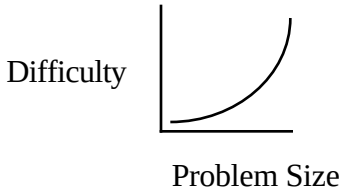
Decomposition & Style

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

Part 1 — Decomposition

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.



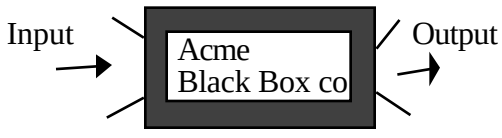
In C the unit of decomposition is the function and the ADT. In C++ the unit of decomposition is the class. For decomposition to work, the subparts of the whole problem should be as independent from each other as possible. That does not mean that they subparts will not depend on each other at all. They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

Independence is especially important in group projects where different subproblems are attacked by different people. The programmer 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 programmers grows. It may seem like Stanford standards are unreasonably high for our 1000 line assignments. We insist on excellence in all aspects of programming, including decomposition, because we are trying to teach skills on 1000-line programs that will get you through 100,000 line programs later.

The Black Box

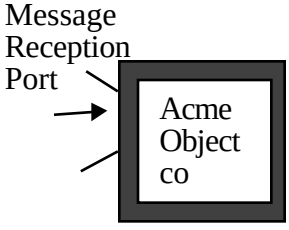
A well decomposed function or a well designed class is sometimes likened to a "black box". The point of the black box is that it is opaque— its inner workings are not apparent. Instead the box is defined by what it accomplishes. The box has well defined behavior in terms of its input. The black box presents the simplest possible abstraction to describe what the output will be and hides the implementation. A black box is a unit of delegation— you get to define it in 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.

Functional



/* Abstracion = what does this accomplish with its input? */

Object Oriented



/* Abstraction = What are the properties and uses of this object? What messages does it respond to ? */

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 independently. To the extent they need to interact, it is through well defined and relatively simple mechanisms. A function is a possibly nasty computation wrapped up in a neat package, ready to fit into any code as easily as possible. 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 subparts 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 basically makes sense. There should be a clear relationship between the input and output. Typically there is a lot of complexity and detail related to the implementation which should not be part of the abstraction. As input, it should require only what is strictly needed to compute the output. The acid test is: is it easy to describe what this component accomplishes? For functions, it's a good sign if you can form a description where the name of the routine is the verb and the parameters are the nouns. For a C++ method, the operation should have a clear definition relative to the receiver.

Function Rules In Practice

What should the parameters be? When is a subtask sufficiently complex or independent to merit being put in its own 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. That does not mean that the function needs to be one line long— the lines in the function step through the subparts of the problem. At some point, a subpart becomes sufficiently independent that it should be factored out into its own function. It should be easy to describe what the function accomplishes.

- Length 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 one of its subparts needs to be decomposed out. Some functions are forced to be long because their problem demands a long sequence of related sub-parts where no subpart is independent or complex enough to merit its own function. A long function automatically creates doubts about the quality of its decomposition— so be doubly sure that everything in the function needs to be there. Ideally a function should be 5-25 lines long.

- Short Routines Very short routines (1-2 lines) are questionable. The added decomposition may not be wort the additional conceptual overhead of another function. Something which can be accomplished in 2 lines is probably not complex enough to merit its own routine. The decomposition can be worthwhile if the lines are very complex, are distracting in the calling function and require there own local variables, or are called a large number of times. A short routine can also be worthwhile just for readability. Replacing the expression `Sqrt((a.x - b.x)^2 + (a.y - b.y)^2)` with the expression `Distance(a,b)` makes the code much more readable, even though the distance function is only one line long. Do not worry about the run-time overhead of an additional function call. A modern compiler can inline the code where appropriate to avoid the call overhead.

  In C++, one-line accessors are fine. Because clients go through methods to access everything, the one-line GetXXX or SetXXX method is a very common idiom. C++ compilers can easily inline these to avoid efficiency problems.

- 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 makes it easier to reuse the function later in the largest number of contexts. Sometimes there is no way around having a large number of parameters. As with function length above, a large number of parameters is a warning sign that the decomposition may not be ideal. If a set of parameters always seem to travel together— should they be packaged 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 problem at hand, then the step should be decomposed out to isolate its problem. It's suspicious if a step requires several local variables which are not used in the other steps. It's good if the steps in a function all operate at the same level of detail.

- Repetition Avoid repeating more than a couple lines of code. A repeated sequence should be put in its own function which is then called where the repeated code used to be. A reasonable rule of thumb is: decompose out a repeated sequence of code if it will make the program shorter.including the new function declaration and calls. The common sequences do not need to be identical. Slight differences may be factored out in the parameters of the function. It's a sure warning sign that you've got some duplication if you find yourself copying and pasting code.

- Generality A subproblem which is an extremely familiar or common idiom should be decomposed out. Recurring, general problems such as Searching, Sorting, Distance, Set intersection, etc… make excellent functions because their abstractions are immediately understood since they have been seen so many times. Such functions have the best chance of being reused in other parts of the program or in other programs. Also, a modern language will include pre-canned solutions to all the common general problems— so having identified the decomposition, you don't have to write anything, you just call the library routine.

- Non-Local Access Access to a variable outside the function should always be through the parameters. Non-local access to a global variable violates the black-box paradigm. The routine is no longer portable and its relationship with all the rest of the code in the program becomes complex, especially when stepping through when debugging. The main advantage of a black box is that it interacts with the outside world through a small, well-defined parameter mechanism to keep the complexity in and the abstraction simple. This helps decomposition and readability,and makes the routine reusable in other routines. Non-local access is only ok for constants and sometimes the standard input and output files.

Part 2 — Style

There's a lot of variation allowable in C coding style. If nothing else, try to be consistent. Here's a just a few style tips which are widely accepted.

Break

Break is ok in loops. Ideally, the loop should be structured to iterate in the most straighforward way. The break in the body can detect the exception case which comes up during the iteration.

```
for (i =0; i < length; i++ ) {
   if (<found>) break;
   ...
}
```

or

```
while (current != NULL) {
   if (<found>) break;
   ...
}
```

While

While (TRUE) types loops are fine if they are really necessary— if the test is the first thing in the loop, then it should be coded as a straight while. If the bounds are known, then a for loop is preferable.

Return

Returns not at the end of a function body are potentially vulgar. They can be used nicely if they detect and immediately return an exceptional case. For example, a base case or error case right at the beginning of a function. Sometimes return can be used like a break inside a loop when some conditition becomes true. Be careful with returns in the bodies of your functions— experience shows they are responsible for a disporportionate number of bugs. The programmer forgets about the early-return case and assumes the function runs all the way to its end.

++, --

Nice obvious uses of these are fine, but nesting it inside something complicated is just asking for trouble. Find a more useful outlet for your cleverness.

```
for (i = 0; i < length; i++) {   // ok
...

while (--length) {               // ok
...

while (*t++ = *s++)              // not ok
...
```

Switch

If you ever exploit the fall-through property of the switch, your documentation should definitely say so. It's pretty unusual.

!=, ==

This is just a minor readability issue, but it's nice to put in what you are really testing for, rather than rely on the anything-non-zero-is-true property. Even though the code may compile to exactly the same thing, it reads a little nicer.

Terse	Nicer
`while (*current) {`	`while (*current != '\0') {`
`while (current) {`	`while(current != NULL) {`
`while (!count) {`	`while (count != 0)`

Boolean Values

The one exception to the above rule is, if you have a boolean value, don't use additional != or == operators on it to test its value. Just use its value directly or with ! to invert. Also, watch for if statements with else cases which assign a vaiable to true or false. The test can go right into the variable.

Terse	Nicer
`while (playing == TRUE) {`	`while (playing) {`
`return(playing == FALSE);`	`return(!playing);`

`if (points<0)`
`  gameOver = TRUE;`
`else`
`  gameOver = FALSE;`
	`gameOver = (points < 0);`