```

Coincidental

Items are in a module simply because they happen to fall together. There is no relationship.

The weakest form of cohesion is Coincidental. Here, items are in a module simply because they happen to fall together. There is no relationship. If you find it impossible to develop a concise name for a module, or even a coherent description of what it does, then there is a good chance it has Coincidental cohesion.

```
void doStuff()
{
  readRecords();
  average = totalScore / 10;
  printNames();
  done = true;

  return;
}
```

Coupling

After Cohesion, the next most important modularization consideration is how information flows between functions. We call this measurement "Coupling." Please view each of the Coupling videos:

- What is Coupling: The purpose of this video is to describe Coupling and relate it to the context of everyday experiences.
- Ways to Evaluate Coupling: The purpose of this video is to get us to think how we can measure Coupling and to describe properties of a good taxonomy.
- Levels of Coupling: The purpose of this video is to learn the name of the Coupling levels, what they mean, how to identify the level of a module, and approaches to minimize the Coupling of a given module.

The four levels of Coupling (from loosest to tightest) are Data, Stamp, Control, and External:

Data

The data passed between functions is very simple.
This occurs when a single atomic data item or highly cohesive data items are passed.

The weakest form of coupling is when a function is integrated with the rest of the program through an interface that passes only simple data. This makes it very simple to hook up the function, and makes it very difficult for the function to have any unintended side effects. There are several types of Data coupling. The first is when only a single variable is passed between functions. The second is when an array or string is passed. In this case, the format of the data is trivial and all items are assumed to be cohesive. Since we conceptually think of a string or an array as a single data item, it qualifies as being data coupling. The final example is when cohesive data items are passed. Since all the items are closely related and are part of the same concept, they qualify as a simple Data interface.

```
// a single return value
float getValue();

// a single parameter
void displayAge(int age);

// two parameters relating to a single concept: coordinates
void getCoordinates(int &x, int &y);
```

Stamp

Complex data or a collection of unrelated data items are passed between modules.

Stamp coupling is similar to Data except more complicated data is passed between modules. This is typically in the form of a data structure or a large amount of non-cohesive data. We will learn more about data structures in CS 165. In many ways, Stamp coupling is similar to an overly-complex stereo system with a thousand wires hanging out the back. You need to know what each wire does and how it works to create a new component for this system. Stamp coupling can be difficult to work with. Contrast that to the headphone jack for a portable media player. That interface is closer to Data coupling where everything is reduced to a single wire. What could be easier than that?

```
void display(float income,
             float projLiving,
             float actualTax,
             float actualTithing,
             float actualLiving,
             float actualOther);
```

Control

One module passes data to another that is interpreted as a command.

Control coupling is where one module directs or controls the execution of another module. This is typically performed by passing control information in a variable or data structure. The difference between Control coupling and Data or Stamp is not what is passed, but how the parameters are interpreted. If a parameter tells the function what to do or how to do it, then it is a control parameter. A typical example is a Boolean variable that specifies whether an action is to be performed. These variables are called flags because they are analogous to a flag signal passed between two naval warships so they can coordinate their actions.

```
{
   cout << "Select an option: ";
   char option;
   cin >> option;


   performOption(option);
}
```

External

Two modules communicate through a global variable, a file, or similar external means.

External coupling, along with common and content coupling, are when functions are related through some external medium. This could be a file that more than one function accesses, through global variables or data structures, or when one function is able to alter the inner workings of another function. In each of these cases, the execution of the function has the potential for causing side effects. Side effects greatly increase the complexity of a program and make its behavior extremely difficult to predict. The human body uses external coupling because the organ are so interrelated. If you ever encounter a function that exhibits external coupling, it is typically worth the effort to weaken the coupling or to avoid using the function altogether.

```
int income = 0.0;

void display()
{
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);

   cout << "Income: $";
        << income
        << endl;
}
```