# How to Design Functions

CMSC21 - University of the Philippines Cebu

*Adapted from Systematic Program Design*

```
// float -> float
// Returns double the value of the given number

/* stub
float twice(float n) {
  return 0;
}
*/

/* template
float twice(float n) {
  return ... n;
}
*/

float twice(float n) {
  return n * 2;
}

int main() {
  printf("%d, %f == %f\n", twice(2.0) == 4.0, twice(2.0), 4.0);
  printf("%d, %f == %f\n", twice(1.5) == 1.5 * 2, twice(1.5), 1.5 * 2);
  printf("%d, %f == %f\n", twice(5.0) == 10.0, twice(5.0), 10.0);
  return 0;
}
```

The HtDF recipe consists of the following steps:

1. Signature, purpose and stub
2. Define examples
3. Template and inventory (create constants)
4. Code the function body
5. Test and debug until correct

The How to Design Functions (HtDF) recipe is a design method that enables systematic design of functions. Note that each of these steps build on the ones that precede it. The signature helps write the purpose, the stub, and the examples; it also helps code the body. The

purpose helps write the examples and code the body. The stub helps to write the examples. The examples help to code the body as well as to test the complete design.

It is sometimes helpful to do the steps in a different order. Sometimes it is easier to write examples first, then do signature and purpose. Often at some point during the design you may discover an issue or boundary condition you did not anticipate, at that point go back and update the purpose and examples accordingly. But you should never write the function definition first and then go back and do the other recipe elements -- for some of you that will work for simple functions, but you will not be able to do that for the complex functions later in the course!

Throughout the HtDF process be sure to "run early and run often". Run your program whenever it is well-formed. The more often you press run the sooner you can find mistakes. Finding mistakes one at a time is much easier than waiting until later when the mistakes can compound and be more confusing. Run, run, run!

---

## Signature, purpose and stub

Write the function signature, a one-line purpose statement and a function stub.

A signature has the type of each argument, separated by commas, followed by `->`, followed by the type of result. So a function that accepts two `int`s and produces a `float` would have the signature `int, int -> float`.

The purpose is a one-line statement which answers the question, "What does this function compute?" A well-formed purpose mentions each of the given parameters and how they relate to the return value.

The stub is a syntactically complete function definition that produces a value of the right type. If the type is `int` or `float` it is common to use 0, if the type is `char` it is common to use `'a'` and so on. The value will not, in general, match the purpose statement. In the example below the stub produces 0, which is a `int`, but only matches the purpose when twice happens to be called with 0.

```
// float -> float
// Returns double the value of the given number
float twice(float n) {
  return 0;
}
```

The purpose of the stub is to serve as a kind of scaffolding to make it possible to run the examples even before the function design is complete. Most of the tests will fail of course, but

the fact that they can run at all allows you to ensure that they are at least well-formed: parentheses are balanced, function calls have the proper number of arguments, function and constant names are correct and so on. This is very important, the sooner you find a mistake -- even a simple one -- the easier it is to fix.

---

## Define examples

Write at least one example of a call to the function and the expected result the call should produce.

You will often need more examples, to help you better understand the function or to properly test the function. If you are unsure how to start writing examples use the combination of the function signature and purpose to help you generate examples.

The first role of an example is to help you understand what the function is supposed to do. If there are boundary conditions be sure to include an example of them. If there are different behaviours the function should have, include an example of each. Since they are examples first, you could write them in this form:

```
int main() {
  printf("%f == %f\n", twice(2.0), 4.0);
  printf("%f == %f\n", twice(1.5), 3.0);
  printf("%f == %f\n", twice(5.0), 10.0);
  return 0;
}
```

When you write examples it is sometimes helpful to write not just the expected result, but also how it is computed. For example, you might write the following instead of the above:

```
int main() {
  printf("%f == %f\n", twice(2.0), 4.0 * 2);
  printf("%f == %f\n", twice(1.5), 1.5 * 2);
  printf("%f == %f\n", twice(5.0), 10.0 * 2);
  return 0;
}
```

While the above form satisfies our need for examples, adding the equality check in the `printf` statement will allow us to spot failed test cases more easily.

```
int main() {
  printf("%d, %f == %f\n", twice(2.0) == 4.0, twice(2.0), 4.0);
  printf("%d, %f == %f\n", twice(1.5) == 1.5 * 2, twice(1.5), 1.5 * 2);
  printf("%d, %f == %f\n", twice(5.0) == 10.0, twice(5.0), 10.0);
  return 0;
}
```

The existence of the stub will allow you to run the tests. Most (or even all) of the tests will fail since the stub is returning the same value every time. But you will at least be able to check that the parentheses are balanced, that you have not misspelled function names etc.

---

## Template and inventory

Before coding the function body it is helpful to have a clear sense of what the function has to work with -- what is the contents of your bag of parts for coding this function? The template provides this.

For primitive data like numbers and characters the body of the template is simply `return ... x;` where `x` is the name of the parameter to the function. It is also often helpful to add constant values which are likely to be useful to the template body at this point.

Once the template is done the stub should be commented out. Note that this template won't compile! The compiler doesn't recognize `...`. You'll have to comment the template out as well, but first copy-paste a duplicate of the template for use in the next step.

```
// float -> float
// Returns double the value of the given number

/* stub
float twice(float n) {
  return 0;
}
*/

float twice(float n) {
  return ... n;
}
```

---

## Code the function body

Now complete the function body by filling in the template.

Note that:

- the signature tells you the type of the parameter(s) and the type of the data the function body must produce
- the purpose describes what the function body must produce in English
- the examples provide several concrete examples of what the function body must produce
- the template tells you the raw material you have to work with

You should use all of the above to help you code the function body. In some cases further rewriting of examples might make it more clear how you computed certain values, and that may make it easier to code the function.

```
// float -> float
// Returns double the value of the given number

/* stub
float twice(float n) {
  return 0;
}
*/

/* template
float twice(float n) {
  return ... n;
}
*/

float twice(float n) {
  return n * 2;
}
```

## Test and debug until correct

Run the program and make sure all the tests pass, if not debug until they do. Many of the problems you might have had will already have been fixed because of following the "run early, run often" approach. But if not, debug until everything works.