

# Unit 2. Design & Loops

|  |     |
|--|-----|
| 2.0 Modularization .....               | 108 |
| 2.1 Debugging.....                     | 130 |
| 2.2 Designing Algorithms.....          | 145 |
| 2.3 Loop Syntax .....                  | 156 |
| 2.4 Loop Output .....                  | 168 |
| 2.5 Loop Design.....                   | 181 |
| 2.6 Files.....                         | 190 |
| Unit 2 Practice Test .....             | 207 |
| Unit 2 Project : Calendar Program..... | 209 |

## 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

Up to this point, our programs have been relatively manageable in size and complexity. In other words, one could conceivably keep the entire design in your head. Most interesting software problems, however, are far too large and far too complex for this. Very soon, this become so difficult that it is impossible for any one person or even group of people to understand everything. What is to be done?

One of the main techniques we have at our disposal to tame these size and complexity challenges is **modularization**. Modularization is a collection of tools metrics, and techniques that together enable us to reduce large problems into smaller ones.

The first tool we have at our disposal is the Structure Chart. This is a graphical representation of the functions in a program, including how they “talk” to each other. You may have noticed an example of a structure chart in the Unit 1 project.

The second modularization tool is a metric by which we measure the “strength” of a function. This will tell us the degree in which a given function is dedicated to a single task. We call this metric Cohesion.

The third and final modularization tool is a metric by which we measure the complexity of the information interchange between two functions. We call this metric Coupling.

These three tools (Structure Chart, Cohesion, and Coupling) together help a programmer to more effectively modularize a program so it is easier to write the code, easier to fix bugs, and easier to understand.

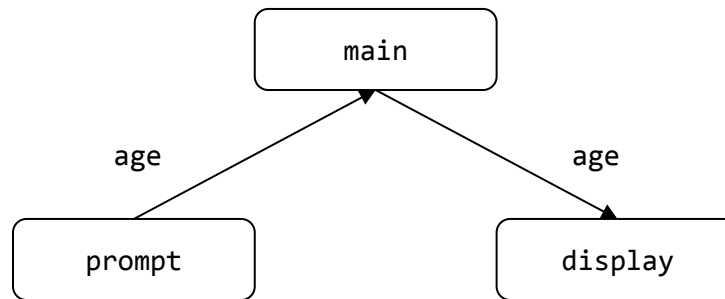
# Structure Chart

A Structure Chart is a tool enabling us to design with functions without getting bogged down in the details of what goes on inside the functions. This is basically a graph containing all the functions in the program with lines indicating how the functions get called.

For example, consider a program designed to prompt the user for his age and display a witty message:

```
What is your age: 29
You are 29 again? I was 29 for over a decade!
```

The Structure Chart would be:



There are three components to a Structure Chart: the function, the parameters, and how the functions call each other (program structure).

## Functions

Each function in the Structure Chart is represented with a round rectangle. You specify the function by name (remember to camelCase the name as we do with all variable and function names). Since functions are typically verbs, there is typically a verb in the name. In the above example, there are three functions: `main`, `prompt`, and `display`.

## Parameters

The second part of a Structure Chart is how information flows between functions. This occurs through parameters as well as through the return mechanism. If a function takes two parameters, then one would expect an arrow to flow into the function with two variables listed. In the above example, the function `prompt` accepts no data from `main` though it sends out a single piece of data: the `age`. Thus the following prototype for `prompt`:

```
int prompt();
```

The next function is `display`. It sends no data back to `main` but accepts a single piece of data, the `age`. Thus one would expect the following prototype for `display`:

```
void display(int age);
```

## Program structure

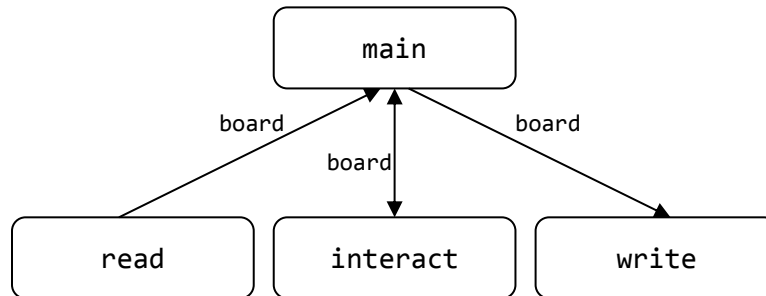
The final part of a Structure Chart is how the functions in a program call each other. Typically, we put `main` on top and, below `main`, all the functions that `main` calls. Note that you can have a single function that is called by more than one function. In this case, arrows will be reaching this function from multiple sources. Please see Project 1 for an example of a Structure Chart.

## Designing with Structure Charts

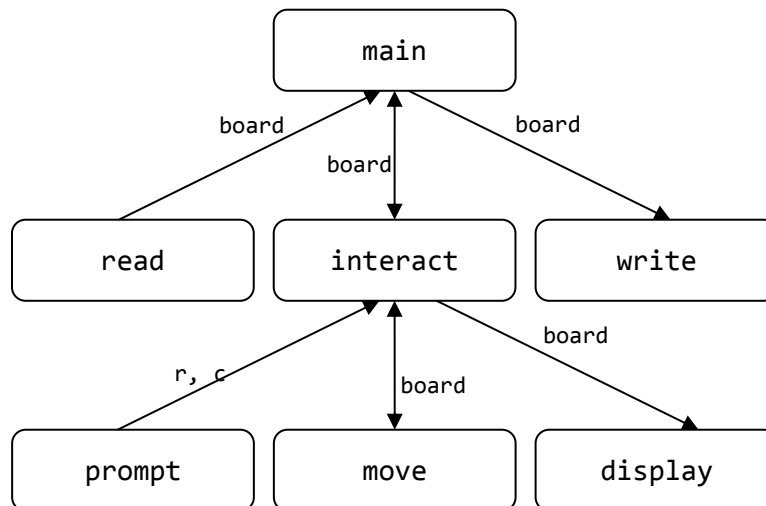
The Structure Chart is often the first step in the design process. Here we answer the big questions of the program:

- What will the program do?
- What are the big parts of the program?
- How will the various parts communicate with each other?

To accomplish this, we start with the top-down approach. We start with very general questions and slowly work to the details. Consider, for example, a program designed to play Tic-Tac-Toe. We would start with a very general design: the program will read a board from a file, allow the user to interact with the board, and then write the board back to the file.

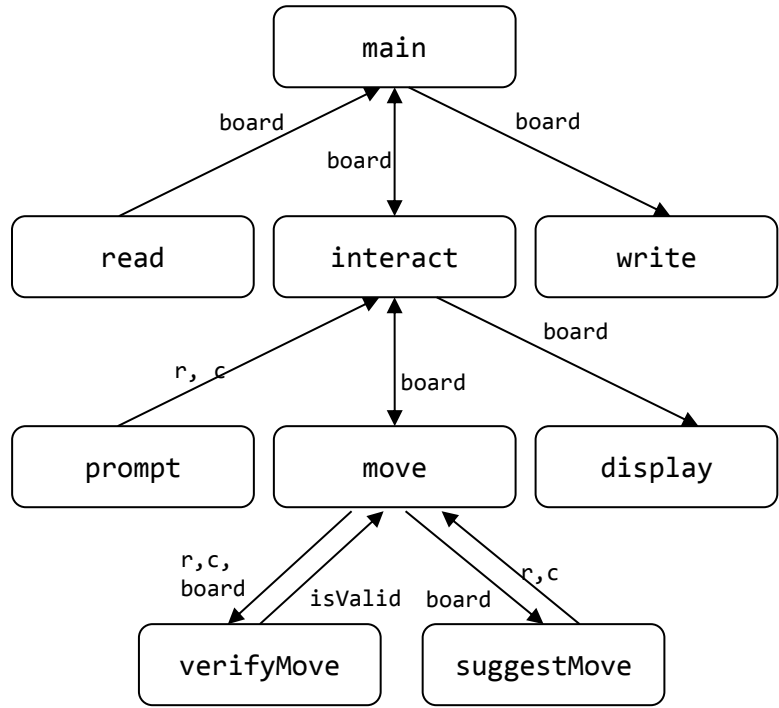


Note how each of the functions is Cohesive (does one thing and one thing only) and has simple Coupling (only one parameter is passed between the functions). That being said, the `interact()` function is probably doing too much work. We will delegate some of that work to three other functions:



Again, each of the functions (`prompt()`, `move()`, and `display()`) are Cohesive and have loose Coupling. However, it appears that the `move()` function is still too complex. While it is still Cohesive, we still might want to delegate some of the work to other functions.

The final Structure Chart for our Tic-Tac-Toe program is the following:

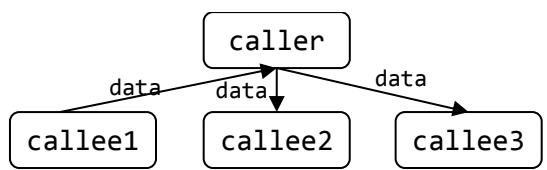


There are a few things to observe about this and all Structure Charts:

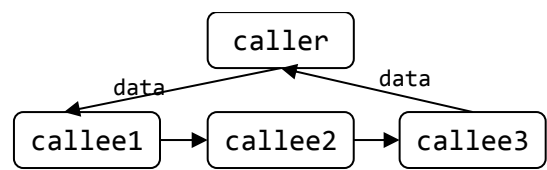
7. We always put `main()` on top. This means that control goes from the top down, rather than following the flow of the arrows. A common mistake new programmers make is to put `main()` in the center of the Structure Chart with arrows extending in all directions. We call this “spider” Structure Charts.
8. There are seldom more than three functions called from a single function. If too many arrows emanate from a given function, then that function is probably doing too much. There are exceptions from this rule-of-thumb of course. One example is when all the child functions do the same type of thing. The Structure Chart from Project 1 is an example. When in doubt, ask yourself if each function is functionally cohesive.
9. A Structure Chart is a tree; there are no circuits. If a function (`interact()` in the above example) calls another function (`prompt()`), control returns to the caller when the callee is finished.

```

void caller()           // caller will call three functions
{
    int data = callee1(); // first callee1 is called with the return value in data
    callee2(data);        // next data is sent to callee2.
    callee3(data);        // finally data is sent to callee3
}
  
```



Structure Chart of one function calling three in sequence



ERROR: there are no circuits in a Structure Chart! When `callee1()` is finished, data flows back to `caller()`. It cannot flow to `callee2()`

## Example 2.0 – Count Factors

### Demo

This example will demonstrate how to break a large and complex program into a set of smaller and more manageable functions. When considering how this will be done, the principles of Cohesion and Coupling will be considered. The resulting solution will be presented both as a structure chart and as a set of function prototypes.

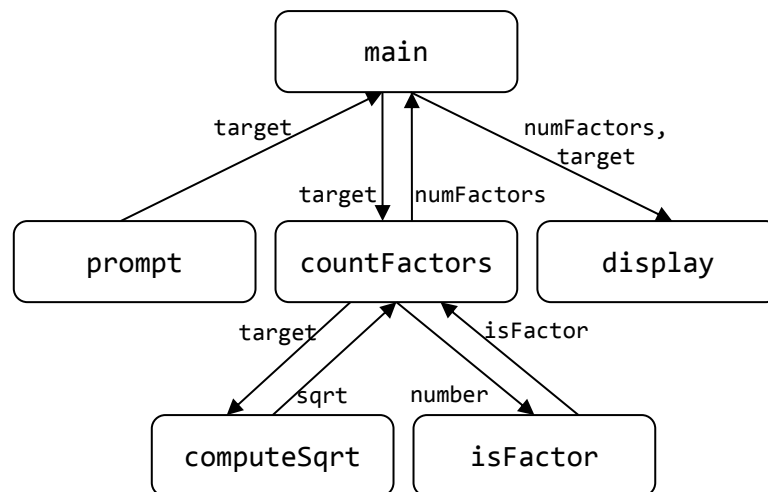
### Problem

Write a program to find how many factors a given number has. To accomplish this, it is necessary to enumerate all the values between 1 and the square root of the target number. Each of these values will then need to be checked to see if it evenly divides into the target number.

What number would you like to find the factors for? 16  
There are 4 factors in 16.

### Solution

The first part of the solution is to identify the functions and how they will call each other. The following structure chart represents one possible solution.



The second part is to convert this map into a set of function prototypes.

```

int main(); // calls prompt, countFactors, & display
int prompt(); // called by main
int countFactors(int target); // called by main, calls computeSqrt, isFactor
void display(int target, int num); // called by main
int computeSqrt(int target); // called by countFactors
bool isFactor(int x, int y); // called by isFactor
  
```

Observe how the structure chart tells you the function names, what parameters go into the functions, and who calls a given function. This allows us to create an outline for the program.

### Challenge

As a challenge, try to fill in the functions for this program. The `computeSqrt()` function may look something like this:

```

#include <cmath> // for the SQRT function

int computeSqrt(int target)
{
    return (int)sqrt((double)target);
}
  
```

# Cohesion

Recall that Cohesion is a metric by which we measure the “strength” of a function. A more formal definition of Cohesion is:

*Cohesion is a measurement of how well a function performs one task.*

This definition has two components:

- “a measurement:” Cohesion is a metric, reporting on the quality of one aspect of the design.
- “performs one task:” Cohesion is a property of a single function.

A well-designed function will be completely focused on a single task. There are four different levels of Cohesion (presented from best to worst): Strong, Extraneous, Partial, and Weak.

## Strong Cohesion

The strongest and most desirable level of Cohesion is where all the code in a function is directed to one purpose. The formal definition of Strong Cohesion is:

*All aspects of a function are directed to perform a single task, and the task is completely represented.*

There are two parts to this definition. The first is that the unit of software does nothing extra. Any extra code thrown into a function will forfeit its classification as Strong Cohesion. The second part of the definition is that the task or concept is completely represented. Anything that leaves part of the task undone or relies on the client to complete the work cannot be considered Strong. Of course, every function should strive for Strong cohesion. Observe that it does not matter how simple or complex the task is; it is Strongly Cohesive as long as only that task is being performed.

Consider the following function:

```
/* *****  
 * COMPUTE PAY  
 * Determine an employee's pay based on hourly wage and number of hours worked  
 * ***** */  
float computePay(float hours, float wage)  
{  
    // regular pay  
    if (hours < 40.0)  
        return hours * wage;  
  
    // overtime  
    else  
        return (wage * 40.0) +  
               (wage * 1.5 * (hours - 40.0));  
}
```

Observe how `computePay()` does one thing and one thing only: it computes pay given an employee's hourly wage and number of hours worked. This task is completely accomplished; no other work is needed in order to finish this task.

## Extraneous Cohesion

The first weak form of Cohesion is Extraneous. Here exists something unnecessary in the unit of software. A political analogy would be a “rider” on a bill and a sports analogy would be a “bench warmer” on a team. The formal definition of Extraneous Cohesion is:

*At least one part of a function is not directed towards a single task.  
However, the principle task is completely represented.*

Streamlined software should have no extraneous functionality. An example of an Extraneous compute-tax function would be one that correctly performs the calculation and then asks the user what to do with the refund. Any time the word “and” is used to completely describe a unit of software, it is probably Extraneous Cohesion.

Consider the following function:

```

/*****
 * COMPUTE PAY
 * Determine an employee's pay based on hourly wage and number of hours worked.
 * This function also displays a warning message if too many hours are worked.
 *****/
float computePay(float hours, float wage)
{
    // display error message if more than the maximum amount of work was done
    if (hours > 60.0)
        cout << "WARNING: Special permission is required to work more than 60 hours.\n";

    // regular pay
    if (hours < 40.0)
        return hours * wage;

    // overtime
    else
        return (wage * 40.0) +
               (wage * 1.5 * (hours - 40.0));
}

```

This function completely accomplishes the task of computing the pay for a worker. Unfortunately, it also does something that is not directly related to the task at hand. In this case, it warns if too much work is reported. As a general rule, a function designed to “compute” should not also “display.”

Any time a function has code that is not directly related to the task at hand, there is a good chance that the function is Extraneous Cohesion (or worse!). Fortunately, the fix is very easy: move the extra code to a more appropriate location.



## Partial Cohesion

Another weak form of cohesion is Partial, where a task is left incomplete. In other words, additional data or work needs to be stored or completed elsewhere for the concept or task to be completed. Note that Partial is not below Extraneous in the hierarchy but rather a peer. One does not improve a Partial class by making it Extraneous. Instead, one finished the job it was designed to do. The formal definition of Partial Cohesion is:

*All aspects of a function are directed to perform a single task, but the task is not completely represented by the function.*

Partial Cohesion is particularly difficult for the client of a system because the onus is on the client to figure out how to finish the job. It is also difficult for the author of the software because, in order to thoroughly test the system, all possible implementations that complete the task needs to be discovered. Any time a description of a system necessitates a detailed description of the context in which it is used, that system is a candidate for Partial cohesion.

Consider the following functions:

```

/*****
 * COMPUTE OVERTIME PAY
 * Determine an employee's pay based on hourly wage and number of hours worked.
 * WARNING: Call computeNormalPay() if hours is less than 40
 *****/
float computeOvertimePay(float hours, float wage)
{
    return (wage * 40.0) +
           (wage * 1.5 * (hours - 40.0));
}

/*****
 * COMPUTE NORMAL PAY
 * Determine an employee's pay based on hourly wage and number of hours worked.
 * WARNING: Call computeOvertimePay() if hours is less than 40
 *****/
float computeNormalPay(float hours, float wage)
{
    return hours * wage;
}

```

Here each of the two functions accomplishes part of the task at hand. Anyone using one of these functions will also have to use the second to get the job done.

There are two ways to fix this problem and make the function(s) Strongly Cohesive: either combine the functionality into a single function or create a wrapper function that calls both of the components:

```

/*****
 * COMPUTE PAY
 * Determine an employee's pay based on hourly wage and number of hours worked
 *****/
float computePay(float hours, float wage)
{
    if (hours < 40.0)
        return computeNormalPay(hours, wage);
    else
        return computeOvertimePay(hours, wage);
}

```

This function is now Strongly Cohesive. It may be desirable to pursue a design like this when there is another part of the program that needs computeNormalPay() or computeOvertimePay() without the other part.

## Weak Cohesion

The worst form of cohesion is Weak. One should never design for Weak Cohesion; it is a state that is to be generally avoided. The formal definition of Weak Cohesion is:

*At least one part of a function is not directed towards performing a single task.  
Additionally, the task is not completely represented by the function.*

In other words, weak cohesion is a combination of Extraneous and Partial. In theory, one should never come across Weak cohesion. Alas, if only this were true.

Consider the following function:

```
/* *****  
 * COMPUTE PAY  
 * Determine an employee's pay based on hourly wage and number of hours worked.  
 * This function also configures the display for money output.  
 * WARNING: Call computeNormalPay() if hours is less than 40  
 * ***** */  
float computePay(float hours, float wage)  
{  
    // set up the display for money  
    cout.setf(ios::fixed);  
    cout.setf(ios::showpoint);  
    cout.precision(2);  
  
    // regular pay  
    if (hours < 40.0)  
        cout << "ERROR: This only works for hours greater than 40\n";  
  
    // compute overtime pay  
    return (wage * 40.0) +  
           (wage * 1.5 * (hours - 40.0));  
}
```

This function exhibits Extraneous Cohesion. We can tell first because the function comment block contains the word “also.” A subsequent inspection of the code will reveal the code to configure output for money. Since this function does not display anything, the code clearly does not belong here.

This function exhibits Partial Cohesion because it only produces correct output if the employee’s wage is not less than 40 hours. In this case, the program will display an error message on the screen and still produce erroneous output.

Since this function is both Extraneous and Partial, it can be classified as Weakly Cohesive. It appears that the programmer threw code together hoping it would work, rather than properly designed the function. On the surface, it might seem that the best approach from this point is to add the missing functionality and remove the extraneous parts. In practice, a better approach is to start the design process from scratch with Cohesion in mind.

# Coupling

Recall that Coupling is a metric of the complexity of the information interchange between two functions. A more formal definition of Coupling is:

*Coupling is a measurement of the complexity of the interface between functions.*

This definition has two components:

- “a measurement:” Coupling is a metric, reporting on the quality of one aspect of the design.
- “the complexity of the interfaces:” This can be summed up with the question: “how much does the programmer need to know to successfully use and how much does the resulting software need to do?”
- “between:” Coupling fundamentally is a measure of how different parts of a system communicate. It is not a property of an individual function, but rather how it interacts with the rest of the system.

A well-designed interface between functions will be easy to understand and use. There are seven different levels of Coupling (presented from best to worst): Trivial, Encapsulated, Simple, Complex, Document, Interactive, and Superfluous.

## Trivial Coupling

Trivial is the weakest or best form of Coupling. Here the client of a unit of software needs to provide no information and receives no information from another unit of software. The formal definition of Trivial Coupling is:

*There is no information interchange between functions.*

In other words, one unit may instantiate, call, or activate another, but no information is passed. Similarly, no information can be gleaned from the timing of the function call. An example would be a function with no return value and no parameters.

Consider the following function:

```
/* *****  
 * DISPLAY INSTRUCTIONS  
 * Inform the user about the functionality of this program  
 * ***** */  
void displayInstructions()  
{  
    // Inform the user about what this program will do  
    cout << "This program will display the status of your monthly budget.\n"  
        << "The produced report will include both your projected expenditures, "  
        << "as well as what you actually spent last month.\n\n";  
  
    // Inform the user about the type of data that will be requested  
    cout << "To accomplish this, it is necessary to prompt you for several "  
        << "confidential financial details.\n"  
        << "Do not worry, no confidential data will be saved in the process.\n";  
}
```

Note that a function may have no input parameters and have no return type yet still not be Trivially Coupled. Sometimes a hidden global variable is at play, making the function something other than Trivially Coupled.

```
void displayPay()  
{  
    cout << '$' << pay << endl;    // NOT trivially coupled!  
}
```

## Encapsulated Coupling

Encapsulated is a very weak form of Coupling defined by all of the parameters being in a trusted and accessible state. In other words, there are logical checks in place to ensure that no invalid data is sent between modules. This level makes no reference to the degree of complexity of the data nor the number of data items passed between modules.

The formal definition of Encapsulated Coupling is:

*All the information exchanged between functions  
is in a convenient form and is guaranteed to be in a valid state.*

Note that Encapsulation is an Object-Oriented programming topic and is therefore not a topic for CS 124. That being said, there is a single example of encapsulated data that we have learned about thus far: Boolean data. Since a Boolean variable can have only two states and both are, by definition, valid, it is impossible to pass an invalid Boolean parameter. Enumerations are another validated data-type present in most languages. Here the compiler ensures that the data is always in a valid state. These will be discussed in more detail in CS 165. With modern languages and modern programs, the most common way to achieve the validated status is to use a class whose methods contain checks to guarantee data validity.

Consider the following function:

```

/*****
 * GET IS MALE
 * Prompt the user for his/her gender and return if he/she is male
 *****/
bool getIsMale()
{
    char input;
    cout << "Please select your gender, 'm' for male and 'f' for female: ";
    cin >> input;

    return (input == 'm') || (input == 'M');
}

```

This function takes no input parameters and returns a single Boolean value. The data is in a convenient form and is guaranteed to be valid. Note that Coupling makes no reference to information interchanges between the user and the program; it only concerns itself with information interchange between parts of the program.

Consider this function:

```

/*****
 * DISPLAY CHILD TAX CREDIT STATUS
 * Display to the user the status of their Child Tax Credit
 *****/
void displayChildTaxCreditStatus(bool isEligible)
{
    if (isEligible)
        cout << "You are eligible for the child tax credit.";
    else
        cout << "You cannot claim the child tax credit this year.";
}

```

This is also Encapsulated Coupling because a single Boolean value is passed into the function. From this we can see that though a function may have Trivial input parameters, it is an Encapsulated Coupling if it has a single Encapsulated return value (and vice versa).

## Simple Coupling

Another weak form of coupling is Simple, meaning that data can be selected, interpreted, and validated easily. Parameters consisting of simple built-in data-types such as integers or characters are often Simple, assuming that the use of the parameter is easily specified. The formal definition of Simple Coupling is:

*All the information exchanged between functions is easy to select, interpret, and validate.*

Perhaps this can be best captured with a few questions:

- Can I explain the purpose of this parameter with a few words?
- Is it intuitively obvious what this parameter represents and how it is used?
- Is it easy to validate the parameter with a simple IF statement?

Consider the following function:

```

/*****
 * PROMPT FOR INCOME
 * Prompt the user for his/her monthly/yearly income.
 *****/
double promptForIncome(bool isMonthly)
{
    // different prompt according to the value of isMonthly
    if (isMonthly)
        cout << "Please specify your monthly income: ";
    else
        cout << "Enter your yearly income: ";

    // get the income value
    double income;
    cin >> income;
    return income;
}

```

In this case, the input parameter is a Boolean value. It may seem like this is an example of Encapsulated Coupling. Note, however, that the return value is a `double`. Clearly there are some values which are inappropriate (negative values or numbers which are smaller than a cent), meaning some trivial validation may be required. However, the return value is easy to select, interpret, and validate. This function is thus Simple Coupling.

Consider the following function:

```

/*****
 * ELIGIBLE FOR CHILD TAX CREDIT
 * Return true if eligible for a child tax credit for a given child
 *****/
bool eligibleForChildTaxCredit(double income, int ageChild)
{
    // only eligible if the child is under 17 and income between $2,500 and $200,000
    return (ageChild < 17) && (income >= 2500.00) && (income <= 200000.00);
}

```

Note that the return type makes the function a candidate for Encapsulation Coupling. However, the `income` parameter as well as the `ageChild` parameter meet all the criteria for Simple Coupling. Since the lowest Coupling classification of any parameter (input or output) determines the overall Coupling of the function, this can be classified as Simple Coupling.

## Complex Coupling

A tight or bad degree of coupling is Complex. Here the function callers need to know a great deal about the parameters in order to make a connection. The greater knowledge the callers need to have and the more work the callers need to accomplish to make sure the connection is done correctly, the more complex the level of coupling. While it might be worthwhile to enumerate sub-levels of Complex Coupling, it is sufficient to say that all are bad and should be avoided. The formal definition of Complex Coupling is:

*At least one piece of information is non-trivial to create, validate, or interpret.*

There are many things about a parameter which could cause it to be classified as Complex. A few examples:

- Input that is interpreted as a command, requiring the callee to act in response to the command. Understanding all possible commands is non-trivial.
- Two variables that must be in sync with each other for them to make sense, such as a list of names and a list of addresses. Here the 4<sup>th</sup> name on the first list corresponds to the 4<sup>th</sup> address on the second list. Creating and validating these two lists is non-trivial.
- Text that must be in a given format. Passing text consisting of nothing but spaces may constitute an invalid employee last name.

Consider the following function:

```
/* *****  
 * HANDLE ACTION  
 * Execute one of a collection of actions depending on the "command" parameter  
 * ***** */  
void handleAction(int command)  
{  
    if (command == 1)  
        displayInstructions();  
    else if (command == 2)  
        openFile();  
    else if (command == 3)  
        playGame();  
    else if (command == 4)  
        exitProgram();  
    else  
        cout << "Invalid or unknown command: " << command << endl;  
}
```

Notice how the parameter is a single integer. This may seem to be a candidate for Simple Coupling. However, the integer is not used in trivial way. Both the caller and the callee share a complex command language where actions are represented with numbers. If the language is expanded or reduced, both the caller and the callee will need to change their code to accommodate the change. This makes the two functions tightly Coupled.

Consider the following function:

```
/* *****  
 * DISPLAY FULL NAME  
 * Display the user's full name in the format: last name, first name.  
 * ***** */  
void displayFullName(char nameLast[256], char nameFirst[256])  
{  
    cout << nameLast << ", " << nameFirst;  
}
```

This seemingly simple function is actually incomplete. What if the last name string was empty? What if it contained a newline character or a comma? Dealing with all the formatting needs of a last name (no spaces, only limited punctuation, no newline characters, no numbers) is non-trivial to validate. That makes this function's Coupling level Complex.

## Document Coupling

Another tight level of Cohesion is Document. Here data conforming to some language is passed from a producer to a consumer. Note that the producer could be the caller or the callee, depending on the direction of information flow. The important thing to note is that both the producer and the consumer need to fully understand the intricacies of the shared language. The formal definition of Document Coupling is:

*At least one piece of information contains a rich language including syntactic and/or semantic rules.*

On the surface, this may seem rather difficult to understand and perhaps not an important case worth considering. In practice, however, it is more common than you may think.

In order to demonstrate Document Coupling, it is necessary to use C++ language constructs that will not be introduced for several chapters. Just pay attention to the commands and try to get the jist of the algorithm.

```

/*****
 * EXECUTE ASSEMBLY COMMAND
 * Take action according to an assembly opcode such as "ADD 4"
 *****/
void executeAssemblyCommand(string opcode, int & register)
{
    // get "ADD" from "ADD 4"
    string command = opcode.substr(0, 3);

    // get "4" from "ADD 4"
    string parameter = opcode.substr(4);

    // execute the command.
    if (command == "ADD")
        register += integerFromString(parameter);
    if (command == "SUB")
        register -= integerFromString(parameter);

    ... and so on ...
}

```

The above function needs to understand the complete assembly language and syntax in order to properly handle the input in the opcode parameter. This language certainly has “rich syntatic and sematic rules.”

Another example, this time taken from the Unit 3 project. Consider the game MadLib® where a story has placeholders in which the player of the game will insert his or her own answers to certain questions. The game starts with a raw story similar to the following:

I have a very :adjective pet :animal :.

Here the user will be prompted for an adjective and an animal:

Adjective: silly  
Animal: great white shark

The result of this game will be a completed story with the user’s provided prompts:

I have a very silly pet great white shark.

A function interpreting the raw story and generating a completed story will need to exhibit Document Coupling because it will need to understand how to interpret all the prompts.

## Interactive Coupling

Interactive Coupling is closely related to Document with an additional component: the information exchange between components is ongoing rather than a single one-time event. Interactive Coupling is usually characterized by a session where conversations begin, the participants react to each other, and conversations are terminated. There are usually syntactic and semantic rules that need to be followed to correctly maintain the conversation. Another way to look at Interactive coupling is a sequence of Document interactions involving maintenance of state by both parties. The formal definition of Interactive Coupling is:

*There exists a communication avenue between units of software involving non-trivial dialogs, sessions, or interactions.*

It is difficult to show an example of Interactive Coupling using the programming tools presented in CS 124. The main problem is that the two functions exhibiting this interaction need to maintain state. In other words, they both need to keep track of the status of the conversation. To demonstrate this, consider a function called `send()` which sends messages to another function or entity in the system.

```

/*****
 * SEND EMAIL
 * Send an e-mail message using the SMTP protocol
 * Here sendEmail() is demonstrating Simple Coupling whereas send() is Interactive
 *****/
void sendEmail(char data[256])
{
    // initiate the conversation
    char *response = send("HELO");

    // from and to:
    response = send("MAIL FROM:<sender@sourcedomain.com>");
    response = send("RCPT TO:<recipient@destinationdomain.com>");

    // send body of the message
    response = send(data);

    // indicate we are done
    response = send("\n.\n");
    response = send("QUIT");
}

```

The SMTP e-mail protocol involves several many interactions between the client and the server. Notice that all of these interactions occur through the same `send()` function. Here the server responds to messages differently depending on the state of the message. Thus the `send()` function is exhibiting Interactive Coupling.



## Superfluous Coupling

The tightest and therefore worst level of coupling is Superfluous. The formal definition of Interactive Coupling is:

*At least one piece of data or information is passed between functions unnecessarily.*

This is the only level of coupling that makes a reference to the amount of data passed between functions. Data passing between functions is only bad if that data is not necessary. There are two important parts to this definition: what is unnecessary and what is data/information passing. The unnecessary component is completely domain specific. For example, if a function has read/write access to an asset when read-only is required, then the write aspect is unnecessary and the coupling can be classified as superfluous. For example, consider the following function:

```

/*****
 * PROMPT GRADE
 * Prompt the user for his/her GPA
 *****/
float promptGrade(float grade)
{
    cout << "What is your GPA. Please enter a value between 0.0 and 4.0: ";
    cin >> grade;
    return grade;
}

```

On the surface, this appears to be Simple Coupling: the grade parameter needs to be validated before it is used (what if the user selected -1.0 as the GPA?), but that validation can be easily accomplished. However, there is a problem. Data needs to leave this function, but it does not need to enter the function. For some reason, there is an input parameter called grade. This function would be Simple Coupling if grade was a local variable. However, because it is an unnecessary input parameter, this is Superfluous Coupling.

The “passing” component of the definition is a bit more difficult to explain. Consider the range of scope for a language like C++ language (from small to large): block, local variable, one-way parameter (by-value), two-way parameter (by-reference), and global variables. There are legitimate uses for each of these scope levels in many applications. However, if a function utilizes a scope larger than is necessary, then superfluous coupling exists. The degree of superfluosity depends on the number of scope levels beyond that which is necessary that was utilized in a given application. Consider the following function:

```

/*****
 * DISPLAY USER HEIGHT
 * Display the height of the user in convenient units
 *****/
void displayUserHeight(float heightFeet)
{
    // display the height in imperial units
    if (useImperialUnits)
        cout << heightFeet << " feet";

    // display the height in the metric system
    else
        cout << (heightFeet * 0.3048) << " meters";
}

```

This function may appear to be Simple Coupling because there are no output parameters and the single input parameter is easy to verify. However, there is another variable referenced (useImperialUnits) not declared in this function. This function is therefore a global variable, a scope much larger than necessary. The reference to this variable makes the entire function Superfluous Coupling.

# Problems

## Problem 1

Create a list of the levels of Cohesion from best on the top to worst on the bottom.

- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_

*Please see page 117 for a hint.*

## Problem 2

Classify the form of Cohesion from the following example of code:

```
void displayGPA(float gpa)
{
    // configure the output so the GPA appears correctly
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);

    // display the GPA
    cout << gpa;
}
```

Answer:

\_\_\_\_\_

*Please see page 113 for a hint.*

## Problem 3

Classify the form of Cohesion from the following example of code:

```
float getGPA()
{
    // get the student's GPA
    float gpa;
    cin >> gpa;

    // prompt for the number of credits
    cout << "Please enter the number of credits you are taking this semester: ";
    return gpa;
}
```

Answer:

\_\_\_\_\_

*Please see page 113 for a hint.*

**Problem 4**

Create a list of the levels of Coupling from best on the top to worst on the bottom.

- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_

*Please see page 117 for a hint.*

**Problem 5**

Classify the form of Coupling from the following example of code:

```
void displayGreeting()
{
    // prompt for name
    char name[256];
    cout << "What is your name? ";
    cin >> name;

    // display the greeting
    cout << "Hello, " << name << " how are you today?\n";
}
```

Answer:

\_\_\_\_\_

*Please see page 113 for a hint.*

**Problem 6**

Classify the form of Coupling from the following example of code:

```
void setGender()
{
    char input;
    cout << "Are you male? ";
    cin >> input;

    isFemale = (input == 'n') || (input == 'N');
}
```

Answer:

\_\_\_\_\_

*Please see page 113 for a hint.*

### Problem 7

Match the Cohesion name with the definition:

|            |   |
|------------|---|
| Strong     | At least one part of the function is unnecessary to the main task at hand       |
| Extraneous | Components of the function are irrelevant and parts of the function are missing |
| Partial    | Does one thing completely and one thing only                                    |
| Weak       | The main task of the function is not completely done                            |

*Please see page 113 for a hint.*

### Problem 8

Match the Coupling name with the definition:

|              |  |
|--------------|--|
| Trivial      | A dialog exists between functions that involves multiple interactions                |
| Encapsulated | Information is passed between functions that involves a complex language             |
| Simple       | At least one parameter is non-trivial to create, validate, or interpret              |
| Complex      | All the parameters are in a convenient form and is guaranteed to be in a valid state |
| Document     | At least one parameter is passed between functions unnecessarily                     |
| Interactive  | There is no information interchange between functions                                |
| Superfluous  | All the parameters are easy to select, interpret, and validate                       |

*Please see page 113 for a hint.*

### Problem 9

Classify the form of Cohesion from the following example of code:

```
float getGPA()
{
    // set up the display for GPA
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);

    // prompt for GPA
    float gpa;
    cout << "Enter your GPA: ";
    cin >> gpa;
    return gpa;
}
```

Answer:

*Please see page 113 for a hint.*

### Problem 10

Classify the form of Cohesion from the following example of code:

```
double getIncome()
{
    cout << "Enter your income: ";
    return 0.0;
}
```

Answer:

*Please see page 113 for a hint.*

### Problem 11

Identify the type of Coupling that the following function exhibits:

```
float absoluteValue(float value)
{
    if (value >= 0)
        return value;
    else
        return -value;
}
```

Answer:

---

*Please see page 117 for a hint.*

### Problem 12

Identify the type of Coupling that the following function exhibits:

```
char adjustedGrade(char grade, bool cheated)
{
    if (cheated)
        return 'F';
    else
        return grade;
}
```

Answer:

---

*Please see page 117 for a hint.*

### Problem 13

Draw a Structure Chart to convert Fahrenheit to Celsius. The program has three functions besides main():

```
float getTemperature();
float convert(float fahrenheit);
void display(float celsius);
```

Answer:

*Please see page 112 for a hint.*

### Problem 14

Draw a Structure Chart to represent the following problem: prompt the user for his hourly wage and number of hours worked. From this, compute his weekly pay (taking time-and-a-half overtime into account). Deduct from his pay his tax and tithing. Finally, display to the user how much money he has left to spend:

*Please see page 112 for a hint.*

### Problem 15

Sue wants to write a program to help her determine how much money she is spending on her car. Specifically, she wants to know how much she spends per day having the car sit in her driveway and how much she spends per mile driving it. This program will take into account periodic costs such as devaluation, insurance, and parking. It will also take into account usage costs such as gas, repair costs, and tires. Draw a Structure Chart to represent this program.

*Please see page 112 for a hint.*

## Assignment 2.0

Your assignment is to create three Structure Charts representing how you would solve three programming problems. In these examples, you are not concerned with how you would implement each individual function. Instead, you want to identify what the function will do (Cohesion), how information will pass between them (Coupling), and how the functions call each other.

### Problem 1: Compute grade

The first problem is to create a Structure Chart for a program to convert a number grade (ex: 88%) into a letter grade (ex: B+). Consider the following example (input is Underlined):

```
What is your grade in percent: 88
Your grade is B+
```

### Problem 2: Compute tithing

The second problem is designed to help a child set aside part of his allowance for tithing. This program will prompt the user for his allowance, figure out how much is left after tithing is taken out, and display the results.

```
What is your allowance? $10.50
You get to spend: $9.45
```

### Problem 3: Currency

The final problem is designed to help an international traveler convert his money to various currencies. After prompting him for the amount to be converted, the program will display how many British pounds, Euros, or Japanese Yen he will have:

```
How much money do you want to convert? $100.00
British Pounds: £61.50
Euros: €70.09
Japanese Yen: ¥8079.06
```

Please bring these Structure Charts into class on a sheet of paper (face-to-face students) or take a picture and submit it electronically (online students). Don't forget to put your name on your assignment!

*Please see page 112 for a hint.*

## 2.1 Debugging

Sam has just spent an hour and a half in the lab tracking down a bug that turned out to be a small typo. What a waste of time! There are so many better things he could have been doing with that time (such as trying to get a date with that cute girl in his computer class named Sue). If only there was some way to get his program to tell him where the problems were, then this whole process would be much simpler!

### Objectives

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

- Create asserts to catch many of the most common programmer problems.
- Use `#define` to move constants to the top of a program.
- Use `#ifdef` to create debug code in order to test a function.
- Write a driver program to verify the correctness of a function.
- Create stub functions to make an outline of a large program.

### Prerequisites

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

- Create a function in C++ (Chapter 1.4).
- Convert a logic problem into a Boolean expression (Chapter 1.5).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).
- Measure the cohesion level of a function (Chapter 2.0).
- Measure the degree of coupling between functions (Chapter 2.0).
- Create a map of a program using structure charts (Chapter 2.0).

## Asserts

When writing a program, we often make a ton of assumptions. We assume that a function was able to perform its task correctly; we assume the parameters in a function are set up correctly; and we assume our data structures are correctly configured. A diligent programmer would check all these assumptions to make sure his code is robust. Unfortunately, the vast majority of these checks are redundant and, to make matters worse, can be a drain on performance. A method is needed to allow a programmer to state all his assumptions, get notified when the assumptions are violated, and have the checks not influence the speed or stability of the customer's program. Assertions are designed to fill this need.

An assert is a check placed in a program representing an assumption the developer thinks is always true. In other words, the developer does not believe the assumption will ever be proven false and, if it does, definitely wants to be notified. An assert is expressed as a Boolean expression where the true evaluation indicates the assumption proved correct and the false evaluation indicates violation of the assumption. Asserts are evaluated at run-time verifying the integrity of assumptions with each execution of the check.

An assert is said to fire when, during the execution of the program, the Boolean expression evaluates to `false`. In most cases, the firing of an assert triggers termination of the program. Typically the assert will tell the programmer where the assert is located (program name, file name, function, and line number) as well as the assumption that was violated.



Assertions have several purposes:

- **Identify logical errors.** While writing a program, assertions can be helpful for the developer to identify errors in the program due to invalid assumptions. Though many of these can be found through more thorough investigation of the algorithm, the use of assertions can be a time saver.
- **Find special-case bugs.** Testers can help find assumption violations while testing the product because their copy of the software has the asserts turned on. Typically, developers love this class of bugs because the assert will tell the developer where to start looking for the cause of the bug.
- **Highlight integration issues.** During component integration activities or when enhancements are being made, well-constructed assertions can help prevent problems and speed development time. This is the case because the asserts can inform the programmer of the assumptions the code makes regarding the input parameters

Assertions are not designed for:

- **User-initiated error handling.** The user should never see an assert fire. Asserts are designed to detect internal errors, not invalid input provided by the user.
- **File errors.** Like user-errors, a program must gracefully recover from file errors without asserts firing.

## Syntax

Asserts in C++ are in the `cassert` library. You can include asserts with:

```
#include <cassert>
```

Since asserts are simply C++ statements (more precisely, they are function calls), they can be put in just about any location in the code. The following is an example assert ensuring the value `income` is not negative:

```
assert(income >= 0);
```

If this assert were in a file called `budget.cpp` as part of a program called `a.out` in the function called `computeTithing`, then the following output would appear if the assumption proved to be invalid:

```
a.out: budget.cpp:164: float computeTithing(float income): Assertion `income >= 0'
failed.
Aborted
```

It is important that the client never sees a build of the product containing asserts. Fortunately, it is easy to remove all the asserts in a product by defining the `NDEBUG` macro. Since asserts are defined with pre-processor directives, the `NDEBUG` macro will effectively remove all assert code from the product at compilation time. This can be achieved with the following compiler switch:


```
g++ -DNDEBUG file.cpp
```



### Sue's Tips

The three most common places to put asserts are:

1. At the top of a function to verify that the passed parameters are valid.
2. Just after a function is called, to ensure that the called function worked as expected.
3. Whenever any assumption is made.

| Example 2.1 – Asserts |   |
|-----------------------|---|
| Demo                  | <p>This example will demonstrate how to add asserts to an existing program. This will include how to brainstorm of where bugs might exist, common checkpoints where asserts may reside, and how to interpret the messages that asserts give us.</p>   |
| Problem               | <p>Given a program to compute how much tithing an individual should pay given an income, add asserts to catch common bugs.</p> <div>           What is the income? <u>100</u><br/>           Tithe for \$100 is \$10         </div>   |
| Solution              | <p>The first part is to include the assert library:</p> <div>#include &lt;cassert&gt;</div> <p>Next, we will add asserts to the computeTithe() function.</p> <div> <pre>float computeTithing(float income) {     assert(income &gt;= 0.00);           // this only works for positive income      // compute the tithing     float tithe = income * 0.10;     assert(tithe &gt;= 0.00);           // The Lord doesn't owe us, right?     assert(income &gt; tithe);          // 10% should be less than 100%, right?      // return the answer     return tithe; }</pre> </div> <p>Observe how the asserts both validate the input (<code>income &gt;= 0</code>) and perform sanity checks that the resulting output is not invalid (<code>tithe &gt;= 0.0</code> as well as <code>income &gt; tithe</code>). The first assert is designed to make sure the function is called correctly. The second is to make sure the math was performed correctly..</p> |
| Challenge             | <p>As a challenge, add asserts to your Project 1 solution. Would any of these have caught bugs you ran into when you were writing your code?</p>  |
| See Also              | <p>The complete solution is available at <a href="#">2-1-asserts.cpp</a> or:</p> <div>/home/cs124/examples/2-1-asserts.cpp</div>   |

## #define

The `#define` mechanism is a way to get the compiler to do a search-replace through your file before the program is compiled. This is useful for values that never change (like  $\pi$ ). We will also use this to do more advanced things. An example of `#define` in action is:

| Before expansion   | After expansion   |
|--|---|
| <pre>include &lt;iostream&gt; using namespace std;  #define PI 3.14159  /*****  * MAIN  * Simple program to  * demonstrate #define  *****/ int main() {     cout &lt;&lt; "The value of pi is "           &lt;&lt; PI           &lt;&lt; endl;     return 0; }</pre> | <pre>include &lt;iostream&gt; using namespace std;  /*****  * MAIN  * Simple program to  * demonstrate #define  *****/ int main() {     cout &lt;&lt; "The value of pi is "           &lt;&lt; 3.14159           &lt;&lt; endl;     return 0; }</pre> |

Note that `#defines` are always ALL\_CAPS according to our style guidelines.

Observe how the value in the `#define` can be used much like a variable. In many ways, it is like a variable with one important exception: it can't vary. In other words, the value represented by the `#define` is guaranteed to not change during the course of the program.

### Sam's Corner

There is another way to make a variable that does not change: a constant variable:

```
const float PI = 3.14159;
```

Observe how the syntax is similar to any other variable declaration, including using the assignment operator with the semicolon. It is important to realize that this is not a global variable: the `const` modifier guarantees that the value in the variable does not change. Global variables are only dangerous because they can change in an unpredictable way



A few common uses of `#defines` are:

- **Constants:** Values that never change. Through the use of `#defines`, it is much easier to verify that all instances of the constant are the same in the program. We don't want to have more than one version of  $\pi$  for example.
- **Static file names:** Some file names, such as configuration files, are always in the same location. By using a `#define` for the file name, it is easy to see which files the program accesses and to ensure that all the parts of the code refer to the same file.

We can also put parameters in `#define` macros. Here, the syntax is similar to that of a function:

```
#define NEGATIVE(x) (-x)           // also ALL_CAPS
```

Again, this will expand just before compilation just like the non-parameter `#define` does. Consider the following code:

| Before expansion  | After expansion  |
|---|--|
| <pre>#include &lt;iostream&gt; using namespace std;  #define ADD_TEN(x) (x + 10)  /*****  * MAIN  * #define expansion demo  *****/ int main() {     int value = 5;     cout &lt;&lt; value           &lt;&lt; " + 10 = "           &lt;&lt; ADD_TEN(value)           &lt;&lt; endl;     return 0; }</pre> | <pre>#include &lt;iostream&gt; using namespace std;  /*****  * MAIN  * #define expansion demo  *****/ int main() {     int value = 5;     cout &lt;&lt; value           &lt;&lt; " + 10 = "           &lt;&lt; (value + 10)           &lt;&lt; endl;     return 0; }</pre> |

For more information about the `#define` pre-processor directive, please see:

[#define](#)

## #ifdef

Another pre-processor directive (along with `#define` and `#include`) is the `#ifdef`. The `#ifdef` preprocessor directive tells the compiler to optionally compile some code depending on the state of a condition. This makes it possible to have some code appear only in a Debug version of the program. Consider the following code:

```

/*****
 * COMPUTE TAX
 * Compute the monthly tax
 *****/
float computeTax(float incomeMonthly)
{
    float incomeYearly = incomeMonthly * 12.0;

#ifdef DEBUG
    cout << "incomeYearly == "          // the code between #ifdef and #endif
          << incomeYearly << endl;      // only gets compiled if the
                                          // DEBUG macro is defined
#endif // DEBUG

    float taxYearly;

    // tax code
    ...

#ifdef DEBUG
    cout << "taxYearly == "              // observe how we format the output so we
          << taxYearly << endl;          // can tell which variable we are
                                          // looking at in the output stream
#endif // DEBUG

    return taxYearly / 12.0;
}

```

In this example, we have debug code displaying the values of key variables. Note that we don't always want this code to execute; test bed will certainly complain about the unexpected output. Instead, we only want the

code to run when we are trying to fix a problem. The `#ifdef` mechanism allows this to occur. We can “turn on” the debug code with:

```
#define DEBUG
```

If this appears before the `#ifdefs`, then all the code will be included in the compilation and the `couts` will work as one expects. This allows us to have two versions in a single code file: the ship version containing code only for the customer to see, and the debug version containing tons of extra code to validate everything.



### Sam's Corner

An `#define` can also be turned on at compilation time without ever touching the source code. We do this by telling the compiler we want the macro defined:

```
g++ -D<MacroName>
```

For example, if you want to turn on the `DEBUG` macro without using `#define DEBUG`, this can be accomplished with:

```
g++ -DDEBUG file.cpp
```

As an exercise, please take a close look at (`/home/cs124/examples/2-1-debugOutput.cpp`). Please see the following link for more detail on how `#ifdef` works.

[#ifdef](#)



### Sam's Corner

The aforementioned `#ifdef` technique to display debug code can be tedious to write. Fortunately there is a more convenient way to do this. First, start with the following macro at the top of your program:

```
#ifdef DEBUG
#define Debug(x)  x
#else
#define Debug(x)
#endif
```

This macro actually does something quite clever. If `DEBUG` is defined in your program, then anything inside the `debug()` statement is executed. If `DEBUG` is not defined, then nothing is executed. Consider the following code:

```
void function(int input)
{
    Debug(cout << "input == " << input << endl);
}
```

If `DEBUG` is defined, then the above is expanded to:

```
void function(int input)
{
    cout << "input == " << input << endl;
}
```

If `DEBUG` is not defined, we get:

```
void function(int input)
{
}
```

## Example 2.1 – Debug Output

Demo

This example will demonstrate how to add COUT statements to get some insight into how the program is behaving. It will do this by utilizing `#define` directives, `#ifdef` directives, and asserts.

Problem

Write a program to compute an individual's pay taking into account time-and-a-half overtime.

```
What is your hourly wage? 12  
How many hours did you work? 39.5  
Pay: $ 474.00
```

From this example, insert debug code to help discover the location of bugs.

```
What is your hourly wage? 12  
How many hours did you work? 39.5  
main: hourlyWage: 12  
main: hoursWorked: 39.5  
computePay(12.00, 39.50)  
computePay: Regular rate  
Pay: $ 474.00
```

The first thing to do is to add a mechanism to easily put debug code in the program.

```
#ifdef DEBUG  
#define Debug(x) x  
#else  
#define Debug(x)  
#endif // !DEBUG
```

Now, if `DEBUG` is not defined, none of the code in `Debug()` gets executed. If it is defined, then we can get all our debug output. The `computePay()` function with `Debug()` code is:

```
float computePay(float hourlyWage, float hoursWorked)  
{  
    Debug(cout << "computePay(" << hourlyWage << ", " << hoursWorked << ")\n");  
    float pay;  
  
    // regular rate  
    if (hoursWorked < CAP)  
    {  
        Debug(cout << "computePay: Regular rate\n");  
        pay = hoursWorked * hourlyWage;                // regular rate  
    }  
  
    // overtime rate  
    else  
    {  
        Debug(cout << "computePay: Overtime\n");  
        pay = (CAP * hourlyWage) +                    // first 40 normal  
              ((hoursWorked - CAP) * (hourlyWage * OVERTIME)); // balance overtime  
    }  
    return pay;  
}
```

See Also

The complete solution is available at [2-1-debugOutput.cpp](#) or:

```
/home/cs124/examples/2-1-debugOutput.cpp
```



## Driver programs

Drivers are special programs designed to test a given function. This is an exceedingly important part of the programming process. An aerospace engineer would never put an untested engine in an airplane. He would instead mount the engine on a testing harness and run it through the paces. Only after exhaustive testing would he feel confident enough to put the engine in the airplane. We should also treat new functions with skepticism. When we validate functions before integrating them into the larger program, it is far easier to localize problems. After the function has been validated, then we can safely copy it to the project. Typically drivers consist of just the function `main()` and the function to be tested. Consider, for example, the prototype for the function `computePay()`:

```
float computePay(float hourlyWage, float hoursWorked);
```

A driver program for `computePay()` might be:

```

/*****
 * MAIN
 * Simple driver for computePay()
 *****/
int main()
{
    float wage;
    cout << "wage: ";                // get the data as quickly as possible
    cin  >> wage;

    float hours;
    cout << "hours: ";               // again, just the simplest prompt
    cin  >> hours;

    cout << "computePay("
         << "hourlyWage = " << wage << ", " // show what was passed
         << "hoursWorked = " << hours
         << ") == "
         << computePay(wage, hours)         // show what was returned
         << endl;

    return 0;
}

```

Observe how the driver program is just a bare-bones program whose only purpose is to prompt the user for the data to pass to the function and to display the results. When you use the driver-program development methodology, you:

1. Start with a blank file. The only thing this program will do is test your function.
2. Write the function. As long as the coupling is loose, this should not be too complex.
3. Create a `main()` that only calls your function. This is typically done in three steps:
  - a. First call your function with the simplest possible data.
  - b. If your function requires any parameters, create simple `cin` statements in `main()` to fetch that data directly from the user.
  - c. If your function returns something, display the results directly on the screen so it is easy to verify how the function responded to input.
4. Test your function with a variety of input. Start with simple input and work to more complex scenarios.

## Example 2.1 – Driver

### Demo

This example will demonstrate how to write a simple driver program to test a function we used in Project 1: `computeTax()`.

### Solution

The driver program exists entirely in `main()`. We start with the prototype of the function we are testing:

```
double computeTax(double incomeMonthly);
```

There will be two parts: fetching the data from the user for the function parameters, and displaying the output of the function.

```
/* *****  
 * MAIN  
 *   A simple driver program for computeTax()  
 * ***** */  
int main()  
{  
    // get the income  
    double income;           // the inputs to the function being  
    cout << "Income: ";      //   tested is gathered directly from  
    cin  >> income;          //   the user and sent to the function  
  
    // call the function and display the results  
    cout << "computeTax(" << income << ") == " // what we are sending...  
         << computeTax(income)                 // what the output is  
         << endl;  
  
    return 0;  
}
```

Driver programs are very streamlined and simple. Once we have tested our function, we can safely throw them away.

### Challenge

As a challenge, write a driver program for `computeTithe()` from your Project 1 code.

### See Also

The complete solution is available at [2-1-driver.cpp](#) or:

```
/home/cs124/examples/2-1-driver.cpp
```





This process works well when you are in the development phase of the project. You can also use the driver-program technique when you are in the testing and debugging phase of the project. This can be accomplished by modifying our `main()` to be a driver for any function in the program. Recall the `computeTax()` function from Project 1. We might think we have worked out all the bugs of the functions before they were integrated together. When running the program, however, it becomes apparent that something is broken.

Consider the following `main()` from Project 1 after it has been modified to test `computeTithing()`. Note how we use a `return` statement to ensure only the top part of the function is executed.

```

/*****
 * MAIN
 * Keep track of a monthly budget
 *****/
int main()
{
    double incomeTest = getIncome();           // use the get function or a cin
    cout << computeTithing(incomeTest) << endl; // simple display of the output
    return 0;                                  // return ensures we exit here and
                                              // only test computeTithing()
                                              // rest of main below here

    // instructions
    cout << "This program keeps track of your monthly budget\n";
    cout << "Please enter the following:\n";
    // prompt for the various data
    double income      = getIncome();
    double budgetLiving = getBudgetLiving();
    double actualLiving = getActualLiving();
    double actualTax    = getActualTax();
    double actualTithing = getActualTithing();
    double actualOther  = getActualOther();

    // display the results
    display(income, budgetLiving, actualTax, actualTithing,
            actualLiving, actualOther);
    return 0;
}

```

The driver program technique has been used for almost all the assignments we have done this semester.

## Stub functions

A stub is a placeholder for a forthcoming replacement promising to be more complete. In the context of designing and building a program, a stub is a tool enabling us to put a placeholder for all the functions in our structure chart without getting bogged down with how the functions will work. In other words, stubs allow us to:

- **Get Started:** Stubs allow us to get the design from the Structure Chart into our code before we have figured out how to implement the functions themselves. This helps answer the question “how do I start on this project?”
- **Figure out data flow.** Because stubs include the parameters passed between functions, you can model information flow early in the development process.
- **Always have your program compile.** A program completely stubbed-out will compile even though it does not do anything yet. Then, as you implement individual functions, any compile errors you encounter are localized to the individual function you just implemented.

Consider the computeTax function from Project 1. The following would be an example of a stub:

```
float computeTax(float income)
{
    // stub for now...
    return 0.0;
}
```

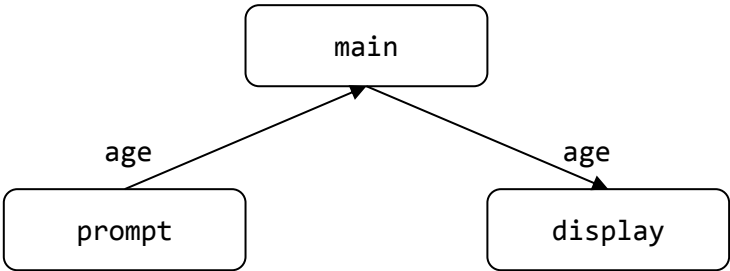
### Example 2.1 – Stub Functions

Demo

This example will demonstrate how to turn a structure chart into stub functions.

Problem

Write stub functions for the following structure chart.



Solution

The stubbed version of this program would be:

```
int prompt()                // don't bother with the function comment block
{                            // with stubbed functions. We will do them later
    return 0;                // make sure to return some value because the return
}                            // type is not void in this function

void display(int age)        // all the parameters need to be present in the stub
{                            // even if the body of the functions are empty.
}

int main()                  // make sure the stubs call all the children
{                            // functions so we can make sure the data flow
    display(prompt());        // works correctly. This should enable us to
    return 0;                // implement the functions in any order we choose
}
```

Observe how the stubbed functions consist of:

- **Prototypes:** the function name, return type, and parameters.
- **Empty body:** except for a return statement, the body is mostly empty.
- **Called functions:** include code to call the child functions. It is OK to use dummy parameters if none are known.

Challenge

As a challenge, try to stub Project 1. A sample solution is available at [2-1-stubbed.cpp](#) or:

```
/home/cs124/examples/2-1-stubbed.cpp
```

## Problem 1

Which of the following most clearly illustrates the concept of coupling?

- The parameters passed to a function
- The task that the function performs
- The subdivision of components in the function or program
- The degree in which a function can be used for different purposes

*Please see page 117 for a hint.*

## Problem 2

Identify the type of coupling that the following function exhibits:

```
float lastGrade = 0.0;

float getGPA()
{
    cout << "What is your grade? ";
    cin >> lastGrade;

    return lastGrade;
};
```

Answer:

---

*Please see page 117 for a hint.*

## Problem 3

Create a stub for the following function:

```
void displayIncome(float income)
{
    // configure output to display money
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    // display the income
    cout << "Your income is: $"
        << income
        << endl;
};
```

Answer:

*Please see page 140 for a hint.*

### Problem 4

Create a stub for the following function:

```
float getIncome()
{
    float income;

    // prompt
    cout << "Please enter your income: ";
    cin >> income;

    return income;
}
```

Answer:

*Please see page 140 for a hint.*

### Problem 5

Create a driver program for the following function:

```
float sqrt(float value);
```

Answer:

*Please see page 138 for a hint.*

### Problem 6

Create a driver program for the following function:

```
void displayTable(int numDaysInMonth, int offset);
```

Answer:

*Please see page 138 for a hint.*

### Problem 7

Create an assert to verify the following variable is within the expected range:

```
float gpa
```

Answer:

*Please see page 132 for a hint.*

### Problem 9

What asserts would you add to the beginning of the following function:

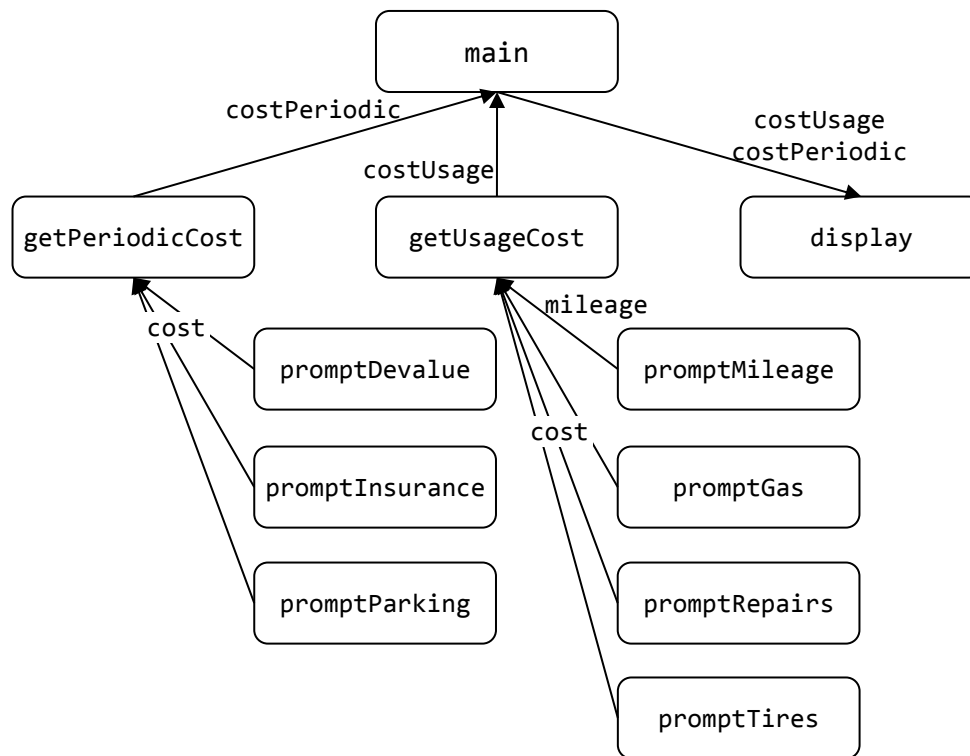
```
void displayDate(int day, int month, int year);
```

Answer:

*Please see page 49 for a hint.*

## Assignment 2.1

Sue wants to write a program to help her determine how much money she is spending on her car. Specifically, she wants to know how much she spends per day having the car sit in her driveway and how much she spends per mile driving it. While working through this problem, she came up with the following structure chart:



Please create stub functions for all the functions in Sue's program. In other words, write a program to stub out every function represented in the above structure chart. If a function calls another function (ex: `getPeriodicCost()` calls `promptParking()`), then make sure that function call is in the stub. Finally, make sure all the parameters and return values from the structure chart are represented in the stub functions.

One final note: you do not need to have function headers for each individual function.

Please:

- Create stub functions for all the functions mentioned in the structure chart.
- If you were able to do this, then enter the following code in the function `display()`:

```
cout << "Success\n";
```

- Use the following testbed:

```
testBed cs124/assign21 assign21.cpp
```

- Submit to Assignment 21

*Please see page 140 for a hint.*

## 2.2 Designing Algorithms

Sue just spent a half hour writing the code for a certain function before she realized that she got the design all wrong. Not only was her code broken, but her entire approach to the problem was all wrong. Sue is frustrated: writing code is hard! If only there was a way to design a function without having to go through the work of getting it to compile...

### Objectives

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

- Recite the pseudocode keywords.
- Understand the degree of detail required for a pseudocode design.
- Generate the pseudocode corresponding to a C++ function.

### Prerequisites

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

- Create a map of a program using structure charts (Chapter 2.0).

### Reading

All of the reading in this section will consist of the pseudocode videos:

- [Design First](#): This video discusses why it is important to design a problem before the implementation is begun.
- [When to Design](#): This video will discuss different times in the software development process when design activities are most effective.
- [Different Design Approaches](#): This video will introduce four techniques to drafting a design: the paragraph method, flowchart, structure diagram, and pseudocode.
- [Using Pseudocode](#): How pseudocode can be used as a tool to more effectively write software.
- [Rules of Pseudocode](#): The conventions and rules of pseudocode.
- [Pseudocode Keywords](#): The seven classes of keywords that are used in pseudocode.



### Another design tool

Pseudocode, along with the Structure Chart, is one of our most powerful design tools. While the Structure Charts is concerned with what function we have in our program and how the functions interact with each other, Pseudocode is concerned with what goes on inside individual functions.

There are several reasons why we need to draft a solution before we start writing code. First, we need to be able to think big before getting bogged down in the details. Programming languages resist this process; they need you to always work at the most detailed level. This can be very frustrating; you are trying to solve a large problem but the language keeps forcing you to specify data types and work through syntax errors.

## Using pseudocode

Pseudocode is a tool helping the programmer bridge the high-level English description of a problem and the C++ solution. To illustrate how this works, we will begin with a natural language definition of a problem. In this case, how to compute your tax liability with a simple 2 tier tax table.

Given a user's income, compute their tax burden by looking up the appropriate formula on the following table:

| If taxable income is over-- | But not over-- | The tax is:                                     |
|-----------------------------|----------------|---|
| \$0                         | \$15,100       | 10% of the amount over \$0                      |
| \$15,100                    | \$61,300       | \$1,510.00 plus 15% of the amount over 15,100   |
| \$61,300                    | \$123,700      | \$8,440.00 plus 25% of the amount over 61,300   |
| \$123,700                   | \$188,450      | \$24,040.00 plus 28% of the amount over 123,700 |
| \$188,450                   | \$336,550      | \$42,170.00 plus 33% of the amount over 188,450 |
| \$336,550                   | no limit       | \$91,043.00 plus 35% of the amount over 336,550 |

First, we will introduce structure in the problem definition by dividing the process into discrete steps. We are entering the realm of pseudocode here, but there is still far too much natural language to be much of a help.

Determine tax bracket the user's income according to the ranges in this table:

| Min       | Max       | Bracket |
|-----------|-----------|---------|
| \$0       | \$15,100  | 10%     |
| \$15,100  | \$61,300  | 15%     |
| \$61,300  | \$123,700 | 25%     |
| \$123,700 | \$188,450 | 28%     |
| \$188,450 | \$336,550 | 33%     |
| \$336,550 | no limit  | 35%     |

Apply the appropriate formula:

| Bracket | Formula   |
|---------|---|
| 10%     | 10% of the amount over \$0                        |
| 15%     | \$1,510.00 plus 15% of the amount over \$15,100   |
| 25%     | \$8,440.00 plus 25% of the amount over \$61,300   |
| 28%     | \$24,040.00 plus 28% of the amount over \$123,700 |
| 33%     | \$42,170.00 plus 33% of the amount over \$188,450 |
| 35%     | \$91,043.00 plus 35% of the amount over \$336,550 |

Return the results back to the caller.



Next we will be more precise with our equations and fill in more detail whenever possible. We are beginning to specify how things will be accomplished, not just what will be accomplished. Observe how our pseudocode is moving closer to our programming language with every step.

```

IF income is less than $15,100 then
    tax is 10% of the amount over $0
IF income between $15,100 and $61,300 then
    tax is $1,510 plus 15% of the amount over $15,100
IF income between $61,300 and $123,700 then
    tax is $8,440 plus 25% of the amount over $61,300
IF income between $123,700 and $188,450 then
    tax is $24,040 plus 28% of the amount over $123,700
IF income is above $188,450 then
    tax is $91,043 plus 35% of the amount over $188,450

```

Finally, we reduce all operations to pseudocode keywords. Now we are ready to start writing code. The problem has been solved and now it is just a matter of translating the syntax-free pseudocode into the specific syntax of the high level language.

```

computeTax(income)

    IF ($0 ≤ income < $15,100)
        tax ← income * 0.10
    IF ($15,100 ≤ income < $61,300)
        tax ← $1,510 + 0.15 * (income - $15,100)
    IF ($61,300 ≤ income < $123,700)
        tax ← $8,440 + 0.25 * (income - $61,300)
    IF ($123,700 ≤ income < $188,450)
        tax ← $24,040 + 0.28 * (income - $123,700)
    IF ($188,450 ≤ income < $336,550)
        tax ← $42,170 + 0.33 * (income - $188,450)
    IF ($336,550 ≤ income)
        tax ← $91,043 + 0.35 * (income - $336,550)

    RETURN tax
END

```

Always remember that pseudocode is just a design tool. With the exception of a few assignments in this class, your purpose is to develop great software rather than write pseudocode. Therefore, you should use pseudocode only as far as it helps you to develop software. This means that sometimes you will design right down to the pseudocode keywords while other times you will be able to stop designing before that point because the solution presents itself.

## Pseudocode keywords

There are seven classes of keywords: receive, send, math, remember, compare, repeat, and call functions.

### Receive

A computer can receive information from a variety of input sources, such as keyboard, mouse, a network, a sensor, or wherever.

| Keyword | Description  | Example           |
|---------|--|-------------------|
| READ    | Receive information from a file                        | READ studentGrade |
| GET     | Receive input from a user, typically from the keyboard | GET income        |

### Send

A computer can send information to the console, display it graphically, write to a file, or operate on a device.

| Keyword | Description   | Example                 |
|---------|---|-------------------------|
| PRINT   | When sending data to a permanent output device like a printer | PRINT full student name |
| WRITE   | When writing data to a file                                   | WRITE record            |
| PUT     | When sending data to the screen                               | PUT instructions        |
| PROMPT  | Just like PUT except always preceding a GET instruction       | PROMPT for user name    |

### Arithmetic

Most processors have the built-in capability to perform the following operations:

| Keyword            | Description  | Example               |
|--------------------|--|-----------------------|
| ()                 | Parentheses are used to override the default order of operations | $c = (f - 32) * 5/9$  |
| * x<br>/ ÷ mod div | Any mathematical convention can be used                          | numWeeks = days div 7 |
| + -                | Addition / subtraction   | SET count = count + 1 |
| √ π                | Common mathematical values or operations                         | PUT √10               |
| ≤ ≠                | Common comparison operations                                     | IF grade ≥ 60         |

### Remember

A computer can assign a value to a variable or a memory location

| Keyword       | Description                    | Example         |
|---------------|--------------------------------|-----------------|
| SET<br>=<br>← | Assign a value to a variable ← | SET answer ← 42 |

## Compare

A computer can compare two values and select one of two alternate actions

| Keyword     | Description   | Example  |
|-------------|---|--|
| IF<br>ELSE  | For two possible outcomes   | IF income / 10 ≤ tithingPaid<br>PUT full tithe message<br>ELSE<br>PUT scripture from Malachi                     |
| SWITCH CASE | For multiple possible outcomes.<br>This will be discussed in more detail in Chapter 3.5 | SWITCH option<br>CASE 1<br>PUT great choice!<br>CASE 2<br>PUT could be better<br>CASE 3<br>PUT please reconsider |

## Repeat

A computer can repeat a group of actions.

| Keyword | Description   | Example                                |
|---------|---|--|
| WHILE   | When repeating through the same code more than once | WHILE studentGrade < 60<br>takeClass() |
| FOR     | When counting                                       | FOR count = 1 to 10 by 2s<br>PUT count |

## Functions

A computer can call a function and pass parameters between functions.

| Usage     | Conventions  | Example  |
|-----------|--|--|
| Declaring | Functions are named.<br>Input parameters are enumerated.<br>Statements are indented.<br>RETURN values.<br>END. | computeTithing( income )<br>SET tithing = income / 10<br>RETURN tithing<br>END |
| Calling   | Call by name<br>Specify parameters   | PUT computeTithing(income)   |



### Sue's Tips

On the surface, it may seem like pseudocode is no different than C++. Why learn another language when C++ already does the job? The short answer is that pseudocode is less detailed than C++ so you can concentrate on more high-level design decisions without getting bogged down in the minutia of detail that C++ demands. The long answer is a bit more complicated.

In its truest form, pseudocode is syntax free. This means that anything goes! It starts very free-form much like natural language (English). As you refine your thinking and work out the program details, it begins to take on the structure of a high-level programming language. When you get down to the pseudocode keywords listed above, you have worked out all the design decisions. While it is not always necessary to develop pseudocode to this level of detail, it is an important skill to develop. This is why pseudocode is an important CS 124 skill to learn.

| Example 2.2 – Compute Pay |   |
|---------------------------|---|
| Demo                      | <p>This example will demonstrate the relationship between pseudocode and C++. Specifically, it will show what kinds of details are necessary in pseudocode (variable names, equations, and program logic) and which are not (variable declarations, C++ syntax, and comments).</p>  |
| Problem                   | <p>Write the pseudocode corresponding to the following C++ function:</p> <pre>/* *****  * COMPUTE PAY  *   Based on the user's wage and hours worked, compute the pay  *   taking into account time-and-a-half overtime  * ***** */ float computePay(float hourlyWage, float hoursWorked) {     float pay;      // regular rate     if (hoursWorked &lt; 40)         pay = hoursWorked * hourlyWage;           // regular rate      // overtime rate     else         pay = (40.0 * hourlyWage) +                // first 40 normal               ((hoursWorked - 40.0) * (hourlyWage * 1.5)); // balance overtime      return pay; }</pre> |
| Solution                  | <p>The pseudocode for computePay() is the following:</p> <pre>computePay(hourlyWage, hoursWorked)   IF hoursWorked &lt; 40     SET pay = hoursWorked x hourlyWage   ELSE     SET pay = (40 x hourlyWage) + ((hoursWorked - 40) x (hourlyWage x 1.5))   RETURN pay END</pre> <p>Observe how the pseudocode completely represents the logic of the program without worrying about data-types, declaring variables, or the syntax of the language.</p>   |
| Challenge                 | <p>As a challenge, look at the Project 1 definition. The pseudocode for most of the functions is provided. Compare your C++ code with the provided pseudocode. See if you can write the pseudocode for the remaining functions (computeTithing(), getActualTax(), etc.).</p>  |

### Problem 1

What is the value of `b` at the end of execution?

```
bool vegas(bool b)
{
    b = false;
    return true;
}

int main()
{
    bool b;
    vegas(b);
    return 0;
}
```

Answer:

---

*Please see page 65 for a hint.*

### Problem 2

What is the output when the user types 'a'?

```
void function(char &value)
{
    cin >> value;
    return;
}

int main()
{
    char input = 'b';
    function(input);
    cout << input << endl;
    return 0;
}
```

Answer:

---

*Please see page 65 for a hint.*

### Problem 3

Which of the following is not a basic computer operation?

- Receive information
- Connect to the network
- Perform math
- Compare two numbers

*Please see page 145 for a hint.*

### Problem 4

What is wrong with the following pseudocode?

```
IF age < 18  
  PUT message about not being old enough to vote
```

Answer:

---

*Please see page 49 for a hint.*

### Problem 5

Which of the following is the correct pseudocode for sending a message to the user?

DISPLAY

cout << "The following are the instructions\n";

WRITE instructions on the screen

PUT instructions on the screen

*Please see page 148 for a hint.*

### Problem 6

Which of the following is the correct pseudocode for computing time-and-a-half overtime?

pay = (hours - 40) \* pay \* 1.5 + 40 \* pay

pay = ((float)hours - 40.0) \* pay \* 1.5 + 40 \* (float)pay;

pay = hours - 40 \* pay \* 1.5 + 40 \* pay

pay = hours over 40 times time-and-a-half plus regular pay for first 40 hours

*Please see page 148 for a hint.*

### Problem 7

Which is the pseudocode command to differentiate between two options?

IF

COMPARE

=

same?

*Please see page 149 for a hint.*

### Problem 8

Which pseudocode command displays data on the screen?

*Please see page 148 for a hint.*

### Problem 9

Write the pseudocode corresponding to the following C++:

```
float computeTithing(float income)
{
    float tithing;

    // Tithing is 10% of our income. Please
    // see D&C 119:4 for details or questions
    tithing = income * 0.10;

    return tithing;
}
```

Answer:

*Please see page 49 for a hint.*

### Problem 10

Write the pseudocode for a function to determine if a given year is a leap year:

According to the Gregorian calendar, which is the civil calendar in use today, years evenly divisible by 4 are leap years, with the exception of centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

Answer:

*Please see page 150 for a hint.*

### Problem 11

Write the pseudocode for a function to compute how many days are in a given year. Hint: the answer is 365 or 366.:

Answer:

*Please see page 150 for a hint.*

### Problem 12

Write the pseudocode for a function to display the multiples of 7 under 100.

Answer:

*Please see page 148 for a hint.*



## Assignment 2.2

Please create pseudocode for the following functions. The pseudocode is to be turned in by hand in class for face-to-face students and submit it as a PDF for online students:

### Part 1: Temperature Conversion

```
int main()
{
    // Get the temperature from the user
    float tempF = getTemp();

    // Do the conversion
    float tempC = (5.0 / 9.0) * (tempF - 32.0);

    // display the output
    // I don't want showpoint because I don't want to show a point!
    cout.setf(ios::fixed);
    cout.precision(0);
    cout << "Celsius: " << tempC << endl;

    return 0;
}
```

### Part 2: Child tax credit

```
int main()
{
    // prompt for stats
    double income = getIncome();
    int numChildren = getNumChildren();

    // display message
    cout.setf(ios::fixed);
    cout.precision(2);
    cout << "Child Tax Credit: $ ";
    if (qualify(income))
        cout << 1000.0 * (float)numChildren << endl;
    else
        cout << 0.0 << endl;

    return 0;
}
```

### Part 3: Cookie monster

```
void askForCookies()
{
    // start with no cookies :-(
    int numCookies = 0;

    // loop until the little monster is satisfied
    while (numCookies < 4)
    {
        cout << "Daddy, how many cookies can I have? ";
        cin >> numCookies;
    }

    // a gracious monster to be sure
    cout << "Thank you daddy!\n";
    return;
}
```

*Please see page 150 for a hint.*

## 2.3 Loop Syntax

Sue's little brother is learning his multiplication facts and has asked her to write them on a sheet of paper. Rather than spend a few minutes to hand-write the table, she decides to write a program instead. This program will need to count from 1 to 10, so a FOR loop is the obvious choice.

### Objectives

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop.
- Create a loop to solve a simple problem.

### Prerequisites

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

- Convert a logic problem into a Boolean expression (Chapter 1.5).
- Generate the pseudocode corresponding to a C++ function (Chapter 2.2).

### Overview

This is the first of a three part series on how to use loops to solve programming problems. The first part will focus on the mechanism of loops, namely the syntax.

Loops are mechanisms to allow a program to execute the same section of code more than once. This is an important tool for cases when an operation needs to happen repeatedly, when counting is required to solve a problem, and when the program needs to wait for an event to occur.

There are three types of loops in C++: WHILE, DO-WHILE, and FOR:

| while   | do-while  | for   |
|---|---|---|
| A WHILE loop is good for repeating through a given block of code multiple times.      | Same as WHILE except we always execute the body of the loop at least once.                  | Designed for counting, usually meaning we know where we start, where we end and what changes.         |
| <pre>{   while (x &gt; 0)   {     x--;     cout &lt;&lt; x &lt;&lt; endl;   } }</pre> | <pre>{   do   {     x--;     cout &lt;&lt; x &lt;&lt; endl;   }   while (x &gt; 0); }</pre> | <pre>{   for (x = 10;        x &gt; 0;        x--)   {     cout &lt;&lt; x &lt;&lt; endl;   } }</pre> |

## WHILE

The simplest loop is the WHILE statement. The WHILE loop will continue executing the body of the loop until the controlling Boolean expression evaluates to false. The syntax is:

```
while (<Boolean expression>)  
    <body statement>;
```

As with the IF statement, we can always have more than one statement in the body of the loop by adding curly braces {}s:

```
while (<Boolean expression>)  
{  
    <body statement1>;  
    <body statement2>;  
    ...  
}
```

Observe how the body of the loop is indented three spaces exactly as the body of an IF statement is indented.



### Sue's Tips

The WHILE loop keeps iterating as long as the Boolean expression evaluates to true. This may seem counter-intuitive at first. Some find it easier to think that the loop keeps iterating until the Boolean expression evaluates to true.

One way to keep this straight in your mind is to read the code as “while <condition> continue to <body>.” For example consider the following code:

```
while (input < 0)  
    cin >> input;
```

This would read “While input less-than zero, continue to prompt for new input.”

It is possible to count with a WHILE loop. In this case, you initialize your counter before the loop begins, specify the continuation-condition (what must remain true as we continue through the loop) in the Boolean expression, and specify the increment logic in the body:

```
{  
    int count = 1;  
    while (count <= 10)                // continue as long as this evaluates to true  
    {                                  // use {}s because there is more than one  
        cout << count << endl;         // statement in the body of the loop  
        count++;  
    }  
}
```

This code will display the numbers 1 through 10 on the screen, each number on its own line.

## Example 2.3 – While Loop

Demo

This example will demonstrate the syntax of a simple WHILE loop.

Problem

Write a program to prompt the user for his letter grade. If the grade is not in the valid range (A, B, C, D, or F), then the program will display an error message and prompt again.

```
Please enter your letter grade: B  
Your grade is B
```

...or...

```
Please enter your letter grade: E  
Invalid grade. Please enter a letter grade {A,B,C,D,F} G  
Invalid grade. Please enter a letter grade {A,B,C,D,F} C  
Your grade is C
```

The most challenging part of this problem is the Boolean expression capturing whether the user input is in the valid range. Anything that is not an A, B, C, D, or F is classified as invalid.

Solution

```
char getGrade()  
{  
    char grade;    // the value we will be returning  
  
    // initial prompt  
    cout << "Please enter your letter grade: ";  
    cin >> grade;  
  
    // validate the value  
    while (grade != 'A' && grade != 'B' && grade != 'C' &&  
           grade != 'D' && grade != 'F')  
    {  
        cout << "Invalid grade. Please enter a letter grade {A,B,C,D,F} ";  
        cin >> grade;  
    }  
  
    // return when done  
    return grade;  
}
```

Challenge

As a challenge, modify `getGrade()` so either uppercase or lowercase letter grades are accepted. This can be done by either doubling the size of the Boolean expression to include lowercase letters or by converting the grade to uppercase if it is lowercase.

See Also

The complete solution is available at [2-3-while.cpp](#) or:

```
/home/cs124/examples/2-3-while.cpp
```



## DO-WHILE

The DO-WHILE loop is the same as the WHILE loop except the controlling Boolean expression is checked after the body of the loop is executed. This is called a **trailing-condition** loop, while the WHILE loop is a **leading-condition** loop. As with the WHILE statement, the loop will continue until the controlling Boolean expression evaluates to `false`. The syntax is:

```
do
    <body statement>;
while (<Boolean expression>);
```

DO-WHILE loops are used far less frequently than the aforementioned WHILE loops. Those scenarios when the DO-WHILE loop would be the tool of choice center around the need to ensure the body of the loop gets executed at least once. In other words, it is quite possible the controlling Boolean expression in a WHILE loop will evaluate to `false` the first time through, removing the possibility the body of the loop is executed. This is guaranteed to not happen with the DO-WHILE loop because the body always gets executed *first*.

### Example

Consider the following code prompting the user for his age:

```
{
    int age;
    do
    {
        cout << "What is your age? ";
        cin >> age;
    }
    while (age < 0);
}
```

// the "do" keyword on its own line  
// use {}s when there is more than one  
// statement in the body of the loop  
  
// keep the {}s on their own line  
// continue until this evaluates to false

In this example, we want to prompt the user for his age at least once. The code will continue prompting the user for his age as long as the user enters negative numbers. Note how the `while` part of the DO-WHILE loop is on a separate line from the `{}`s. This is because the style guide specifies that `{}`s must be on their own lines.

### Sam's Corner



It turns out that the WHILE loop and the DO-WHILE loop solve exactly the same set of problems. Any DO-WHILE loop can be converted to a WHILE loop by bringing one instance of the body outside the loop. In the above example, the equivalent WHILE loop would be:

```
{
    int age;
    cout << "What is your age? ";
    cin >> age;

    while (age < 0)
    {
        cout << "What is your age? ";
        cin >> age;
    }
}
```

// one copy of the body outside the loop  
  
// same condition as the DO-WHILE  
  
// second copy in the body of the loop

Note how redundant it is to have the second copy of the body outside the loop. This is the primary advantage of the DO-WHILE: to reduce the redundancy.

## Example 2.3 – Do-while Loop

Demo

This example will demonstrate how to use a DO-WHILE loop to validate user input.

Problem

Write a program to sum the numbers the user entered. The program will continue to prompt the user until the value zero (0) is typed.

```
Please enter a collection of integer values. When
you are done, enter zero (0).
> 10
> 15
> -7
> 0
The sum is: 18
```

Solution

Because the user needs to be prompted at least once, a DO-WHILE loop is the right tool for the job.

```
int promptForNumbers()
{
    // display instructions
    cout << "Please enter a collection of integer values. When\n"
         << "\tyou are done, enter zero (0).\n";
    int sum = 0;
    int value;

    // perform the loop
    do
    {
        // prompt for value
        cout << "> ";
        cin >> value;

        // add value to sum
        sum += value;
    }
    while (value != 0);
    // continue until the user enters zero

    // return and report
    return sum;
}
```

Challenge

As a challenge, can you modify the above function so the value zero is not added to the sum?

To take this one step further, modify the above problem so -1, not 0, is the terminating condition. This will require you to modify the instructions as well.

See Also

The complete solution is available at [2-3-doWhile.cpp](#) or:

```
/home/cs124/examples/2-3-dowhile.cpp
```



# FOR

The final loop is designed for counting. The syntax is:

```
for (<initialization statement>; <Boolean expression>; <increment statement>)
    <body statement>;
```

Here the syntax is quite a bit more complex than its WHILE and DO-WHILE brethren.

```
for (int count = 0; count < 5; count++)
    cout << count << endl;
```

## Initialization:

The first statement to be executed in a loop.

- Can be any statement.
- We can declare and initialize a variable inside the loop:

```
for (int i = 0; ...
```

- We can initialize more than one variable that is already defined elsewhere in the code:

```
for (j = 0, k = 0; ...
```

- We can also leave it empty:

```
for (; i < 10; i++)
```

## Boolean expression:

Is executed immediately before the body of the loop.

- Can be any expression.
- As long as the expression evaluates to **true**, the loop continues:
- If it is left empty, the expression evaluates to **true**. This means it will loop forever:

```
for (i = 0; ; i++)
```

## Increment:

Is executed immediately after the body of the loop.

- Can be any statement.
- Usually we put a ++ or -- here:
- You can put more than one statement here:

```
for (... ; ...; i++, j--)
```

- Can be left empty:

```
for (; i < 10; )
```

While the syntax of the FOR loop may look quite complex, it has the three things any counting problem needs: where to start (initialization), where to end (Boolean expression), and how much to count by (the increment statement). For example, a FOR loop to give a countdown from 10 to zero would be:

```
{
    // a countdown, just like what Cape Kennedy uses
    for (int countDown = 10; countDown >= 0; countDown--)
        cout << countDown << endl;
}
```



## Sue's Tips

While it may seem difficult to remember what the three fields of a FOR loop (Initialization, Boolean expression, and increment) are for, there is a memory clue to help you remember. Consider the loop to count to four:

```
for (int count = 1; count < 5; count++)
    cout << count << endl;
```

This reads “For count equals one, *as long as* count is less than five, add one to count.”

## Example 2.3 – For Loop

### Demo

This example will demonstrate how to use a FOR loop. It will allow the user to specify each of the three components of the loop (Initialization, Boolean expression, and Increment). In many ways, this is a driver program for the FOR loop.

### Problem

Write a program prompt the user the parameters for counting. The program will then display the numbers in the specified range.

```
What value do you want to start at? 4
What value do you want to end at? 14
What will you count by (example: 2s): 3
    4
    7
   10
   13
```

### Solution

The following program will count from start to end.

```
int main()
{
    // start
    cout << "What value do you want to start at? ";
    int start;
    cin >> start;

    // end
    cout << "How high do you want to count? ";
    int end;
    cin >> end;

    // increment
    cout << "What will you count by (ex: 2s): ";
    int increment;
    cin >> increment;

    // count it
    for (int count = start; count <= end; count += increment)
        cout << "\t" << count << endl;

    return 0;
}
```

Notice the three parts of a FOR loop: the starting condition (`int count = start`), the continuation condition (`count <= end`), and what changes every iteration (`count += increment`). As long as the continuation condition is met, the loop will continue and the body of the loop will execute.

### Challenge

As a challenge, modify the above program to ensure that  $\text{start} \leq \text{end}$  if increment is positive, and  $\text{start} \geq \text{end}$  if increment is negative.

### See Also

The complete solution is available at [2-3-for.cpp](#) or:

```
/home/cs124/examples/2-3-for.cpp
```





## Problem 1

What is the output?

```
void function(int a, int &b)
{
    a = 0;
    b = 0;
}

int main()
{
    int a = 1;
    int b = 2;

    function(a, b);

    cout << "a == " << a << '\t'
         << "b == " << b << endl;
}
```

Answer:

---

*Please see page 65 for a hint.*

## Problem 2

What is the output?

```
int value = 1;

int main()
{
    int value = 2;
    cout << value;

    if (true)
    {
        int value = 3;
        cout << value;

        {
            int value = 4;
            cout << value;
        }
        cout << value;
    }
    cout << value << endl;
    return 0;
}
```

Answer:

---

*Please see page 67 for a hint.*

### Problem 3

What is the output?

```
{
    int j = 10;

    for (int i = 1;
        i < 3;
        i++)
        j++;

    cout << i << endl;

    return 0;
}
```

Answer:

*Please see page 161 for a hint.*

### Problem 4-7

Write the code to implement each of the following loops:

Count from 1 to 10

Keep asking the user for input until he enters a value other than 'q'

Display the powers of two below 2,000

Sum the multiples of seven below 100

*Please see page 161 for a hint.*

### Problem 8

Which of the following has no syntax errors?

```
do while (true)
    cout << "Infinite loop!\n";
```

```
do
    cout << "Infinite loop!\n";
while (true);
```

```
do (true)
    cout << "Infinite loop!\n";
```

```
while (true)
    cout << "Infinite loop!\n";
do;
```

*Please see page 159 for a hint.*

### Problem 9-10

Write a program that keeps prompting the user for a number until he inputs the number 0:

Use a WHILE loop

Use a DO-WHILE loop

*Please see page 159 for a hint.*

### Problem 11

Write the code to count down from 100 to 10 in steps of 10:

```
100
90
80
70
60
50
40
30
20
10
```

Answer:

*Please see page 162 for a hint.*

### Problem 12-13

Write the code to prompt the user for 5 numbers and display the sum

```
Please enter 5 numbers:
#1: 54
#2: 99
#3: 12
#4: 65
#5: 34
Sum: 264
```

Answer in pseudocode:

Answer in C++:

*Please see page 161 for a hint.*

## Assignment 2.3

Sue's silly brother Steve has a teacher who loves to give tons of math homework. This week, the assignment is to add all the multiples of 7 that are less than 100. Last week, he had to add all the multiples of 3 that are less than 100. Sue wants to make sure that her brother gets a 100% on each assignment so she decided to write a program to validate each assignment.

### Example

User input in underline.

```
What multiples are we adding? 5  
The sum of multiples of 5 less than 100 are: 950
```

Another example:

```
What multiples are we adding? 7  
The sum of multiples of 7 less than 100 are: 735
```

### Assignment

Make sure you run test bed with:

```
testBed cs124/assign23 assignment23.cpp
```

Don't forget to submit your assignment with the name "Assignment 23" in the header.

*Please see page 162 for a hint.*

## 2.4 Loop Output

Sue can't seem to find the bug in her assignment. The code looks right and it compiles, but the loop keeps giving her different output than she expects. How can she ever hope to find the bug if everything executes so quickly? If only there was a way to step through the code one line at a time to see what each statement is doing...

### Objectives

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

- Predict the output of a given block of code using the desk check technique.
- Recognize common pitfalls associated with loops.

### Prerequisites

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop (Chapter 2.3).
- Create a loop to solve a simple problem (Chapter 2.3).
- Use `#ifdef` to create debug code in order to test a function (Chapter 2.1).

## Desk Check

Please watch the following videos:

- [Predicting Output](#): The purpose of this video is to illustrate the importance of being able to predict the output of code by inspection.
- [Levels of Understanding](#): The purpose of this video is to illustrate different levels of understanding of an algorithm and the benefits of working at each level. There are three levels:
  1. **Concrete**: A low level understanding of the specific details of what the code is doing. It consists of a description of what happens at every step of the program execution.
  2. **Abstract**: Describes what parts of a program do, and how they relate to the larger whole.
  3. **Conceptual**: A high level comprehension that enables the programmer to explain in simple English what a program does.
- [Dataflow](#): The purpose of this video is to illustrate that tracking dataflow is the most effective way to predict the output of a program.
- [Desk Check Steps](#): The process of desk checking a program, from start to finish. It will describe how desk checking is a form of dataflow measurement, how the complete state of the program is captured at each level, and how every aspect of the algorithm is captured in the end. This chapter will describe the steps of performing a desk check, including line numbering, variable enumeration, and finally building the desk check table.
- [Desk Check Table](#): The purpose of this video is to illustrate how to build and interpret a desk check table on a single-function problem.
- [Desk Check with Functions](#): The purpose of this video is to illustrate how desk checking works across multiple functions.
- [Online Desk Check](#): The purpose of this video is to illustrate how to desk check existing code.



## Example 2.4 – Expression Desk Check

Desk check the following code:

```
{
    // get paid!
    double income = 125.37;

    // remove tithing, you keep 90%
    income *= 0.9;

    // don't forget half goes to savings
    income /= 2.0;

    // a man has got to eat
    income -= 15.50;

    // display the results
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "$" << income << endl; }
```

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
{
    double income = 125.37;           // (1) extra code removed for brevity
    income *= 0.9;                    // (2)
    income /= 2.0;                    // (3)
    income -= 15.50;                  // (4)
    cout << income << endl;           // (5)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution.

| Line | income  | output  |
|------|---------|---------|
| 1    | 125.37  |         |
| 2    | 112.833 |         |
| 3    | 56.4165 |         |
| 4    | 40.9165 |         |
| 5    | 40.9165 | \$40.92 |

Notice how we are able to track each step of execution of the code with the Desk Check table. Even though income changed many times in a few lines of code, we can always see the value at a given moment in time.

## Example 2.4 – Conditional Desk Check

Desk check the following code:

```
int computeTax(double income)
{
    // 10%, 15%, 25%, 28%, and 33% brackets here.
    if (income < 0.00)
        return 0;
    else if (income <= 15100.00)
        return 10;
    else if (income <= 61300.00)
        return 15;
    else if (income <= 123700.00)
        return 25;
    else if (income <= 188450.00)
        return 28;
    else if (income <= 336550.00)
        return 33;

    return 35;
}
```

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
int computeTax(double income)
{
    if (income < 0.00)                // (1)
        return 0;                    // (2)
    else if (income <= 15100.00)      // (3)
        return 10;                   // (4)
    else if (income <= 61300.00)      // (5)
        return 15;                   // (6)
    else if (income <= 123700.00)     // (7)
        return 25;                   // (8)
    else if (income <= 188450.00)     // (9)
        return 28;                   // (10)
    else if (income <= 336550.00)     // (11)
        return 33;

    return 35;                        // (12)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution. We will start with income set to \$50,000.00

| Line | income  | return |
|------|---------|--------|
| 1    | 50000.0 |        |
| 3    | 50000.0 |        |
| 5    | 50000.0 |        |
| 6    | 50000.0 | 15     |

Note how line (2) is not executed because the Boolean expression in line (1) evaluated to `false`. This means that we move to line (3). Since it too evaluated to `false`, we move on to line (5). Now since `(50000.0 <= 61300.00)` evaluates to `true`, we move on to line (6). Line (6) is a return statement, meaning it is the last line of the function we execute.

It is important to realize that line (2), (4), (7), (8), (9), (10), (11), and (12) are never executed.



## Example 2.4 – Loop Desk Check

Desk check the following code:

```
{
    int i;
    int j = 0;

    for (i = 1; i < 10; i *= 2)
        j += i;

    cout << j << endl;
}
```

The first step is to number the lines of code. These line numbers will then correspond to the rows in the Desk Check table. For convenience, the lines numbers are displayed in the line comments.

```
{
    int i;
    int j = 0;                // (1) along with the preceding line

    for (i = 1; i < 10; i *= 2) // (2)
        j += i;              // (3)

    cout << j << endl;       // (4)
}
```

Next, a table will be created to reflect the state of the variables at various stages of execution.

| Line | i  | j  |
|------|----|----|
| 1    | ?  | 0  |
| 2    | 1  | 0  |
| 3    | 1  | 1  |
| 2    | 2  | 1  |
| 3    | 2  | 3  |
| 2    | 4  | 3  |
| 3    | 4  | 7  |
| 2    | 8  | 7  |
| 3    | 8  | 15 |
| 2    | 16 | 15 |
| 4    | 16 | 15 |

Observe how the desk-check is a record of the execution of the loop. You can always “look back in time” to see exactly what was happening at a given stage in execution. For example, the third execution of the loop (highlighted) occurred with *i* set to 4 and *j* set to 3. Since *i* < 10 evaluated to true (because 4 is less than 10), the loop continued on to the body (step 3). From here, *i* remained unchanged, but *j* increased its value by 4. You can always read the current values of all the variables off the desk check table. As expected, the value of *j* jumped from 3 to 7 on this line of code

Notice how the FOR loop has three components: the Initialization, the Boolean expression, and the Increment. As a challenge, create a desk check where step 2 is split into these three components:

2a: Initialization

2b: Boolean expression

2c: Increment

## Example 2.4 – Online Desk Check

### Problem

Modify the following code to perform an online desk check.

```
{
    int i;
    int j = 0;

    for (i = 1; i < 10; i *= 2)
        j += i;

    cout << j << endl;
}
```

The first step is to insert a COUT statement labeling the variables that will be displayed in the various columns. When this is finished, COUT statements are inserted where the line numbers would be on the paper desk check. The resulting code is:

```
{
    int i = 99; // some value
    int j = 0;
    cout << "\ti\tj\n";                                // row header of Desk Check table

    cout << "1\t" << i << "\t" << j << endl;           // (1)

    for (i = 1; i < 10; i *= 2)
    {
        cout << "2\t" << i << "\t" << j << endl; // (2)
        j += i;
        cout << "3\t" << i << "\t" << j << endl; // (3)
    }

    cout << "4\t" << i << "\t" << j << endl;           // (4)
}
```

The output of this modified code should appear “very similar” to the desk check performed by hand:

|   | i  | j  |
|---|----|----|
| 1 | 99 | 0  |
| 2 | 1  | 0  |
| 3 | 1  | 1  |
| 2 | 2  | 1  |
| 3 | 2  | 3  |
| 2 | 4  | 3  |
| 3 | 4  | 7  |
| 2 | 8  | 7  |
| 3 | 8  | 15 |
| 4 | 16 | 15 |

Observe how this table is the same as the paper desk check output.

### Challenge

As a challenge, perform a paper and online desk check on the `computeTax()` function from Project 1. What kind of bugs would it help you find?

### See Also

The complete solution is available at [2-4-deskCheck.cpp](#) or:

```
/home/cs124/examples/2-4-deskCheck.cpp
```

## Pitfalls

As with the pitfalls associated with IF statements, a few pitfalls are common among loops.

### Pitfall: = instead of ==

Remember that '=' means assign, and '==' means compare. We almost always want '==' in loops:

```
{
    bool done = false;

    do
    {
        ...
        if (x == 0)
            done = true;
    }
    while (done = false); // PITFALL! We probably want to compare done with false!
}
```

### Pitfall: < instead of <=

Pay special attention to the problem you are trying to solve. Some loops require us to add “the numbers less than 100.” This implies `count < 100`. Other loops require us to count “from 1 to 10.” This implies `count <= 10`. This class of errors is called “off-by-one” errors:

```
{
    // count from 1 to 10
    for (int count = 1;
        count < 10;    // PITFALL! The comment says 1 to 10 implying count <= 10
        count++)
    ;
}
```

### Pitfall: Extra semicolon

The entire loop statement includes both the loop itself and the body. This means we do not put a semicolon on the FOR loop itself. If we do so, we are implying that there is no body of the loop (just like an IF statement):

```
{
    // count from 1 to 10
    for (int i = 1; i <= 10; i++); // PITFALL! This signifies that there is no body
    cout << i << endl;           // so this statement isn't part of the loop
}
```

### Pitfall: Infinite loop

Please make sure that your loop will end eventually:

```
{
    // count from 1 to 10
    for (int i = 1; i > 0; i++) // PITFALL! I will always be greater than 0!
        cout << i << endl;
}
```

### Problem 1

Which of the following is the definition of the conceptual level of understanding of an algorithm?

- What the program does, not how the solution is achieved
- What the components do and how they influence the program
- The value of every variable at every stage of execution
- Realization where the flaws or bugs are

*Please see page 168 for a hint.*

### Problem 2

Where do you put the line numbers in a desk check table?

Answer:

---

*Please see page 171 for a hint.*

### Problem 3

Given a program that converts feet to meters, create a desk check table for the input value of 2 feet.

```
convert
  PROMPT for feet
  GET feet
  SET meters = feet * 0.301
  PUT meters
END
```

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*Please see page 171 for a hint.*

### Problem 4

Desk check the following program.

```
addNumbers
  SET number = 1
  DOWHILE number ≤ 5
    SET number = number + number
  ENDDO
END
```

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*Please see page 171 for a hint.*

### Problem 5

What is the output?

```
{
    int i;
    for (i = 0; i < 4; i++)
        ;
    cout << "i == " << i;
}
```

Answer:

---

*Please see page 161 for a hint.*

### Problem 6

What is the output?

```
{
    bool done = false;
    int n = 5;

    while (!done)
    {
        if (n == 2)
            done = true;
        n--;
    }
    cout << "n == " << n << endl;
}
```

Answer:

---

*Please see page 157 for a hint.*

### Problem 7

What is the output for the input of 'a' and 'x'?

```
{
    char input;

    do
    {
        cout << "input: ";
        cin >> input;

        cout << "\t"
              << input
              << endl;
    }
    while (input != 'x');
}
```

Answer:

---

*Please see page 157 for a hint.*

### Problem 8

What is the output?

```
{
    int i;

    for (i = 0; i < 4; i++);
        cout << "H";
    cout << endl;
}
```

Answer:

---

*Please see page 173 for a hint.*

### Problem 9

What is the output?

```
{
    int i;

    for (i = 0; i <= 4; i++)
        ;

    cout << "i == " << i << endl;
}
```

Answer:

---

*Please see page 157 for a hint.*

### Problem 10

What is the output?

```
{
    int sum = 0;
    int count;

    for (count = 0;
        count < 4;
        count++)
        sum += count;

    cout << "sum == " << sum;
}
```

Answer:

---

*Please see page 161 for a hint.*

## Problem 11

What is the output?

```
{
    bool done    = false;
    int  number = 1;

    while (!done)
    {
        cout << number << endl;
        number *= 2;

        if (number > 4)
            done = true;
    }
}
```

Answer:

---

*Please see page 171 for a hint.*

## Problem 12

What is the output?

```
{
    int sum = 0;
    int count;

    for ( count = 1;
          count < 9;
          count *= 2)
        sum += count;

    cout << "sum == " << sum;
}
```

Answer:

---

*Please see page 171 for a hint.*

## Problem 13

What is the output?

```
{
    int count = 0;

    while (count < 5)
        count++;

    cout << "count == " << count;
}
```

Answer:

---

*Please see page 157 for a hint.*

### Problem 14

What is the output?

```
{  
    int count = 10;  
  
    while (count < 5)  
        count++;  
  
    cout << "count == " << count;  
}
```

Answer:

---

*Please see page 157 for a hint.*

### Problem 15

What is the output?

```
{  
    int count = 0;  
  
    do  
        count++;  
    while (count < 3);  
  
    cout << "count == " << count;  
}
```

Answer:

---

*Please see page 159 for a hint.*



## Unit 2

### Problem 1: Convert grade

A full page of blank graph paper with a uniform grid of small squares. The grid consists of 20 columns and 20 rows, creating a total of 400 squares. The lines are thin and black, set against a white background. There are no margins or additional markings on the page.

The user input is 2, 0, 10 in the following code:

[illegible]

---

|                               |                        |                 |                 |
|-------------------------------|------------------------|-----------------|-----------------|
| Procedural Programming in C++ | Unit 2: Design & Loops | 2.4 Loop Output | <b>Page 179</b> |
|-------------------------------|------------------------|-----------------|-----------------|

[illegible]

# Unit 2

## 2.5 Loop Design

Sam is upset because he is trying to find a copy of the ASCII Table containing both hexadecimal (base 16) values and decimal (base 10) values. None of his favorite C++ web sites have the right table and Google is turning out to be useless. Rather than stooping to ask the professor, Sam decides to write his own code to display the table. But how to start? What will the main loop look like? (For an example solution of this problem, please see `/home/cs124/examples/2-5-asciitable.cpp`).

### Objectives

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

- Recognize the three main types of loops.
- Use a loop to solve a complex problem.

### Prerequisites

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

- Demonstrate the correct syntax for a WHILE, DO-WHILE, and FOR loop (Chapter 2.3).
- Create a loop to solve a simple problem (Chapter 2.3).

## Three Types of Loops

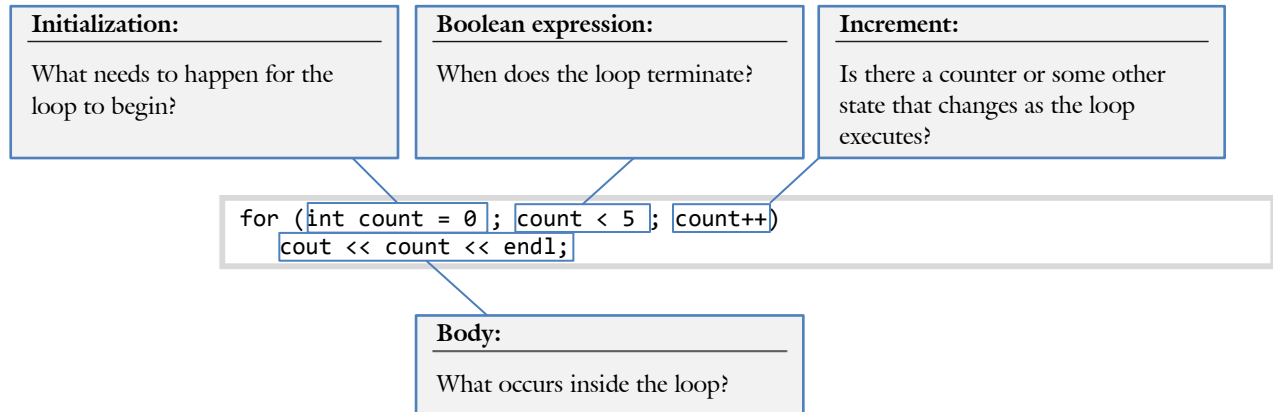
It takes a bit of skill to think in terms of loops. The purpose of this chapter is to give you design practice working with this difficult construct. Almost all looping problems can be broken into one of three categories: counter-controlled, event-controlled, and sentinel controlled. It is usually a good idea to identify which of the three types your problem calls for and design accordingly.

- **Counter Controlled:** Keep iterating a fixed number of times. The test is typically a single integer that changes with each iteration.
- **Event Controlled:** Keep iterating until a given event occurs. The number of iterations is typically unknown until the program runs.
- **Sentinel Controlled:** Keep iterating until a marker (called a sentinel) indicates the loop is done. The test is a given variable (typically a bool) that is set by any one of a number of events.

# Counter-Controlled

A counter-controlled loop is a loop executing a fixed number of times. Typically that number is known before the loop starts execution. If, for example, I were to determine the number of students who had completed the homework assignment, a counter-controlled loop would be the right tool for the job.

In almost all circumstances a `for` statement is used for counter-controlled loops because `for` statements are designed for counting. Observe how the four parts to a `FOR` loop correspond to the four parts of the typical counting problem:




Counter-controlled loops are readily identified by the presence of a single variable that moves through a range of values. In other words, counter-controlled loops do not exclusively increment by one: they might increment by 10 or powers of 3.

When designing with a counter-controlled loop, it is usually helpful to answer the following four questions:

- **How does the loop start?** In other words, what is the beginning of the range of values? This corresponds to the initialization part of the `FOR` loop.
- **How does the loop end?** In other words, by what condition do you know that you have yet to reach the end of the range of values? This corresponds to the Test or controlling Boolean expression part of the `FOR` loop.
- **What do you count by?** In other words, as you move through the range of values, how does the variable change? This corresponds to the Update part of the `FOR` loop.
- **What happens each iteration?** In other words, is some action taken at each step of the counting? Frequently no action is taken so this step can be skipped. If there is an action, then it goes in the Body section of the `FOR` loop.

## Example 2.5 – Counter-Controlled Loop

|           |  |
|-----------|--|
| Demo      | This example will demonstrate how to design a counter-controlled loop.   |
| Problem   | <p>Sue heard the following story in class one day:</p> <p>When Carl Friedrich Gauss was 6, his schoolmaster, who wanted some peace and quiet, asked the class to add up the numbers 1 to 100. “Class,” he said, coughing slightly, “I’m going to ask you to perform a prodigious feat of arithmetic. I’d like you all to add up all the numbers from 1 to 100, without making any errors.” “You!” he shouted, pointing at little Gauss, “How would you estimate your chances of succeeding at this task?” “Fifty-fifty, sir,” stammered little Gauss, “no more...”</p> <p>What the schoolmaster did not realize was that young Gauss figured out that adding the numbers from 1 to <math>n</math> is the same as: <math>sum = (n + 1)n / 2</math>. Sue wants to write a program to verify this. Her program will both add the numbers one by one, and use Gauss’ equation.</p> |
| Solution  | <p>With counter-controlled loops, four questions need to be answered:</p> <ul style="list-style-type: none"> <li>• <b>Initialization:</b> The loop starts at 1: <code>int count = 1</code></li> <li>• <b>End:</b> The loop stops after the number is reached: <code>count &lt;= n</code></li> <li>• <b>Update:</b> The loop counts by 1’s: <code>count++</code></li> <li>• <b>Body:</b> Every iteration we increase the sum by count: <code>sum += count</code></li> </ul> <p>With these four answers, we can write the function.</p> <pre>int computeLoop(int n) {     int sum = 0;     for (int count = 1; count &lt;= n; count++)         sum += count;      return sum; }</pre>  |
| Challenge | As a challenge, find values where Gauss’s equation does not work. Can you modify the program so it works with all integer values?  |
| See Also  | <p>The complete solution is available at <a href="#">2-5-counterControlled.cpp</a> or:</p> <pre>/home/cs124/examples/2-5-counterControlled.cpp</pre>    |

# Event-Controlled

An event-controlled loop is a loop that continues until a given event occurs. The number of repetitions is typically not known before the program starts. For example, if I were to write a program to prompt the user for her age but would re-prompt if the age was negative, an event-controlled loop is probably the right tool for the job. Typically event-controlled loops use `while` or `do-while` statements, depending if the loop needs to execute at least once.

## Boolean expression:

How do you know when you are done? Event controlled loops keep going until an event has occurred. In this case, until the GPA is within an acceptable range.

```
while (gpa > 4.0 || gpa < 0.0)
{
    cout << "Enter your GPA: ";
    cin >> gpa;
}
```

## Body:

What changes every iteration? Typically something happens during event-controlled loops. This may be a re-prompt or a value is updated some other way. The programmer cannot predict how many iterations will be required; it depends on execution.

When designing with an event-controlled loop, it is usually helpful to answer the following two questions:

- **How do you know when you are done?** In other words, what is the termination condition? This condition maps directly to the controlling Boolean expression of a `WHILE` or `DO-WHILE` loop.
- **What changes every iteration?** Unlike counter-controlled loops where the counter changes in a predictable way, event controlled loops are typically driven by an external event. Usually there needs to be code in an event-controlled loop to re-query this external event. Often this comes in the form of re-prompting the user for input or reading more data from a file.

## Sam's Corner



Notice how the two parts to an event-controlled loop (ending condition and what changes every iteration) look a lot like the two of the four parts of a counter-controlled loop (start, ending condition, counting, and what happens every iteration). This is because an event-controlled loop is a sub-set of a counter-controlled loop. If the counter-controlled loop does not have a starting condition or a counting component, then it probably is an event-controlled loop.

All `FOR` loops can be converted into a `WHILE` loop.

```
for ( i = 0;    // initialize
      i < 10;   // condition
      i++)     // increment
    cout << i << endl;
```

```
i = 0;          // initialize
while (i < 10)   // condition
{
    cout << i << endl;
    i++;        // increment
}
```

All `WHILE` loops can be converted into a `FOR` loop.

```
while (grade < 70.0)
    grade = takeClassAgain();
```

```
for (; grade < 70.0; )
    grade = takeClassAgain();
```

## Example 2.5 – Event-Controlled Loop

Demo

This example will demonstrate how to design an event-controlled loop.

Problem

Write a program to keep prompting the user for his GPA until a valid number is entered.

```
Please enter your GPA (0.0 <= gpa <= 4.0): 4.1
Please enter your GPA (0.0 <= gpa <= 4.0): -0.1
Please enter your GPA (0.0 <= gpa <= 4.0): 3.9
GPA: 3.9
```

Solution

With event-controlled loops, two questions need to be answered:

- **End condition:** The loop ends when the condition is reached  $0 \leq \text{gpa} \leq 4.0$ .
- **Update:** Every iteration, we re-prompt the user for the GPA.

With these two answers, we can write our pseudocode:

```
WHILE gpa > 4.0 or gpa < 0.0
  PROMPT for gpa
  GET gpa
```

With this pseudocode, it is straight-forward to write the function:

```
float getGPA()
{
    float gpa = -1.0; // any value outside the expected range will do

    // loop until a valid value is received
    while (gpa > 4.0 || gpa < 0.0)
    {
        cout << "Please enter your GPA (0.0 <= gpa <= 4.0): ";
        cin >> gpa;
    }

    // return with the loot
    assert(gpa <= 4.0 && gpa >= 0.0); // paranoia will destroy-ya
    return gpa;
}
```

Observe how much more complex the C++ for `getGPA()` is than the pseudocode. This is because variable initialization, comments, the text of the prompt, and asserts are not required for pseudocode

Challenge

As a challenge, try to change the above function from a WHILE loop to a DO-WHILE loop. Both loops are commonly event-controlled.

See Also

The complete solution is available at [2-5-eventControlled.cpp](#) or:

```
/home/cs124/examples/2-5-eventControlled.cpp
```



## Sentinel-Controlled

A sentinel is a guardian or gatekeeper. In the case of a sentinel-controlled loop, the sentinel is a variable used to determine if a loop is to continue executing or if it is to terminate. We typically use a sentinel when multiple events could cause the loop to terminate. In these cases, the sentinel could be changed by any of the events. Typically event-controlled loops use `while` or `do-while` statements where the sentinel is a `bool` used as the condition. Perhaps, this is best explained by example.

### Example 2.5 – Sentinel-Controlled Loop

Demo

This example will demonstrate how to design a sentinel-controlled loop.

Problem

Consider a professor trying to determine if a student has passed his class. There are many criteria to be taken into account (the grade and whether he cheated, to name a few). Rather than making a single highly-complex controlling Boolean expression, he decides to use a sentinel-controlled loop:

```
Welcome to CS 124!
What is your class grade? 109
Did you cheat in the class? (y/n) y

Welcome to CS 124!
What is your class grade? 81
Did you cheat in the class? (y/n) n
Great job! Get ready for CS 165
```

Solution

The solution is to have one variable (passed) be set by a variety of conditions.

```
{
    bool passed = false;                // the sentinel. Initially we have
                                        //      not passed the class

    // the main loop
    while (!passed)                     // common sentinel, read
    {                                   //      "while not passed..."

        cout << "\nWelcome to CS 124!\n";

        // if you got a C or better, you may have passed...
        float grade;
        cout << "What is your class grade? ";
        cin >> grade;
        if (grade >= 60.0)
            passed = true;              // one of the ways the
                                        //      sentinel may change

        // if you cheated, you did not pass
        char cheated;
        cout << "Did you cheat in the class? (y/n) ";
        cin >> cheated;
        if (cheated == 'y' || cheated == 'Y')
            passed = false;             // another sentinel condition
    }
    cout << "Great job! Get ready for CS 165\n";
}
```

See Also

The complete solution is available at [2-5-sentinelControlled.cpp](#) or:

```
/home/cs124/examples/2-5-sentinelControlled.cpp
```





## Problem 1

Write a program to put the alphabet on the screen. A.K.A. Sesame Street Karaoke.

A  
B  
C  
...  
Y  
Z

Answer:

*Please see page 182 for a hint.*

## Problem 2

Write a program to display the multiplication table of numbers less than 6.

|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 2  | 3  | 4  | 5  |
| 2 | 4  | 6  | 8  | 10 |
| 3 | 6  | 9  | 12 | 15 |
| 4 | 8  | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |

Answer:

*Please see page 182 for a hint.*

### Problem 3

Write a program to compute how many numbers under a user-specified value are both odd and a multiple of 5.

What is the number: 20

The number of values under 20 that are both odd and a multiple of 5 are: 2

Answer:

*Please see page 182 for a hint.*

### Problem 4

Write a function to compute whether a number is prime:

```
bool isPrime(int number);
```

Answer:

*Please see page 49 for a hint.*

## Assignment 2.5

Write a function (`displayTable()`) to display a calendar on the screen. The function will take two parameters:

- `numDays`: The number of days in a month.
- `offset`: The offset from Monday. If the offset is zero, then the month starts on Monday. If the offset is 2, the month starts on Wednesday. If the offset is 6, the month starts on Sunday.

This function will be “very similar” to the `displayTable()` function in Project 2. Please see the project for details on the spacing between the columns and a hint on how the algorithm might work. Next write `main()` so that it prompts the user for the number of days in the month and the offset.

Note that this is probably the most difficult assignment of the semester. Of course, this will also get you very far along on the project as well. Please, allocate a couple hours for this assignment.

### Example

Three examples. The user input is underlined.

Number of days: 30

Offset: 3

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
|    |    |    |    | 1  | 2  | 3  |
| 4  | 5  | 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |    |

Number of days: 28

Offset: 0

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
|    | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 |    |    |    |    |    |    |

Number of days: 31

Offset: 6

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| 8  | 9  | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 |    |    |    |    |

### Assignment

The test bed is available at:

```
testBed cs124/assign25 assignment25.cpp
```

Don't forget to submit your assignment with the name “Assignment 25” in the header,

*Please see page 182 for a hint.*

## 2.6 Files

Sue is home for the Christmas holiday when her mother asks her to fix a “computer problem.” It turns out that the problem is not the computer itself, but some data their bank has sent them. Instead of e-mailing a list of stock prices in US dollars (\$), the entire list is in Euros (€)! Rather than perform the conversion by hand, Sue decides to write a program to do the conversion. This is done by opening the file with the list of Euro prices, performing the conversion to US dollars, and writing the resulting values to another file.

### Objectives

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

- Write the code to read data from a file.
- Write the code to write data to a file.
- Perform error checking on file streams.
- Understand the different ways the end-of-file marker can be found.

### Prerequisites

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

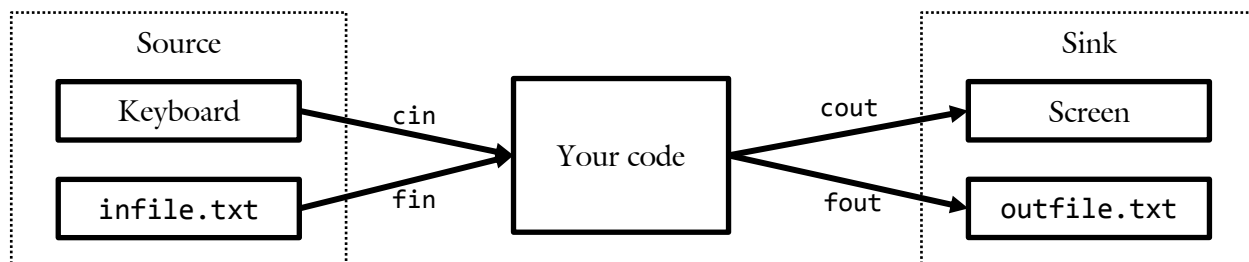
- Create a loop to solve a complex problem (Chapter 2.5).

## Overview

After a program ends, all memory of its execution is removed. This fact is particularly unsatisfactory if the program was charged with maintaining the user’s valuable data. To overcome this shortcoming, it is necessary to save to and retrieve data from a file.

In many ways, writing data to a file is similar to writing data to the screen. In the file case, however, one needs to specify the target file instead of just using `cout`. In other words, `cout << number << endl;` would put the value of the variable `number` on the screen. If, on the other hand, the variable `fout` corresponded to a file, one could put the value of `number` in the file with `fout << number << endl;`.

Similarly, reading data from a file is similar to accepting user input from a keyboard. Here again, the programmer needs to specify the name of the source file. There is one additional difference, however. When reading data from the keyboard, the programmer can assume there is an infinite amount of data on hand (assuming an infinitely patient user!). Files, on the other hand, are of finite length. At some point, the end will be reached and the program needs to be ready to handle that event.



## Writing to a File

When we write text to the screen, we use `cout`. This variable is defined in the `iostream` library and keeps track of where the cursor is on the screen. In other words, as we continue to send data to the screen, the cursor keeps moving to the right or down much the same way a typewriter advances. Sending data to a file is conceptually the same; we need a variable to keep track of the location of the cursor in the file so, as more data is sent to the file, the cursor advances. It follows that we would need a variable very similar to `cout` to do this. However, there is one key difference between writing to a file and writing to the screen: there is only one screen to write to while there may be many files. Therefore, it is necessary to also specify *which* file we are writing to. Consider the following code:

```
#include <fstream>

void writeFile()
{
    // declare the output stream
    ofstream fout;

    // open the file
    fout.open("greeting.txt");

    // write some text
    fout << "Hello world\n";

    // close the stream
    fout.close();
    return;
}
```

**Include the FSTREAM library:**  
 All the functions needed to read and write to a file are included in the `fstream` library. It must be included just like `iostream`.

**Declare a stream variable:**  
 You must declare a variable associated with the file. Since this is an output stream (writing to a file), use `ofstream`.

**Open the stream**  
 This will associate the variable with the file

**Insertion operator <<**  
 Write to the file as you would with `cout`

**Close the stream**  
 When finished, indicate you are done with the `close()` function

Simple file writing, in other words, consists of several components: the `fstream` library, declaring a stream variable (`fout`), opening the file, streaming data to the file, and closing it when finished.

### Sam's Corner

We have previously mentioned that `cout` is a variable. While this is true, it is a special type of variable: an object. An object is a variable that contains both data (position of the cursor on the screen) and functions (think `cout.setw(5)` and `cout.getline(text, 256)`) associated with the data. Objects and the classes that define them are subjects of Object Oriented programming, the topic of CS 165. Don't worry about objects now; this is a topic for next semester.



## FSTREAM library

When we were writing programs to display text on the screen, we needed to use the `iostream` library. This library defined two variables (`cout` and `cin`) and the functions associated with them. When writing to a file, we use the `fstream` library. This library contains two data types: `ofstream` and `ifstream`. **OFSTREAM** stands for Output File STREAM used to send data out of a program and into a file. This is done with:

```
#include <fstream> // need this line every time a file is used in the program
```

## Declaring a stream variable

The next step is to declare a variable that will be used to send data to the file. Note that you may use any variable name you like (as long as it conforms to the Elements of Style guidelines). As a rule, you might want to use `fout` for a stream variable name because it looks and feels like “cout.” The only difference is that `F` stands for File while `c` stands for Console (or screen). This is done with:

```
ofstream fout;           // declare an output file stream variable.
```

It is possible to have more than one output file stream active at once. A large and advanced program might, for example, write one file for user data, another for a log, and a third to save configuration data. This is not a problem; just have three `ofstream` variables:

```
{
    ofstream foutData;    // For user data
    ofstream foutLog;     // For a log
    ofstream foutConfig;  // For config. data. All 3 can be used at the same time
}
```

## Opening a file

After a stream variable is declared, the next step is to connect that variable to a given file. This can be done with a “hard-coded” string (a string literal)...

```
fout.open("data.txt");    // Always use the same file. We seldom do this.
```

... or it could be done with a variable:

```
{
    char fileName[256];    // string variable to hold the name of the file
    cin >> fileName;       // prompt user for the file name
    fout.open(fileName);   // open the file by referencing the variable.
}
```

It is also possible to both declare a stream variable and associate it with a file in one step:

```
{
    ofstream fout;
    fout.open(fileName);
}
```

```
{
    ofstream fout(fileName);
}
```

Both of the above lines of code do exactly the same thing; it is a matter of convenience which you choose. The final thing to note about opening a file is that, on occasion, there is no file to open. In other words, what would happen when the user attempts to write to a directory that does not exist, to a thumb drive that is full, or to a file where the user lacks the required permission? In each of these cases, an error message will need to be presented to the user. Thus, it is absolutely necessary to check for the error condition and quit the file writing process. Consider the following function:

```

/*****
 * Write GPA
 * This function will write the user's GPA to a file
 *****/
bool writeGPA(char fileName[], float gpa)
{
    // declare and open the stream
    ofstream fout(fileName);           // declare and initialize in one step
    if (fout.fail())                   // check for error with fail()
        return false;                  // indicate to caller that we failed!

    // write the data
    fout << gpa << endl;               // actually do the work

    // quite
    fout.close();                      // don't forget to close when done
    return true;                       // report success to the caller
}

```

This function takes the filename as a parameter. To call the function, it is necessary to specify a string as the first parameter.

```

{
    writeGPA("myGrade.txt", 3.9);      // first parameter must be a string
}

```

Note how we check for error with the `fail()` function. If we are unable to open the file for writing for any reason (permissions, lack of space, general hardware error, etc.), then `fail()` will return `true`. This will mean that the function `writeGPA()` will be unable to do what it was asked to do: write to a file. We therefore commonly make file functions return Boolean values: `true` corresponds to success and `false` corresponds to failure.

### Sam's Corner



By default, `OFSTREAM` will replace any file of the same name that is being written. This might be what the programmer intended if there is no other file or if data is to be updated. However, it is often necessary to append data onto the end of a file rather than replace it. This can be done by adding another parameter to the output stream declaration:

```
ofstream fout(fileName, ios::app);
```

In this case, the `ios::app` means to append the file rather than overwrite. Other modes include.

| Mode                     | Meaning  |
|--------------------------|--|
| <code>ios::app</code>    | Append output to end of file (EOF)   |
| <code>ios::ate</code>    | Seek to EOF when the file is opened  |
| <code>ios::binary</code> | Open file in binary mode   |
| <code>ios::in</code>     | Open file for reading. This happens automatically for <code>ifstream</code>      |
| <code>ios::out</code>    | Open file for writing. This happens automatically for <code>ofstream</code>      |
| <code>ios::trunc</code>  | Overwrite existing file instead of truncating. Default for <code>ofstream</code> |

## Streaming data to a file

We use `fout` to send data to a file in exactly the same way we use `cout` to send data to the screen. This means that all tools we had for screen display we also have for file writing. Consider the following code:

```
{
    // configure FOUT for displaying money, just like COUT
    fout.setf(ios::fixed);
    fout.setf(ios::showpoint);
    fout.precision(2);

    // display my budget
    fout << "\t$" << setw(9) << income << endl;
    fout << "\t$" << setw(9) << spending << endl;
    fout << "\t ----- \n";
    fout << "\t$" << setw(9) << income - spending << endl;
}
```

The above code might be very familiar if `cout` were used instead of `fout`. The only real difference is that this data is sent to a file rather than the screen



### Sue's Tips

While it is easy to verify the screen output of a program, it is often inconvenient to verify the file output. As a result, beginner programmers tend to forget details such as putting spaces between numbers. One easy way to work around this tendency is to first write your `writeFile()` function with `couts`. After you have run the program a few times and you are sure of the formatting, turn your `couts` in to `fouts` to send the same data to the file.

## Closing the file

When we are finished writing data to a file, it is important to remember to close the file. On primitive operating systems (think [MS-DOS](#)), an un-closed file could never be reopened. Modern operating systems, however, will handle this step for you if you forget. However, it is “good form” to close a file as soon as the last data has been written to it:

```
fout.close();
```



## Reading from a File

Just as writing text to a file with `fout` is similar to writing text to the screen with `cout`, reading text from a file has a `cin` equivalent: `fin`. There is, however, one important difference between reading text from the keyboard and reading text from a file. Eventually the end of the file will be reached. It is therefore necessary to make sure logic exists in the program to handle the unexpected end-of-file condition.

As with writing data to a file, several steps are involved: using the `FSTREAM` library (`#include <fstream>`), declaring the input file stream variable (`ifstream fin;`), checking for errors (`fin.fail()`), using the extraction operator (`fin >> data;`), and closing the file.

```

#include <fstream>

int readFile()
{
    // declare the output stream
    ifstream fin("number.txt");
    if (fin.fail())
        return -1;

    // read the data
    int data;
    fin >> data;

    // close the stream
    fin.close();
    return data;
}

```

**Include the FSTREAM library:**  
As with writing to a file, the code necessary to read from a file is in `fstream`

**Declare a stream variable:**  
You must declare a variable associated with the file. Since this is an input stream (reading from a file), use `ifstream`

**Check for errors**  
If the file does not exist or you don't have permission to read it, you must handle it

**Extraction operator >>**  
Read from a file just like you would with `cin`

**Close the stream**  
When finished, indicate you are done with the `close()` function

## FSTREAM library

As with writing to a file, it is necessary to remember to include the `FSTREAM` library. If this step is skipped, one can expect the following compile error:

```

example.cpp: In function "int readFile()":
example.cpp:6: error: aggregate "std::ifstream fin" has incomplete type and cannot be defined

```

This cryptic compiler error means that `std::ifstream` is an unknown type. The reason, of course, is that `IFSTREAM` is defined in `fstream`. Therefore, don't forget:

```
#include <fstream>
```

## Declaring a stream variable

Input stream variables are defined in much the same way as output stream variables. The most important difference, of course, is we use `ifstream` for Input File STREAM. Also, like output streams, we can declare and initialize the variable in a single line.

```
{
    ifstream fin;
    fin.open(fileName);
}
```

```
{
    ifstream fin(fileName);
}
```

Again, by convention, it is common to use `fin` for the variable name to emphasize the relationship with `cin`.

## Check for errors

As with writing to a file, an essential part of reading from a file includes checking for errors. The same class of errors for writing to a file exists when reading from a file (no permissions, missing directory, general file-system error, etc.). Additionally, the potential exists that there might not be any data in the file to read. In all these cases, we can detect if an error occurred with the `fin.fail()` function call.

```
{
    ifstream fin(fileName);           // attempt to open the file
    if (fin.fail())                   // check for any type of error
    {
        cout << "Unable to open file " // let the user know!
              << fileName << endl;    // he probably wants to know the file name
        return false;                 // report failure to the user
    }
}
```

## Read the data

We write (to the screen or to a file) using the **insertion operator** (`<<`). Similarly, all read operations are done with the **extraction operator** (`>>`).

```
{
    float temperature;                // first item is expected to be a number
    char units[256];                  // next item is expected to be text
    fin >> temperature >> units;      // read both just like with cin
}
```

There are two ways we can tell if the read failed for any reason. The first is to check for a read failure. This can be accomplished with another `fin.fail()` function call. The second is to see if the extraction operator itself failed. The following two lines of code are equivalent:

```
{
    int value;
    fin >> value;
    if (!fin.fail())
        cout << "Success!\n";
}
```

```
{
    int value;
    if (fin >> value)
        cout << "Success!\n";
}
```

In other words, the extraction operator (`>>`) is actually a function call returning `false` when it fails for any reason. One reason may be that the file has been corrupted (or even erased!) during the read. Another may be that there is no more data in the file. Another way to state this last reason is that the “end-of-file” condition may have been met.

## Reading to the end of the file

At the end of every file in a file system is a special marker indicating that there is no more data in the file. This can be thought of as the “end of road” marker on a highway. We can ask the file stream if we are at the end of file (EOF) with a function call:

```
{
    if (fin.eof())                // returns TRUE if we are at the end
        cout << "There is no more data!\n";
}
```

This means that there are two ways to read all the data from a file. The first is to continue looping until the EOF marker is reached. The second is to read until an error has occurred on the read:

| EOF  | Read Failure   |
|--|--|
| <p>IF the end of the file character is encountered, the EOF flag will be set. You can check for this at any time:</p> <pre>{     ifstream fin("file.txt");      while (!fin.eof())     {         char text[256];         fin &gt;&gt; text;         cout &lt;&lt; text &lt;&lt; endl;     }     fin.close(); }</pre> | <p>If a read failure occurs, the extraction operator will return false. This can be checked on any read.</p> <pre>{     ifstream fin("file.txt");      char text[256];     while (fin &gt;&gt; text)     {         cout &lt;&lt; text &lt;&lt; endl;     }      fin.close(); }</pre> |

These two methods are not the same. Consider the case when there is a word and a space in the file.

|   |   |   |   |       |     |
|---|---|---|---|-------|-----|
| w | o | r | d | space | EOF |
|---|---|---|---|-------|-----|

In the first case, we will read the word on the first loop and display the text on the screen. On the second iteration, we will go into the body of the loop (because we are not yet at the end of the file: there is still a space left!). When we attempt to read the next word with `fin >> text`, we fail (there is no non-space data in the file after all). In this case, we will not change the value of `text` so the word will be repeated on the screen.

In the second case, we will successfully read the word on the first iteration of the loop. This, of course, will be displayed on the screen in the body of the loop. On the second iteration, we will fail to read (there is no non-space data in the file) so the loop will exit. This means the last word will not be repeated on the screen.

For more details on the aforementioned differences between using the EOF method and the read-failure method of reading from a file, please see Example – End of File on the following page.

## Closing the file

As with writing data to a file, it is important to always remember to close the file that was read:

```
fin.close();
```

### Sam's Corner



With most operating systems, the failure of a program to close a file is not catastrophic; the operating system will quietly close it for you once the program exits. This is not true on some mobile platforms or older operating systems. An improperly closed file or a file that has not been closed will remain locked and forever unavailable for use. Thus, it is good form to always close a file as soon as reading or writing has been completed.

## Filename

There is one final complication that arises when working with files: the necessity of dealing with filenames. Filenames are c-strings, something we learned about in Chapter 1.2 (page 38) but have done very little with since. The reason for this is that handling c-strings is a bit quirky. We cannot return a c-string from a function as we would any other data-type. Instead, we need to pass it as a parameter.

When passing a c-string, or any other array (which we will learn about in Unit 3), it comes in as pass-by-reference even though we don't have use the '&' operator. Thus the correct way to write a function to prompt the user for a filename is:

```
/******  
 * GET FILENAME  
 * Prompt the user for a filename.  
 *****/  
void getFilename(char fileName[])           // the fileName parameter behaves  
{                                           // like pass-by-reference  
    cout << "What is the name of the file? ";  
    cin >> fileName;                       // text entered here will be sent  
                                           // back to the caller  
}
```

If there are some things about this function that you don't understand, don't worry! We will learn more about this on page 245.

## Example 2.6 – End of File

Demo

This example will demonstrate how to read all the content of the file using two techniques: either using the EOF method or the Read Failure method.

Problem

Write a program to read all the text out of a file and display the results on the screen. Consider, for example, the following text in a file in 2-6-eof.txt:

```
I love software development!
```

The output is:

```
Filename? 2-6-eof.txt  
Use the EOF method? (y/n): y  
'I' 'love' 'software' 'development!' 'development!'
```

Solution

The first solution is to use EOF method.

```
void usingEOF(const char filename[])  
{  
    // open  
    ifstream fin(filename);  
    if (fin.fail())  
    {  
        cout << "Unable to open file " << filename << endl;  
        return;  
    }  
  
    // get the data and display on the screen  
    char text[256];  
    // keep reading as long as:  
    // 1. not at the end of file  
    while (!fin.eof())  
    {  
        // note that if this fails to read anything (such as when there  
        // is nothing but a white space between the file pointer and the  
        // end of the file), then text will keep the same value as the  
        // previous execution  
        fin >> text;  
        cout << "\"" << text << " " << endl;  
    }  
    cout << endl;  
  
    // done  
    fin.close();  
}
```

The second solution uses the Read Failure method. Everything is the same except the loop:

```
// keep reading as long as:  
// 1. not at the end of file  
// 2. did not fail to read text into our variable  
// 3. there is nothing else wrong with the file  
while (fin >> text)  
    cout << "\"" << text << " " << endl;
```

See Also

The complete solution is available at [2-6-eof.cpp](#) or:

```
/home/cs124/examples/2-6-eof.cpp
```



## Example 2.6 – Read Data

Demo

This example will demonstrate how to read a small amount of data from a file. This will include two data types (a string and an integer). All error checking will be performed.

Problem

Consider the following file (2-6-readData.txt):

Sue 19

Read the file and display the results on the screen.

What is the filename? **2-6-readData.txt**  
The user Sue is 19 years old

The main work is performed by the read() function, taking a filename as a parameter.

```
bool read(char fileName[])           // filename we will read from
{
    // open the file for reading
    ifstream fin(fileName);           // connect to fileName
    if (fin.fail())                   // never forget to check for errors
    {
        cout << "Unable to open file " // tell the user what happened
              << fileName << endl;
        return false;                 // return and report
    }

    // do the work
    char userName[256];
    int  userAge;
    fin >> userName >> userAge;        // get two pieces of data at once
    if (fin.fail())
    {
        cout << "Unable to read name and age from "
              << fileName << endl;
        return false;
    }

    // user-friendly display
    cout << "The user "                // display the data
         << userName
         << " is "
         << userAge << " years old\n";

    // all done
    fin.close();                       // don't forget to close the file
    return true;                       // return and report
}
```

Challenge

As a challenge, can you change the above program to accommodate the user's GPA. This will mean that there are three items in the file:

Sue 19 3.95


See Also

The complete solution is available at [2-6-readData.cpp](#) or:

/home/cs124/examples/2-6-readData.cpp



## Example 2.6 – Read List

|           |  |
|-----------|--|
| Demo      | This example will demonstrate how to read large amounts of data from a file. In this case, the file consists of a list of numbers. The program does not know the size of the list at compile time.   |
| Problem   | <p>Write a program to sum all the numbers in a file. The numbers are in the following format:</p> <pre>34 25 10 43</pre> <p>The program will prompt the user for the filename and display the sum:</p> <pre>What is the filename: <u>2-6-readList.txt</u> The sum is: 112</pre>  |
| Solution  | <p>The function <code>sumData()</code> does all the work in this example. It is important to note that the program does not need to remember all the files read from the file. Once the value is added to the <code>sum</code> variable, then it can be ignored.</p> <pre>int sumData(char fileName[]) {     // open the file     ifstream fin(fileName);     if (fin.fail())         return 0;                // some error message      // read the data     int data;                    // always need a variable to store the data     int sum = 0;                 // don't forget to initialize the variable     while (fin &gt;&gt; data)         // read: "while there was not an error"         sum += data;            // do the work      // close the stream     fin.close();     return sum; }</pre> |
| Challenge | See if you can modify the above program (and the file that it reads from) to work with floating point numbers.   |
| See Also  | <p>The complete solution is available at <a href="#">2-6-readList.cpp</a> or:</p> <pre>/home/cs124/examples/2-6-readList.cpp</pre>    |

## Example 2.6 – Round Trip

### Demo

A common scenario is to save data to a file then read the data back again next time the program is run. We call this “round-trip” because the data is preserved through a write/read cycle. This program will demonstrate that process.

### Problem

Consider a file consisting of a single floating point number:

30.36

Write a program to read the file, prompt the user for a value to add to the value, and write the updated value back to the file.

Account balance: \$30.36  
Change: \$5.20  
New balance: \$35.56

First, the program will read the balance from a file. If no balance is found, then return 0.0:

```
float getBalance()
{
    // open the file
    ifstream fin(FILENAME);           // the filename is constant because
    if (fin.fail())                   // it needs to be same every time
        return 0.0;                  // if no file found, start at $0.00

    // read the data
    float value = 0.0;
    fin >> value;                     // read the old value
    if (fin.fail())                   // if we cannot read this value for any
        return 0.0;                  // reason, return $0.00

    // close and return the data
    fin.close();
    return value;                     // send the value back to main()
}
```

Then, after the user has been prompted for a new value and the balance has been updated, the new balance is written to the same file.

```
void writeBalance(float balance)
{
    // open the file for writing
    ofstream fout(FILENAME);          // make sure it is the same file as
    ...                               // we used for getBalance()

    // write the data
    fout.precision(2);                // format fout for money just like we
    fout.setf(ios::fixed);             // would do for cout.
    fout.setf(ios::showpoint);
    fout << balance << endl;          // it is “good form” to end with endl
    ...
}
```

### See Also

The complete solution is available at [2-6-roundTrip.cpp](#) or:

/home/cs124/examples/2-6-roundTrip.cpp





## Problem 1

What is in the file "data.txt"?

```
void writeData(int n)
{
    ofstream fout;
    fout.open("data.txt");

    for (int i = 0; i < n; i++)
        fout << i * 2 << endl;

    fout.close();
    return;
}

int main()
{
    writeData(4);

    return 0;
}
```

Answer:

*Please see page 193 for a hint.*

## Problem 2

Given the following function:

```
bool writeFile(char fileName[])
{
    ofstream fout;
    fout.open(fileName);

    fout << "Hello World!\n";

    fout.close();

    return true;
}
```

Which would **not** cause the program to fail?

*Please see page 193 for a hint.*

### Problem 3

Write a function to put the numbers 1-10 in a file:

- Call the function numbers()
- Pass the file name in as a parameter
- Do error checking

Answer:

*Please see page 193 for a hint.*

### Problem 4

What is the best name for this function?

```
void mystery(char f1[], char f2[])
{
    ofstream fout;
    ifstream fin;

    fin.open(f1);
    fout.open(f2);

    char text[256];
    while (fin.getline(text, 256))
        fout << text << endl;

    fin.close();
    fout.close();

    return;
}
```

Answer:

---

*Please see page 196 for a hint.*

### Problem 5

Given the following file (`grades.txt`) in the format `<name> <score>`:

```
Jack 83
John 97
Jill 56
Jake 82
Jane 99
```

Write a function to read the data and display the output on the screen. Name the function `read()`.

*Please see page 196 for a hint.*

### Problem 6

Write a function to:

- Open a file and read a number. Display the number to the user.
- Prompt the user for a new number.
- Save that number to the same file and quit.

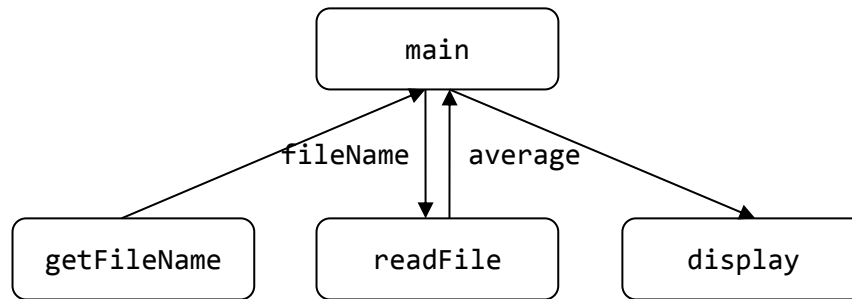
```
The old number is "42". What is the new number? 104
```

Answer:

*Please see page 202 for a hint.*

## Assignment 2.6

Please write a program to read 10 grades from a file and display the average. This will include the functions `getFileName()`, `displayAverage()` and `readFile()`:



### `getFilename()`

This function will prompt the user for the name of the file and return it. The prototype is:

```
void getFileName(char fileName[]);
```

Note that we don't return text the way we do integers or floats. Instead, we pass it as a parameter. We will learn more how this works in Section 3.

### `readFile()`

This function will read the file and return the average score of the ten values. The prototype is:

```
float readFile(char fileName[]);
```

**Hint:** make sure you only read ten values. If there are more or less in the file, then the function must report an error. Display the following message if there is a problem with the file:

```
Error reading file "grades.txt"
```

### `display()`

This function will display the average score to zero decimals of accuracy (rounded). The prototype is:

```
void display(float average);
```

## Example

Consider a file called `grades.txt` (which you can create with emacs) that has the following data in it:

```
90 86 95 76 92 83 100 87 91 88
```

When the program is executed, then the following output is displayed:

```
Please enter the filename: grades.txt  
Average Grade: 89%
```

## Assignment

The test bed is available at:

```
testBed cs124/assign26 assignment26.cpp
```

Don't forget to submit your assignment with the name "Assignment 26" in the header.

*Please see page 201 for a hint.*

## Unit 2 Practice Test

### Practice 2.1

Mike's teacher told his class that a flipped coin lands on "heads" half the time and "tails" the other half. "That means that if I flip a coin 100 times, it should land on heads exactly fifty times!" said Mike. Mike's teacher knows that the law of probability does not quite work that way. To demonstrate this principle, she decides to write a program.

### Program

Write a function to simulate a coin flip, returning true if the coin lands on "heads" and false if it lands on "tails." Next, prompt the user for how many trials will be needed for the experiment. Finally, display how many "heads" and "tails" were recorded in the experiment.

### Example

User input is underlined.

```
How many coin flips for this experiment: 100
There were 49 heads.
There were 51 tails.
```

### Assignment

Please:

- Start from the standard template we use for homework assignments:

```
/home/cs124/tests/templateTest2.cpp
```

- Make sure your professor's name is in the program header.
- Run test bed with

```
testBed cs124/practice21 test2.cpp
```

- Run style checker

Note that the following code might come in handy:

```
#include <stdlib.h> // needed for the rand(), srand()
#include <ctime>    // needed for the time function
int main(int argc, char **argv)
{
    // this code is necessary to set up the random number generator. If
    // your program uses a random number generator, you will need this
    // code. Otherwise, you can safely delete it. Note: this must go in main()
    srand(argc == 1 ? time(NULL) : (int)argv[1][1]);

    // this code will actually generate a random number between 0 and 999
    cout << rand() % 1000 << endl;
}
```

*Continued on the next page*

## Grading for Test2

Sample grading criteria:

|                       | Exceptional<br>100%   | Good<br>90%  | Acceptable<br>70%   | Developing<br>50%   | Missing<br>0%   |
|-----------------------|---|--|---|---|---|
| Expressions<br>10%    | The expression for the equation is elegant and easy to verify           | The expression correctly computes the equation   | One bug exists  | Two or more bugs exist                                      | The expression is missing   |
| Modularization<br>20% | Functional cohesion and loose coupling is used throughout               | Zero syntax errors in the use of functions, but room exists for improvements in modularization | Data incorrectly passed between functions                     | At least one bug in the way a function is defined or called | All the code exists in one function                               |
| Loop<br>40%           | The loop is both elegant and efficient                                  | The loop is syntactically correct <u>and</u> used correctly                                    | The loop is syntactically correct <u>or</u> is used correctly | Elements of the solution are present                        | No attempt was made to use a loop                                 |
| Output<br>20%         | Zero test bed errors  | Looks the same on screen, but minor test bed errors  | One major test bed error                                      | The program compiles and elements of the solution exist     | Program output does not resemble the problem or fails to compile  |
| Style<br>10%          | Well commented, meaningful variable names, effective use of blank lines | Zero style checker errors  | One or two minor style checker errors                         | Code is readable, but serious style infractions             | No evidence of the principles of elements of style in the program |

Solution available at:

```
/home/cs124/tests/practice21.cpp
```

# Unit 2 Project : Calendar Program

Create a calendar for any month of any year from 1753 forward. This is similar to the `cal` utility on the Linux system. Prompt the user for the numeric month and year to be displayed as shown in the example below. Your calculations to determine the first day of the month shall start with January 1, 1753 as a Monday. The challenge here is to take into account [leap years](#).

This project will be done in three phases:

- Project 05 : Design the calendar program
- Project 06 : Compute the offset for a given month and year
- Project 07 : Display the calendar table for a given month and year

## Interface Design

There are three parts of the interface design: the output (that which is displayed on the screen during normal operation), the input (that which the user provides), and the errors (when the user enters different data than the program expects).

### Output

The following is a sample run of the program. The input is underlined.

```
Enter a month number: 1
Enter year: 1753

January, 1753
Su Mo Tu We Th Fr Sa
   1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

The table is formatted in the following way:

| January, 1753 |    |    |    |    |    |    |
|---------------|----|----|----|----|----|----|
| Su            | Mo | Tu | We | Th | Fr | Sa |
|               | 1  | 2  | 3  | 4  | 5  | 6  |
| 7             | 8  | 9  | 10 | 11 | 12 | 13 |
| 14            | 15 | 16 | 17 | 18 | 19 | 20 |
| 21            | 22 | 23 | 24 | 25 | 26 | 27 |
| 28            | 29 | 30 | 31 |    |    |    |

## Input

First the user will be prompted for the month number:

```
Enter a month number: 1
```

Next the user will be prompted for the year:

```
Enter year: 1990
```

## Errors

When the program prompts the user for a month, only the values 1 through 12 are accepted. Any other input will yield a re-prompt:

```
Enter a month number: 13  
Month must be between 1 and 12.  
Enter a month number: 0  
Month must be between 1 and 12.  
Enter a month number: 1
```

The same is true with years; the user input must be greater than 1752:

```
Enter year: 90  
Year must be 1753 or later.  
Enter year: 1990
```

## Project 05

Write a structure chart for the calendar program. Make sure that all the functions exhibit the highest level of cohesion and the lowest level of coupling.

On campus students are required to attach [this rubric](#) to your design document. Please self-grade.



## Project 06

The second part of the Calendar Program project (the first part being the structure part due earlier) is to write the pseudocode for two functions: `computeOffset()` and `displayTable()`.

1. Write the pseudocode for the function `computeOffset`. This function will determine the day of the week of the first day of the month by counting how many days have passed since the 1<sup>st</sup> of January, 1753 (which is a Monday and `offset == 0`). That number (`numDays`) divided by 7 will tell us how many weeks have passed. The remainder will tell us the offset from Monday. For example, if the month begins on a Thursday, then `offset == 3`. The prototype for the function is:

```
int computeOffset(int month, int year);
```

Please do not plagiarize this from the internet; you must use a loop to solve the problem. The output for this function is the following:

| Day       | offset |
|-----------|--------|
| Sunday    | 6      |
| Monday    | 0      |
| Tuesday   | 1      |
| Wednesday | 2      |
| Thursday  | 3      |
| Friday    | 4      |
| Saturday  | 5      |

2. Write the pseudocode for the function `displayTable`. This function will take the number of days in a month (`numDays`) and the offset (`offset`) as parameters and will display the calendar table. For example, consider `numDays == 30` and `offset == 3`. The output would be:

```
Su  Mo  Tu  We  Th  Fr  Sa
    1   2   3
 4   5   6   7   8   9  10
11  12  13  14  15  16  17
18  19  20  21  22  23  24
25  26  27  28  29  30
```

There are two problems you must solve: how to put the spaces before the first day of the month, and how to put the newline character at the end of the week. The prototype of the function is:

```
void displayTable(int offset, int numDays);
```

## Project 07

The final part of the Calendar Program project is to write the code necessary to make the calendar appear on the screen:

```
Enter a month number: 1
Enter year: 1753

January, 1753
Su  Mo  Tu  We  Th  Fr  Sa
    1   2   3   4   5   6
 7   8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30  31
```

A few hints:

- Check the case where `offset == 6` when the month begins with Sunday. July 2001 is an example of such a month. You do not want to have a blank line between the column headers (the days of the week) and the day numbers.
- Check the case where the last day of the month is on a Saturday. March 2001 is an example of such a month. You do not want a blank row at the bottom of the calendar.
- Use the `cal` program built into the Linux system when you test the program by hand.

An executable version of the project is available at:

```
/home/cs124/projects/prj07.out
```

Please do the following:

1. Start with the code written in Project 06.
2. Fix any bugs that escaped your notice the first time through.
3. Verify your solution with `testBed`:

```
testBed cs124/project07 project07.cpp
```

4. Submit your code with "Project 07, Calendar" in the program header.