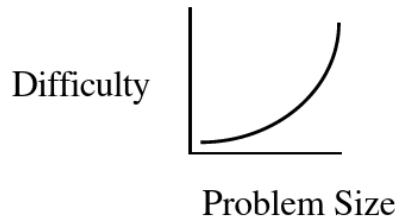


Program Decomposition

Adapted from "[Decomposition & Style](#)" © Nick Parlante, 1996. Free for non-commercial use.

This handout discusses some of the basic ideas for decomposition in large C and C++ programs.

Decomposition is the process of breaking a large problem into more manageable sub-problems. The motivating principle is that large problems are disproportionately harder to solve than small problems. It's much easier to write 2 500-line programs than 1 1000-line program.

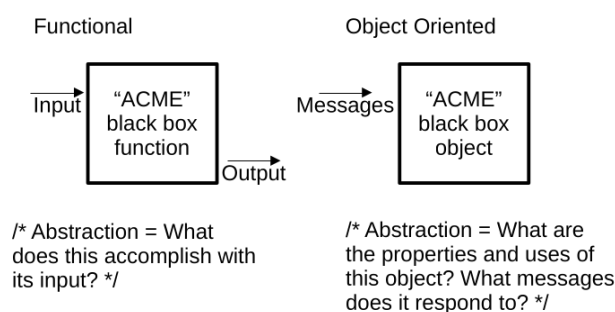


In C the unit of decomposition is the function and the ADT. In C++ the unit of decomposition is the class. For decomposition to work, the sub-parts of the whole problem should be as independent from each other as possible. That does not mean that the sub-parts will not depend on each other at all. They just should not depend on details of each other unnecessarily.

Component independence is especially important in team settings where different sub-problems are solved by different people. A developer needs to be able to focus on each problem without worrying about the rest of the program. The need for good decomposition is magnified as the number of lines and the number of developers on a project grows. It may seem like these rules are overly burdensome for the small programming assignments in your undergraduate classes. However, the goal is develop the essential mental skills now while the programs are small so that later in your career when you work on large programs of tens of thousands or tens of millions of lines you will be well prepared to cope with the complexity.

The Black Box

A well decomposed function or a well designed class is sometimes described as a "black box". The point of the analogy is that the black box is opaque— its inner workings are not observable from the outside. Instead the black box is defined by what it accomplishes. The box has well defined behavior in terms of its input. The black box presents the simplest possible abstraction to describe what the output will be and hides the implementation. A black box is unit of delegation— you get to define it terms of what you want accomplished such as "I want this data structure filled from this file", or "I want a vector of arbitrary size". All the implementation details of how the desired result is achieved are safely isolated inside.



The goal of decomposition is to divide the problem into independent sub-problems. Black boxes are the natural extension of this goal. They can mostly be written and tested independently. To the extent they need to interact, it is through well defined and relatively simple mechanisms. A function is a possibly complex computation wrapped up in a neat package, ready to fit into any code. Similarly, a C++ class represents an object with a tidy looking abstraction and a complete suite of methods.

Abstraction

Decomposition is a divide and conquer strategy. The benefit comes from being able to deal with sub-parts independently. From the outside, black boxes are as simple as possible so they are easy to use and fit together. This process only works if the abstraction presented by the black box makes sense. There should be a clear relationship between the input and output. Typically there is a lot of complexity that should be hidden by the abstraction, necessary in the implementation, but not visible to the user of the black box. The abstraction should only require inputs for what is strictly needed to compute the result. The critical test of the abstraction is: is it easy to describe what this component accomplishes? For functions, you should ask “can you form a description where the name of the function is the verb and the parameters are the nouns”? If the answer is yes then it is a reasonable abstraction. For a C++ member function, the operation should have a clear definition relative to the receiver of the message.

Function Rules in Practice

What should the parameters be for a function? When is a sub-task sufficiently complex or independent to merit being put into a separate function?

A function should solve one problem. Its parameters should include only what is necessary. The abstraction of what the function accomplishes with the parameters should make sense.

- **One problem** A function should solve one problem. It should be easy to describe what a function accomplishes. The lines of the function should step through the sub-parts of the problem. At some point, a sub-part may become sufficiently independent and should be factored out into a separate function.
- **Length** Ideally a function should be 5-25 lines long. However, length is a simplistic measure which cannot replace real analysis based on the structure of the problem. A function may be too long because it solves more than one problem or because it is not decomposed enough (one of its sub-parts needs to be factored out into a separate function). Some functions are forced to be long because their problem solution demands a long sequence of related sub-parts where no portion is independent or complex enough to merit its own function. But, a long function automatically creates doubt about the quality of its decomposition— so be doubly sure that everything in the function is required and it doesn't need further decomposition.
- **Short routines** Very short functions (1-2 lines) are suspicious. The added decomposition may not be worth the additional conceptual overhead of another function. However, even a short function is OK if the code is very complex, requires many local variables not used elsewhere, or is called many times across the code base. Another good argument for a short function is code readability. Replacing a complex expressions such as `sqrt((pow(a.x - b.x), 2) + pow((a.y - b.y), 2))` with the expression `distance(a, b)` clearly improves code readability and is good. Do not worry about the run-time overhead of the additional function calls. Modern compilers are able to easily optimize out this kind overhead.

In C++, one-line accessors/mutators are OK. Because clients of the object go through methods to access everything, the one-line GetXXX and SetXXX method is a very common idiom. C++ compilers are easily able to optimize these to avoid efficiency issues.

- **Parameters** A function should have the smallest number of parameters possible to solve its problem. The function should not add constraints on the input beyond those demanded by the problem. This will make it easier to reuse the function. Sometimes there is no way around having a large number of parameters. As with the function length guidelines above, a large number of parameters is a red flag that the decomposition is inadequate. If a set of parameters are always used together— they should be grouped together in a struct.
- **Complexity** The lines in a function should narrate the steps of the function's algorithm. If the complexity or details of one “step” distract from the core problem, then the offending step should be factored out into a separate function. It is suspicious if a step requires several local variables which are not used in other steps. All of the steps in a function should operate at the same level of detail/abstraction.
- **Repetition** Avoid repeating more than a couple of lines of code. A repeated sequence should be factored out into a separate function. A reasonable “rule of thumb” is: decompose and factor repeated code if it will make the overall program shorter, including the new function and calls. Note that the common sequences of code do not need to be identical. Minor differences can be factored out as parameters of the function. It is a clear warning of duplication if you are copying and pasting code across the code base.
- **Generality** A sub-problem which is an extremely familiar or common idiom should be factored out into a separate function (or functions). Recurring, general problems such as sorting, searching, distance computations, set operations, etc... make excellent independent functions because their abstractions are immediately understood since they have been seen many times. Such functions have the best chance of being reused in other parts of the program or other programs. Furthermore, modern programming languages include built-in or standard library solutions to all the common general problems. Use these pre-defined solutions if they are allowed in assignments (always use them after school).
- **Non-Local Access** Accessing a variable that is outside a function from inside a function should always be through the parameters. Non-local access to a (global) variable violates the black box paradigm. Non-local access means the function is no longer portable (tightly coupled) and its relationship with the rest of the code becomes complex, especially when debugging. Non-local access (global) is only OK for constants and very special types of variables like `sig_atomic_t` flags used in signals.