What is Modular Programming

Modular Programming

Introduction

This week, you will explore the use of modules to break up larger problems into small subproblems. You will expand your knowledge of all control structures by creating a design that modularizes code. We will be creating functions in our code.

Now is a good time to go back and review all control structures in the previous weeks' lectures. Once you review the material, let's get started!

Modular Programming

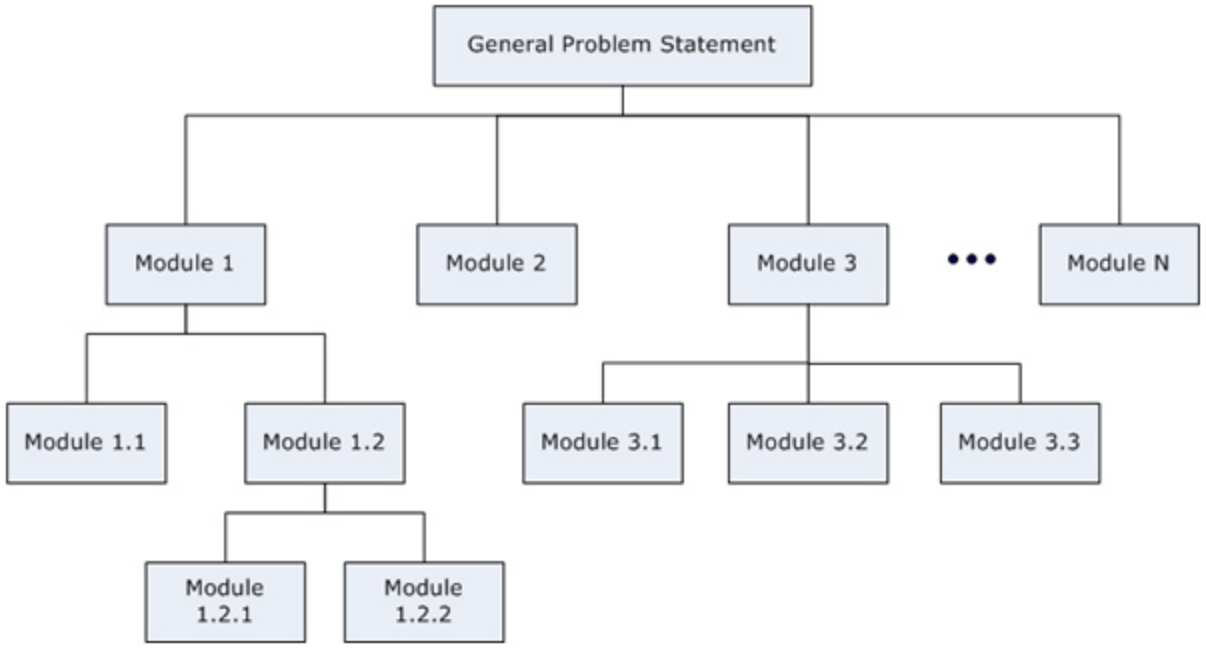
When you are given any type of problem, it usually cannot be solved as a single large problem. For example, if you are going to build a playhouse in your backyard, you don't build the playhouse all at once; you build each of the modules, such as the floor, the walls, and the roof, separately, and then you connect or integrate the modules. This results in a playhouse. This is how you normally solve problems in your daily life: You break the problem down into smaller problems, or modules, that are easier to solve. Then, you integrate the modules to complete the construction. You will find that this is the approach you will take when you analyze, design, and build your programs.

This approach to solving software problems is called *modular design* and involves breaking up the program solution into smaller, more manageable modules; documenting the relationship between the modules using hierarchy charts; and designing the logic of each module using flowcharts or pseudocode. The modules are then assembled into the larger, more complex application. This process greatly simplifies the overall solution to a complex problem, and every real-world software application uses this approach.

You will learn how to break a complex program into modules, and as you move forward, keep in mind that, if you concentrate on smaller problems using the modular approach, you will be able to reduce the overall complexity of the logic and design.

Divide and Conquer!

You typically solve a large problem by breaking it down into more manageable pieces. This is called a divide-and-conquer approach to problem solving, and it is the approach used by every program designer today! You solve the general problem by successively breaking the problem into smaller pieces, which are easier to solve. Then, you put the pieces back together to solve the bigger problem. This process is called *top-down design* and is based on the divide-and-conquer approach to problem solving (see the figure below).



Once you determine the general structure of the large problem, you can determine what step-by-step instructions are required to solve each of the modules and then integrate the modules to solve the general problem. Complex programs are easier to analyze, design, code, and debug when they are refined into smaller, independent modules for the following reasons.

Advantages of Modular Code

- It reduces the complexity of the analysis, design, and coding.
- Different people can work on different modules.
- Modules can be reused.
- It's easier to see the big picture.
- It's easier to desk-check the algorithms.
- It's easier to find errors in the algorithms.

A module is a relatively small, independent block of instructions that performs a single task. Moreover, all of your designs should have a hierarchical organization, as shown in the figure above. That is, we call this a modular design, and the goal is to partition your program into logically independent modules in which each module performs a specific task. Each module will have independent characteristics and as little interaction with the other modules as possible. The principle of modularity is to keep a series of like instructions grouped and treat them as a single unit—a module. Each module performs one specific task (input, calculate tax, calculate pay, enter new data, etc.). This can be a very natural concept that you see and use every day.

The pseudocode for the program consists of main program, input module, process module, and output module.

Modularizing Programs

The first module is usually considered to be the main module. This is the module that will control all other modules. The main is sort of the boss. To execute a different module, the main performs a call to the other module. Therefore, there may be a series of statements that the main will execute. When a call is reached, the main module will transfer control to the called module. The module will execute until it reaches the end of the module (or reaches another call statement). Once it reaches the end of a module, it will transfer control back to the main module, and execution in the main module will continue with the statement immediately following the call statement.

As described above, one of the key characteristics of our modules is that they should be functionally independent. That is, modules should have

- a single-minded function (they do only one thing); and
- no excessive interaction with other modules.

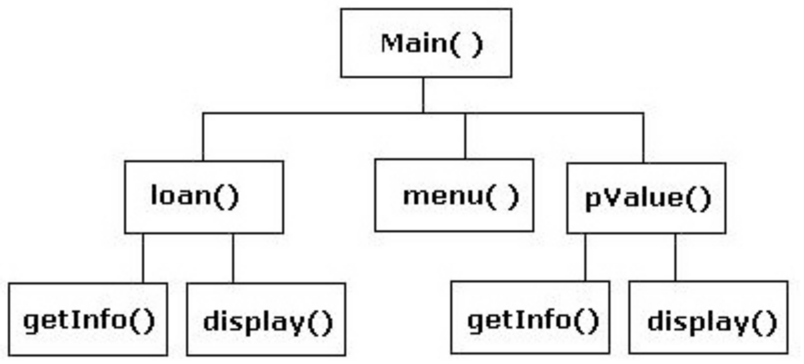
Functional independence is then measured by two very important design characteristics: (1) coupling and (2) cohesion.

- *Coupling* describes how modules in a program are connected. The optimal case is when the modules are minimally dependent; that is, the modules are loosely coupled. Let's look at a home theater example. You know that the interaction between the receiver and the DVD player is, at most, three cables: one for video and two for audio signals. It can be said that the interaction between the DVD player and receiver is low coupling. Low coupling makes your designs much more robust and easy to use. There are many ways to ensure low coupling between modules; three ways are reducing the number of variables passed between modules, avoiding global variables, and minimizing the depth of module calls. These topics are discussed in greater detail below and in the readings.
- *Cohesion* describes the degree to which a module is focused on its purpose. Well-designed modules have *high cohesion,* indicating that the module performs a single task. Modules that perform more than one task have lower cohesion.

In short, a well-defined module is highly cohesive (single minded) and loosely coupled (minimal interaction). You want to strive to make your software designs as functionally independent as possible.

Modularizing a Flowchart

Hierarchy charts are used to identify the relationships between modules within a program. Hierarchy charts do not show any detail other than module names and from where the module is called. Moreover, hierarchy charts do not show loops, variables, absolution, or *If* statements. For example, consider the following hierarchy chart.



First, the hierarchy chart illustrates that the Main() module controls the execution of the program by its ability to call the loan(), menu(), and pValue() modules. The loan() and pValue() modules can call either the getInfo() or display() modules. (Note that getInfo() is the same module in both cases, as is the display() module.)