The Design of Functions

Why We Use Functions

The following facts support the use of functions as a programming tool.

- Functions allow large sections of code to be represented by a single line: this allows programmers to organize programs that are thousands of lines long. It also makes it much easier for a person to understand a large program. Without functions, it would be almost impossible to comprehend large programs.
- Functions allow a commonly performed task (possibly representing a large chunk of code) to be listed just once rather than repeated each time it is used. This can shorten considerably the overall length of a program.
- Functions help programmers deal with large projects: individual functions can be "farmed out" to various programming teams.
- Functions can often be reused. This prevents programmers from not having to "reinvent the wheel". Groups of useful functions can be made accessible by putting them into "libraries".

In view of the importance of functions, it follows that functions should be carefully designed, not just "slapped together". The following choices must be made wisely if you wish to write a well designed function.

1) What the function will do.
2) The name of the function.
3) The function parameters: value parameters and/or reference parameters.
4) Whether the function will return a value.
5) Whether the function will be self-contained or will reference outside constants.

Deciding What a Function Should Do

It would seem that decision 1) is an easy task, but it is here that beginning programmers often make errors. Observe first that functions typically fall into two categories:

a) Functions that perform a commonly used task and are potentially reusable in other programs.
b) Functions that are useful in helping to organize a specific program, but probably won't be reused in other programs.

If possible, you should try to write functions that are reusable. The key to reusability is not to design your functions so that they do too much. A function is said to be **logically coherent** if it addresses a single task. Logically coherent functions are much more likely to be reusable than those that are not. See Examples below:

1) A function that computes ***and*** displays the leftmost digit of an integer is ***not*** logically coherent. On the other hand, a function that simply returns the leftmost digit of an int ***is*** logically coherent.
2) A function that reads an int ***and*** displays an error message if an input error occurs is ***not*** logically coherent. However, a function that reads an int and returns an error code via a parameter ***is*** logically coherent.

A common function design error made by programmers is to include output in a function that would actually be more useful without output! Remember: a function is more useful if the caller of the function maintains control over the output. Here's an example.

```
void ExitProgram(int ErrorCode)
{
    cout << "Exiting Program\n";
    exit(ErrroCode);
}
```

This program cannot be used if a "quiet" exit is desired, i.e. no output to screen!

It will sometimes be appropriate for you to write functions that are not logically coherent. For example, you might want to write a function that is specifically designed to help organize a particular program, such as a function to display menu. Also keep in mind that output is not completely banned in functions (for example, functions whose purpose is to produce output).

Choosing Function Names

Functions that *return* nothing or are void should have a name that reflects the *action* taken by the function, and are normally *verb-like*. On the other hand, functions that *return* a value should have a *noun-like* name that suggests the value *returned* by the function. Avoid "Tarzan" names, names that make no sense.

Choosing Function Parameters

You've already decided what your function should do, so it should not be too difficult to decide

a) What information needs to be sent to the function.
b) What information will be produced by the function.

A piece of information that falls into category a) generally will lead to a value parameter. On the other hand, a piece of information that needs to be returned to the caller of a function will be one of two possibilities 1) a reference parameter or 2) a value returning function. If the sole purpose of a function is to compute a single item, it generally works out best if the function returns the value directly, rather than via a reference parameter.

Writing Functions That Are Independent

Whenever possible, functions should be self-contained and **independent** of other functions, i.e. changing function A should not affect function B and vice versa. Those calling a function should not have to supply part of the code for the function, such as initializing parameters.

The following are useful when trying to achieve function independence;

a) Not referencing external (global) variables or constants.
b) Passing input parameters by value when feasible.
c) Not calling other functions.
d) The caller should not have to initialize function parameters.

It may not always be possible or even desirable to write a function that is totally independent. For example, your function may need to reference a constant or a function that belongs to a library.

Designing main()

Writing a well designed main function is important. Avoid including messy details in `main()`, but include enough detail so that one can get an idea of what your program does by reading `main()`. Your `main()` function should serve as an outline or summary of the underlying program details.

Choosing Good Function Names

Names for void Returning Functions

Be especially careful when choosing names for void returning functions. The readability of your main program will be greatly enhanced if these names are thoughtfully chosen to *reflect the action taken by the function* or that of *verb-like* names. Since most functions are called only once or twice, you impose no serious typing burden on yourself by choosing names that could be rather long.

Poor Function Names	Good Names	Comment
PrintTable	WriteTableOfPrimes	Doesn’t reveal to reader what kind of table
PrimeTable	WritePrimeTable	“PrimeTable” doesn’t describe action taken
Explanation	ExplainPrimeNumbers	“Explanation” is noun- doesn’t reflect action
Explain	ExplainPrimeNumbers	Be Specific!
PrimeExplanation	DisplayPrimeExplanation	“PrimeExplanation” is noun
ExplainingPrime	ExplainPrimes	ExplainingPrimes is not like command

A void returning function performs an action, so the name given to this function should capture the essence of this action in the form of a command. It follows that a void returning function name should *not* be a noun. The lines in a program can be viewed as commands given to the computer. These commands should not sound as though they are coming from Tarzan!

Names for Value Returning Functions

A function that returns a numerical value or a character value should be named with a *noun* that tells what is being returned. By contrast, a function that returns a Boolean value should be named with an adjective, a past participle, or a “predicate”. Here are some examples.

Poor Function Names	Good Names	Comment
FindMaximum	Maximum	Function returns a value, and “FindMaximum” is not noun like.
ComputingMaxValue	MaxValue	Again, not noun like
PunctTest	ContainsPunctuation	Be Specific
TestPrime	Prime	“TestPrime” doesn’t sound like a value
PrimeChecker	IsPrime	Calls such as “if (IsPrime(N))”
CheckingPrime	PrimeNumber	or “if (PrimeNumber(N)) seems natural