Designing and Developing Programs with Functions

Here is a list of guidelines to which to adhere:

1) Do NOT start coding immediately. Instead plan out the following:

> **A) How you will store your data**
> **B) What functions you will have**
> **C) What they will do**
> **D) The parameters they take in**
> **E) What they return**
> **F) The overall flow and design of main**

When planning, you don't necessarily have to plan these in a specific order. It's likely that you'll go back and forth, planning a function, then looking at main, and then realizing you need another function, etc. This part of the process isn't easy. It's something that gets better with practice, like everything else.

2) Once you have your plan, incrementally write code, test code and debug code. Do NOT write the whole program before you try to compile it!!!

The order in which you do things here is also variable. Some people like coding a skeleton for their main function first. Others like trying to write a full function without touching main first. Either approach can work.

The key is not writing too much before you stop to test.

If you choose to write main first, then just write a very scaled down skeleton that simply runs the overall flow of the program and leave tasks to be completed by functions blank. (Use the empty statement if you want to leave a placeholder.) Then compile this skeleton and print out statements in place of functions so you can see that the general structure is working properly.

If you choose to write a function first, write the function, and then test that function by making calls to the function from main. In essence, your main won't be functioning for the program, it's temporary purpose will be to set up variables so that a proper function call can be made, and then checking to see whether the desired result occurred.

3) Basic debugging technique: Inserting printf statements at various points in the code so that you can see the value of variables at different points in time. A better technique is to use a debugger, but until you learn how to do that, printfs will do.

4) Black-box testing of functions: Once a function is written, design test cases based on the input and output specifications of the function and WITHOUT looking at the actual code of the function. Try out all cases that seem to be different based on the specifications.

5) White-box testing of code: Look at the code and create test cases that specifically run each line of code that is written. Basically, you must look at each if statement you have, and each loop, and make sure at least one of your test cases enters each possible branch of that construct.

Fruit Stand Example

Consider creating a program that allows the user to buy fruit from a fruit stand. The user can continually choose to buy fruit until they quit. The two fruits sold at the fruit stand are apples and oranges. Each time the user visits the fruit stand, they are asked whether or not they want to buy apples, and whether or not they want to buy oranges. If they answer yes to either question, then they are asked how many. After this sequence, the user is asked for cash to carry out the transaction. Then, the program calculates the amount of change the user will receive and print out a string representation of this change, once character per each coin. Furthermore, the apples and oranges are taxed in a strange way: The tax is one penny for each fruit, plus 2% of the original transaction price, truncated. At the end of the program a report is printed of how many fruits the user bought and how much he/she has left.

Our main will simply prompt the user for how much cash they have originally, and then it will run a loop that continues as long as the user enters that they want to buy fruit. After this is over, the status report is printed.

Now, let's think about what functions we might use:

1) menu function to prompt to the user.
2) buy function that carries out a whole transaction.
2) status report function to print out the status report
3) give_change function which asks the user how much they are paying and returns their change to them.
4) a print_change function that, given a number of cents, prints out how to give change
5) a tax function that takes in the necessary information and computes the tax based on the formula above.

Let's consider the development of one function: print_change.

Based on what it does, we can come up with the following function prototype:

```
void print_change(int cents);
```

Our precondition will be that cents is non-negative.
Our postcondition will be that the function prints out each coin of change so that the sum total of the coins is equal to cents number of cents.

Consider the following initial implementation:

```
void print_change(int cents) {

  int quarters, dimes, nickels, pennies, i;

  quarters = cents/25;
  dimes = cents/10;
  nickels = cents/5;
  pennies = cents%5;

  for (i=0; i<quarters; i++)
    printf("Q");
  for (i=0; i<dimes; i++)
    printf("D");
  for (i=0; i<nickels; i++)
    printf("N");
  for (i=0; i<pennies; i++)
    printf("P");
  printf("\n");
}
```

Once we write this, we should test it immediately, before moving on. We can do this by simply calling the function in an otherwise empty main. Now the question becomes, what test cases should we try? Here is a list of a few:

1) 0
2) 1
3) 5
4) 10
5) 25
6) 47
7) 100

We should try each denomination, followed by one case that should have each coin and another case that should not. In the following main, I will test two of these cases:

```
int main() {

  printf("0 cents change is ");
  print_change(0);

  printf("5 cents change is ");
  print_change(5);

  printf("47 cents change is ");
  print_change(47);

  return 0;
}
```

Our output:
```
0 cents change is
5 cents change is N
47 cents change is QDDDDNNNNNNNNNPP
```

Is this output correct?

The first two cases are, but the last case prints out too many coins. Upon further examination, we see that the number of each coin printed out approaches the total amount by themselves, except for the pennies which are correct.

In making this observation and looking at the code, we realize that after giving a quarter of change, we need to lower the number of cents we are making change for. Utilizing that idea, we adjust the code as follows:

```
void print_change(int cents) {

  int quarters, dimes, nickels, pennies, i;

  quarters = cents/25;
  cents = cents - quarters*25;
  dimes = cents/10;
  cents = cents - dimes*10;
  nickels = cents/5;
  cents = cents - nickels*5;
  pennies = cents%5;

  for (i=0; i<quarters; i++)
    printf("Q");
  for (i=0; i<dimes; i++)
    printf("D");
  for (i=0; i<nickels; i++)
    printf("N");
  for (i=0; i<pennies; i++)
    printf("P");
  printf("\n");
}
```

Now, consider developing the give_change function. It will take in how much cash the user has, and how much they spent on their last transaction. With this information, it will ask the user how much they are giving, and then provide change for the user. Here is an initial attempt at this function:

```
void give_change(double cash, double spent)
{
  double money;
  int change;
  printf("How much will you pay for this
          purchase?\n");
  scanf("%lf", &money);

  change = (money - spent)*100;
  print_change(change);

}
```

In this implementation, we haven't yet error checked to see if the user is trying to give us more money than they have. We will add that in later. We will try one test case for now, making the following function call from main:

```
give_change(10, 1.35);
```

When running this, we enter 2.00 for the amount we want to pay. The output we get is as follows:

```
Here is your change: QQDPPPP
```

At this point, we see there must be a problem, because QQDN is the desired output. Our problem must be in one of two places:

1) print_change is working incorrectly
2) give_change is passing the wrong value to print_change

Before we fix the problem, we must identify which of these two is occurring. The easiest way to rule one of these out is to print out the value of the parameter to print_change like so, in the give_change function:

```
void give_change(double cash, double spent)
{
  double money;
  int change;
  printf("How much will you pay for this
          purchase?\n");
  scanf("%lf", &money);

  change = (money - spent)*100;
  printf("change is %d.\n", change);
  print_change(change);

}
```

We run this and find out that change is 64, instead of 65. What might be causing this problem??? We are assigning change to a double, so somehow the answer isn't being converted properly. Thus, we must convert an expression that is a double representing dollars to an expression that is an int representing cents. Consider changing the assignment statement like so:

```
 change = (int)(100*money)-(int)(100*spent);
```

Why might this help?

Because we know that money and spent will be stored to two decimal places. When subtracting the two immediately, we may have a slight round-off error. But if we multiply each by 100, that should give us an integer value representing the number of cents. We then cast this to an int. Technically, to be safe, we should call a rounding function on each of these. But, let's try this first. We try it, and see that indeed, it does work this time. It outputs:

```
Here is your change: QQDN
```

If one wanted to be truly thorough, one could try each possible value for spent from 1.00 to 1.99, incrementing by one cent each time. But, this is most likely time-prohibitive for a program like this. (For a super-important program, you would do something tedious like that.)

As it makes sense, in developing code, one would add meaningful portions of main. To do this, the menu function would have to be written, and then a basic shell could be tested.

```
int main() {

  double cash;
  int ans;
  printf("What is your beginning amount of
          cash?\n");
  scanf("%lf", &cash);

  ans = menu();
  while (ans != 2) {
    if (ans == 1)
      give_change(cash, 1.24);

    ans = menu();
  }

  printf("Thanks for coming to the fruit
          stand.\n");
  return 0;

}

int menu() {

  int num;

  printf("\nPlease enter your choice.\n");
  printf("1. Buy apples & oranges.\n");
  printf("2. Quit\n");
  scanf("%d", &num);
  return num;
}
```

Now, with this frame work, you can actually run a somewhat of a meaningful program. Basically, you can test out your looping structure, and instead of buying anything, you just call the give_change function each time to test it out.

After this is written, they you can add it error checking, as appears in the final copy of the fruit stand program.

Conclusion

Hopefully this example sheds some light upon how to go about developing code. It IS important to test code as you write it, and not wait till your whole program is written. As shown above, testing as you go can help catch mistakes early, and usually they are easier to track down, because you are searching for a problem in a relatively small amount of space. Furthermore, pay attention to how to create test cases, and then how to execute those test cases. Before you submit your programs, you should test them at least 10 sets of values, minimum. Those should be chosen using both the black and white box testing strategies, plus some random cases.

The more you practice these strategies, the quicker you will become at debugging and getting your code to work properly!

Above all, I hope this also shows why it's so important to get code compiling early, and why I take off a large deduction for code that doesn't compile. (It's because you definitely didn't test the code if you couldn't compile it. Without testing, you have no clue what the code does.)