University of Cincinnati
Department of Electrical and Computer Engineering

How to Design A Computer Program: An Introductory Approach
H. Carter

Introduction

Efficiently developing an efficient, correct, maintainable computer program is a task that is usually divided into two parts. First, the logical, functional, and structural form of the program needs to be defined, followed by implementing the documented design using a programming language. This tutorial presents a method for designing simple programs – the sort of programming problems freshmen students in engineering and computer science are likely to see.

The approach taken here is necessarily functional and hierarchical in form. If instead object-oriented design is the focus, then consider the approach outlined here most suitable for the design of class methods as well as non-class functional programs. There are other design approaches, complementary to what is described here, the support object-oriented design.

How to create a Design and use it during the Coding Process

A design is started by examining the functional requirements imposed by the problem to be solved by the program. Then functionally structured sequential set of steps at a high level of abstraction are defined. Finally, each one of these abstract steps is refined (ie, defined) by less-abstract steps which are in turn refined by ever-less abstract steps until finally the lowest-level steps can be implemented in code with very little thought.

This approach, which has been well-known for years as iterative refinement, forms an outline if each step is described in English or pseudo-English statements. Another way of looking at the design is as a tree with where the first level nodes are the high-level abstract statements, the next level of nodes represent the next higher-level of abstract definitions, and so forth down to the leaves which represent those definitions which can be easily coded (by, typically, 1-10 code statements).

There are several attractive features about using an outline form of design

1. Pick any node in the hierarchy of design and implement it as a function. Then the subtree of statements below that node is the design of the function.
2. Hopefully (and with design discipline) the creative experience and thinking designing the solution is done in a much more forgiving and useful paradigm than the code. In fact, the ideal is that all the creativity is invested in the process of design rather than in the code writing.
3. The process of design becomes an excellent tool to teach young students the fundamental process used in engineering design.
4. Finally, a proper design can almost mechanize the coding phase in the following way:
   a) Write the entire design as comments in the coding language.
   b) Cut the subtrees (copy the roots) to be implemented as functions and paste them, in desired sequence, after the at the end of the design e.g., for C++ programs).
   c) Expand each statement (which is a comment) so code can be inserted potentially after each one.

1 Created to support EECE180 – Software Methods 1, a freshman-level software concepts and programming course
d) (For C++) wrap the (now much smaller) main design comments with int main () { }, and each cut and pasted subtrees with <return_type> <function_name> ( ) { .... }. What you should exist at this point are the templates of all functions each with it's design, as comments, as the body.

e) For each function, including main ( ), fill in the code after each design statement. The design statement, since it is a comment, will provide commenting for aid in maintenance of the program. Typically, object declarations and initialization take place after higher abstract design statements, while executing code will be inserted after less abstract statements. Filling in the function arguments and defining the function return types will occur during this process as well.

f) Lastly, create the prototypes, include statements, and use statements in the public space at the top of the program. As necessary, split into files (.h and .cpp files for C++) to better organize larger programs.

This approach has been quite successful in freshman-level programming courses in the College of Engineering at the University of Cincinnati where it has been applied. The greatest difficulty is for students to truly conceptualize the design such that it represents a valid solution, and aids in the coding process. But with practice and time, they become quite proficient doing design this way, and even anecdotally enjoy it!

Let's look at a meaningful, step-by-step example of the process of design as outlined above and demonstrate how to code from the design once it is created.

Example

Consider the following requirements for a computer program.²

1. Functional Requirements
   a) The program shall, given an integer number provided by the user, display the first three powers of the number.
   b) Inputs:
      • A single positive integer number
   c) Outputs:
      • Display of the banner: (centered on screen):

          Powers of a Number
          Lecture 1 Example
          <date of lecture>
          <name of programmer>

      • Display of the first three powers of the number using the format (centered on screen):

          First Power of <number>: <number>
          Second Power of <number>: <number*number>
          Third Power of <number>: <number*number*number>

2. Performance Requirements (usually time and space minimums or maximums)
   a) None

² This simple requirements format is also taught to UC ECE freshmen to begin their education in the process of engineering, and to provide a less ambiguous way to provide problem descriptions for labs and such.
3. Implementation Requirements
   a) Implement using the C++ language
   b) Comply with the C++ style rules given in the Laboratory Guide (2010 versions)
   c) Use the g++ compiler
   d) Display output is ASCII text

Given the requirements specification above, we need to next design the program. The following process is a step-wise, top-down approach using iterative decomposition (i.e., proceeding from abstract to concrete statements) of the execution of the program, not how to code the step. The steps end when the concrete design steps can be easily coded in C++.

Not in the following design that
• Each execution step is a complete sentence
• Each step is in (pseudo-) English, not code
• Each substep is a refinement of the step in which it resides

So, to begin, we think about what major execution steps will need to occur, in sequence, by the program: 1) display the banner, 2) get a number from the user, 3) calculate the first three powers of the number provided by the user, and 4) display the powers. This sequence of actions isn't unique. For example, we could display the banner as step 3 rather than step 1, but some steps must occur in sequence such as get the number from the user before calculating the powers (steps 2 and 3).

We write these four major execution steps of our program as the first-level of our design (in the form of an outline).

1. Display the banner
2. Get a number from the user
3. Calculate the first three powers of the number provided by the user
4. Display the powers

We now examine each first-level step and, if necessary, refine it with sub-steps (2nd-level). Starting with Step 1, the only refinement necessary is to either explicitly show the display format of the banner as given in the requirements specification. Note that we fill out the meta-words (e.g., <date of lecture>).

1. Display the banner. Format (centered on screen):
   Powers of a Number
   Lecture 1 Example
   5 January 2010
   Hal Carter

2. Get a number from the user
3. Calculate the first three powers of the number provided by the user
4. Display the powers

Only a few common C++ statements will be needed to code Step 1 so we can leave the design of Step 1 and refine the design of Step 2 to the substep level. Considering what it will take for a C++ program to execute “Get a number from the user”, the program should first request the user to input a number, then get the number the user types in. Finally, the program needs to ensure the user provided a number that is not too large, or is really a number and not something like “asdfkljjj”. So there are three substeps to Step 2: 1) request a number from the user, 2) get the number the user types on the keyboard, and 3) verify the number is a valid one:

1. Display the banner. Format (centered on screen):
2. Get a number from the user
   a. Display request of input message.
      Format (left adjusted): "Please enter a number [integer; < MAX]: "
   b. Obtain positive integer 'userNumber' from user.
   c. Validate 'userNumber'. Exit program if error.

3. Calculate the first three powers of the number provided by the user
4. Display the powers

Upon examining the substeps of Step 2 we decide that Substeps a and b can be easily coded in C++ and will not need any further refinement. But substep c needs to be further designed. Just what conditions does 'userNumber' have to satisfy to be valid? According to the requirements specification, it has to be a single number of type integer. There is probably a limit on the size of the number which we need to check for as well. That leads us to adding some substeps to Step 2a. We'll use the term “MAX” as a placeholder until the design is finished, and then figure out what value MAX should be.

1. Display the banner. Format (centered on screen):
   Powers of a Number
   Lecture 1 Example
   5 January 2010
   Hal Carter

2. Get a number from the user
   a. Display request of input message.
      Format (left adjusted): "Please enter a number [integer; < MAX]: "
   b. Obtain positive integer 'userNumber' from user.
   c. Validate 'userNumber'. Exit program if error.
      1) Is 'userNumber' other than an integer? If so display error message.
         Format: "Input not an integer number. Rerun program and enter a positive integer."
      2) Is 'userNumber' too large (> MAX)? If so display error message.
         Format: "Input too large - must be less than MAX"

3. Calculate the first three powers of the number provided by the user
4. Display the powers

Next, we refine Step 3 followed by refining Step 4. The design below shows the substeps for both of these steps, and, in fact, is the final design, except for figuring out what MAX is.

1. Display the banner. Format (centered on screen):
Powers of a Number
Lecture 1 Example
5 January 2010
Hal Carter

2. Get a number from the user
   a. Display request of input message.
      Format (left adjusted): "Please enter a number [integer; < MAX]:"
   b. Obtain positive integer 'userNumber' from user.
   c. Validate 'userNumber'. Exit program if error.
      1) Is 'userNumber' other than an integer? If so display error message.
         Format: "Input not an integer number. Rerun program and enter a positive integer."
      2) Is 'userNumber' too large (> MAX)? If so display error message.
         Format: "Input too large - must be less than MAX"

3. Calculate first three powers of the number provided by the user
   a. Calculate firstPower = userNumber
   b. Calculate secondPower = userNumber * userNumber
   c. Calculate thirdPower = firstPower * secondPower

4. Display the powers
   Format:
   First Power of 'userNumber': firstPower
   Second Power of 'userNumber': secondPower
   Third Power of 'userNumber': thirdPower

Except for resolving MAX, we're done! But what is MAX? MAX is the maximum value for which a number input by the user is valid. Since a C++ integer is a signed 32-bit number, and $2^{32}$ is a little over 4,000,000,000, then the maximum (approximately) positive or negative integer is +/- $2^{31}$, or approximately 2,000,000,000. We need to ensure that any value calculated from the input does not exceed approximately 2,000,000,000. The largest calculated value will be the third power of the input, or the largest input can be no larger than the cube root of 2,000,000,000 which is (using a calculator) is 1260. Thus, we replace MAX with 1260 in our design:
1. Display the banner. Format (centered on screen):
   Powers of a Number
   Lecture 1 Example
   5 January 2010
   Hal Carter

2. Get a number from the user
   a. Display request of input message.
      Format (left adjusted): "Please enter a number [integer; < 1260]:"
   b. Obtain positive integer 'userNumber' from user.
   c. Validate 'userNumber'. Exit program if error.
      1) Is 'userNumber' other than an integer? If so display error
         message.
         Format: "Input not an integer number. Rerun program and enter a
         positive integer."
      2) Is 'userNumber' too large (> 1260)? If so display error message.
         Format: "Input too large - must be less than 1260"

3. Calculate first three powers of number provided by the user
   a. Calculate firstPower = userNumber
   b. Calculate secondPower = userNumber * userNumber
   c. Calculate thirdPower = firstPower * secondPower

4. Display the powers
   Format:
   
   First Power of 'userNumber': firstPower
   Second Power of 'userNumber': secondPower
   Third Power of 'userNumber': thirdPower

Final Design

**Implementing the Design**

Now that we have our design, how do we convert it into C++ code? It's easy – just cut and paste the design into the space where you want the code, put comment marks in each line of the pasted design, and insert C++ code before and in-between each design statement that implements the design task.

Here's what the file looks like after the design has been pasted into the code area and converted into C++ comments (with blank comment code lines to make the final code-with-comments readable):

```cpp
// 1. Display banner. Format (centered on screen):
// Powers of a Number
// Lecture 1 Example
```
2. Request user input a positive integer
   
a. Display request of input message
      Format (left adjusted): "Please enter a number (integer; < 1260):

b. Obtain positive integer ('userNumber') from user.

c. Validate 'userNumber'. Exit program if error.
   
   1) Is 'userNumber' other than an integer? If so display error
      msg.
      Format: "Input not an integer number. Rerun program and enter a
      positive integer."

   2) Is 'userNumber' too large (> 1260)? If so display error message
      Format: "Input too large - must be less than 1260"

3. Calculate first three powers of number provided by the user
   
a. Calculate firstPower = userNumber
   b. Calculate secondPower = userNumber * userNumber
   c. Calculate thirdPower = firstPower * secondPower

4. Display the powers
   
   Format:
   
   First Power of 'userNumber': firstPower
   Second Power of 'userNumber': secondPower
   Third Power of 'userNumber': thirdPower

And now for the code. Starting at the top (although it isn't necessary to always implement the code starting at
the top – middle out, lowest level first, bottom up – it's your preference), we implement the code, concentrating
only on the substep we're trying to implement. It's also a good idea to implement a little (not the whole program
at once!) and compile/execute/debug the small piece, then add some more code and compile/execute/debug the
enlarged partial program, and continue this way until the entire program is completely and correctly implemented.
Here's the final program, along with the file header comments (which includes the design):

    // *******************************************************
    // File:    main.cpp
    // lec1_example - First three powers of a user-supplied number
    // Author:  Hal Carter
    // Created: 3 January 2010
    // Updated: 4 Jan 10 Updated MAX
    // 4 Jan 10 Modified comments
    // Assignment: Lecture 1, example
    // Compiler: Codeblocks 8.02 build 23 Sep 09, with gcc g++ 4.4.1 compiler
    // *******************************************************
    // Inputs: Integer number 'userNumber' entered by user
    // Outputs: Display 'userNumber' raised to power 1, 2, and 3
    // Constraints: absolute value of 'userNumber' must be less than cube root
    //              of 2^31 (to avoid overflow)
    // *******************************************************
    // Design
    // 1. Display banner. Format (centered on screen):
    // Powers of <userNumber>
    //      Lecture 1 Example
    //       5 January 2010
    //       Hal Carter
    // 2. Request user input a positive integer
    // a. Display request of input message.
       Format (left adjusted): "Please enter a number [integer; < 1260]:
    // b. Obtain positive integer 'userNumber' from user.
    // c. Validate 'userNumber'. Exit program if error.
       1) Is 'userNumber' other than an integer? If so display error
       message.
       Format: "Input not an integer number. Rerun program and enter a
       positive integer."
       2) Is 'userNumber' too large (> 1260)? If so display error message.
       Format: "Input too large - must be less than 1260"
    // 3. Calculate first three powers of number provided by the user
       a. Calculate firstPower = userNumber
b. Calculate secondPower = userNumber * userNumber

// c. Calculate thirdPower = firstPower * secondPower

// 4. Display powers
// Format:
//    First Power of 'userNumber': firstPower
//    Second Power of 'userNumber': secondPower
//    Third Power of 'userNumber': thirdPower

#include <iostream>
#include <cstdlib>
using std::cout;
using std::cin;
using std::endl;
using std::exit;

const int     OK = 0;
const int  NOTOK = 1;
const int    MAX = 1260;

int main (int nargs, char *argv[]) {
  int userNumber;

// 1. Display banner. Format (centered on screen):
//     Powers of a Number
//     Lecture 1 Example
//     5 January 2010
//     Hal Carter
//
  cout << "Powers of a Number\n";
  cout << "Lecture 1 Example\n";
  cout << "5 January 2010\n";
  cout << "Hal Carter\n";
  cout << endl;

// 2. Request user input a positive integer
// a. Display request of input message. )
// Format (left adjusted): "Please enter a number [integer; < 1260):
//
  cout << "Please enter a number [integer; < 1260): ";

// b. Obtain positive integer ('userNumber') from user.
//
cin >> userNumber;

    // c. Validate 'userNumber'. Exit program if error.
    //     1) Is 'userNumber' other than an integer? If so display error
    //        message.
    //        Format: "Input not an integer number. Rerun program and enter a
    //                 positive integer."
    //
    if (!cin) {
        cout << "Input not an integer number. Rerun program and enter a";  
        cout << " positive integer\n\n";  
        exit (NOTOK);
    }

    //          2) Is 'userNumber' too large (> 1260)? If so display error
    //             message.
    //             Format: "Input too large - must be less than 1260"
    //
    if (userNumber > MAX - 1) {
        cout << "Input too large - must be less than " << MAX << ":."
        cout << " Rerun program and enter a smaller number.\n\n";
        exit (NOTOK);
    }

    //   3. Calculate first three powers of numbers provided by the user
    //     a. Calculate firstPower = userNumber
    //     b. Calculate secondPower = userNumber * userNumber
    //     c. Calculate thirdPower = firstPower * secondPower
    //
    int firstPower = userNumber;
    int secondPower = userNumber * userNumber;
    int thirdPower = firstPower * secondPower;

    //   4. Display powers
    //     Format:
    //         First Power of 'userNumber': firstPower
    //         Second Power of 'userNumber': secondPower
    //         Third Power of 'userNumber': thirdPower

    cout << endl;
    cout << " First Power of 'userNumber': " << firstPower << endl;
    cout << " Second Power of 'userNumber': " << secondPower << endl;
    cout << " Third Power of 'userNumber': " << thirdPower << endl;

    return OK;
}