

Optimizing C++/Print Version

Wikibooks.org

December 1, 2012

On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 91. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 89. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 95, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 91. This PDF was generated by the L^AT_EX typesetting software. The L^AT_EX source code is included as an attachment (`source.7z.txt`) in this PDF file. To extract the source from the PDF file, we recommend the use of <http://www.pdfplabs.com/tools/pdftk-the-pdf-toolkit/utility> or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting `Save Attachment`. After extracting it from the PDF file you have to rename it to `source.7z`. To uncompress the resulting archive we recommend the use of <http://www.7-zip.org/>. The L^AT_EX source itself was generated by a program written by Dirk Häjnniger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf. This distribution also contains a configured version of the `pdflatex` compiler with all necessary packages and fonts needed to compile the L^AT_EX source included in this PDF file.

Contents

1	Introduction	3
2	Optimization life cycle	5
2.1	Conventions	7
3	Writing efficient code	9
4	Performance improving features	11
5	Performance worsening features	19
6	Constructions and destructions	23
7	Allocations and deallocations	29
8	Memory access	31
9	Thread usage	33
10	General optimization techniques	35
11	Output	37
12	Memoization	45
13	Sorting	49
13.1	Sorting with small ranges of keys	49
13.2	Partial sorting	50
14	Other techniques	53
15	Code optimization	55
16	Allocations and deallocations	57
17	Run-time support	61
18	Instruction count	63
19	Constructions and destructions	67
20	Pipeline	71
21	Memory access	75
22	Faster operations	79
23	Tools	85
24	Further reading	87
25	Contributors	89
	List of Figures	91
26	Licenses	95
26.1	GNU GENERAL PUBLIC LICENSE	95
26.2	GNU Free Documentation License	96
26.3	GNU Lesser General Public License	96

1 Introduction

One of the main reason for preferring C++ over simpler, higher-level programming languages is that C++ allows the construction of complex software in a way that makes more efficient use of hardware resources than when using these other languages. The language does not guarantee efficient code automatically, but provides a toolchest that aids programmers in the pursuit of efficiency. Sloppy C++ code may be no more efficient than higher-level implementations of the same algorithms, but a good C++ programmer with knowledge of the language can write software that is efficient from the first cut and then **optimize** the code further. This book provides techniques guidelines for writing efficient code and optimizing existing software.

Often, there is no single solution to a programming problem that is optimal for all cases. Thus, optimization¹ generally does not mean writing optimally performing software; rather, it means incrementally changing (refactoring²) software to increase it's performance, bringing it closer to optimality.

Such optimization requires, first, that the software source is written in a sufficiently modular way, to isolate the performance critical parts, and then to use tools, libraries, knowledge, and time, to change those parts in a way to increase the overall execution speed of the overall software.

Nowadays, many optimizations are already performed by compilers, and then they are no longer a programmer's burden. This book discusses higher-level optimizations that present compilers are not (yet) able to perform.

This book is aimed at readers already familiar with the C++ language, who want to use it to develop high performance application software or software libraries.

Almost all the optimization techniques presented are platform independent, and therefore there will be few references to specific operating systems, processor architectures, or compilers. Though, some of the presented techniques come out to be ineffective or not applicable in some combinations of operating system/processor/compiler.

it:Ottimizzare C++/Introduzione³

1 <http://en.wikibooks.org/wiki/%3Aw%3AOptimization%20%28computer%20science%29>
2 <http://en.wikibooks.org/wiki/%3Aw%3ACode%20refactoring>
3 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FIntroduzione>

2 Optimization life cycle

The construction of an efficient application should perform the following development process:

1. **Design**¹. At first, the algorithms and data structures are designed in such a way as makes sense for the application logic, and that is reasonably efficient, but without considering optimization. Where a data structure of wide usage is to be defined, whose optimal implementation is not obvious (for example, it is arguable if is better to use an array or a linked list), an abstract structure² is defined, whose implementation may be changed at the optimization stage.
2. **Implementation**. Then the code that implements the designed algorithms is written, following the guidelines to avoid some inefficient operations, and to encapsulate the operations that will probably require optimization.
3. **Functional testing**³. Then the resulting software is tested, to increase the probability that it doesn't have crippling defects.
4. **Optimization**⁴ (aka *Tuning*). After having completed the development of a correctly working application, the optimization stage begins, having the following sub-stages:
 - a) **Performance testing**⁵. The commands with inadequate performance are spotted, that is the commands that, when processing typical data, require more time or more storage space than what it is specified by requirements are singled out.
 - b) **Profiling**⁶ (aka *Performance analysis*). For every command with inadequate performance, a profiler is used to determine which portions of code are the so-called *bottlenecks*⁷ for that command. Bottlenecks are the portions of code where a disproportionately large amount of time is spent or memory space allocated.
 - c) **Algorithmic optimization**. In bottlenecks, optimization techniques substantially independent from the programming language and totally independent from the platform are applied. They are the techniques that can be found in algorithm theory textbooks. This optimization consists in attempting to decrease the number of the executed machine instructions, and, in particular, the number of the calls to costly routines, or to transform the expensive instructions to equivalent but less costly instructions. For example, the *quick sort*⁸ sorting algorithm is chosen instead of the *selection sort*⁹ algorithm. If this optimization makes the program fast enough, the optimization stage is complete.
 - d) **Platform independent optimization**. In bottlenecks, optimization techniques that are dependent on the programming language and its standard library, but independent both on the software platform and on the hardware platform are applied. For example, integer

1 <http://en.wikipedia.org/wiki/Software%20design>
2 <http://en.wikipedia.org/wiki/Abstract%20data%20type>
3 <http://en.wikipedia.org/wiki/Software%20testing>
4 <http://en.wikipedia.org/wiki/Optimization%20%28computer%20science%29>
5 <http://en.wikipedia.org/wiki/Software%20performance%20testing>
6 <http://en.wikipedia.org/wiki/Performance%20analysis>
7 <http://en.wikipedia.org/wiki/Bottleneck%20%28engineering%29>
8 http://en.wikipedia.org/wiki/Quick_sort
9 http://en.wikipedia.org/wiki/Selection_sort

operations are used instead of floating point operations, or the more appropriate container class is chosen among the ones available in the standard library. If this makes the program fast enough, the optimization stage is complete.

- e) **Software platform dependent optimization.** In bottlenecks, optimization techniques that are dependent both on the programming language and on the software platform, but independent on the hardware platform are applied. For example, compiler options, *pragma* compiler directives, language extensions, non-standard libraries, direct calls to the operating system are exploited. If this makes the program fast enough, the optimization stage is complete.
- f) **Hardware platform dependent optimization.** In bottlenecks, optimization techniques that are dependent on the hardware platform are applied, like machine instructions existing only on a specific processor family or high level features that, even being allowed for every processor architecture, come out to be advantageous only for some processor types.

This development process follows two criteria:

- **Principle of diminishing returns.** The optimizations giving big results with little effort should be applied first, as this minimizes the time needed to reach the performance goals.
- **Principle of diminishing portability.** It is better to apply first the optimizations applicable to several platforms, as they remain applicable even when changing platform, and as they are more understandable by other programmers.

In the rare cases of software that will have to be used with several compilers and several operating systems but just one processor architecture, the stages 4.5 and 4.6 should be swapped.

This stage sequence is not meant to be a one-way sequence, that is such that when a stage is reached, the preceding stage is no more reached. In fact, every stage may succeed or fail. If it succeeds, the next stage is reached, while if it fails, the previous stage is reached, in a sort of backtracking algorithm.

In addition, a partial performance test should be performed after every optimization attempt, just to check whether the attempt comes out to be useful, and, in the positive case, to check if it comes out to be ultimate, that is whether some more optimizations are needed or not.

At last, after having completed the optimization, both the functional testing and the complete performance testing are to be repeated, to guarantee that the new optimized version of the software hasn't worsened either its correctness nor its general performance.

This book is about only three of the above stages:

- Stage 2, specifically to the usage of the C++ language, in chapter "Writing efficient code".
- Some general techniques for the stage 4.3, with examples in C++, in chapter "General optimization techniques".
- Stage 4.4, specifically to the usage of the C++ language, in chapter "Code optimization".

2.1 Conventions

By **object**¹⁰, it is meant an allocated region of memory. In particular, a piece of data associated to a variable of a fundamental type (like `bool`, `double`, `unsigned long`, or a pointer) is an object, as it is such the data structure associated to an instance of a class. With every variable an object is associated, whose size is given by the `sizeof` C++ operator, but an object could have no variable associated with it, or several variables associated with it. For example, a pointer is an object, but it can point to another object; this pointed object is not associated with any variable. On the other hand, in the following code, both the variable `a` and the variable `b` are associated with the same object:

```
int a;
int& b = a;
```

Arrays, structs, and class instances are objects which, if not empty, contain sub-objects. Therefore, they will be called *aggregate* objects.

We say that an object **owns** another object when the destruction of the former object causes the destruction of the latter. For example, a non-empty `vector` object typically contains a pointer to a buffer containing all the elements; the destruction of the `vector` causes the destruction of such buffer, and therefore we say that this buffer is *owned* by the `vector` object.

Some optimizations are useful only for short data sequences, others for longer sequences. Later on, the following classification will be used for objects sizes:

- **Tiny:** No more than 8 bytes. It fits in one or two 32-bit registers or in one 64-bit register.
- **Small:** More than 8 bytes, but no more than 64 bytes. It doesn't fit in a processor register, but it fits in a processor data cache line, and it can be wholly referenced by very compact machine instructions using an offset from the start address.
- **Medium:** More than 64 bytes, but no more than 4096 bytes. It doesn't fit in a processor data cache line, and it cannot be wholly referenced by compact machine instructions, but it fits in the processor first-level data cache, it fits in a virtual memory page, and it fits in a mass storage cluster of blocks.
- **Large:** More than 4096 bytes. It doesn't fit in the processor first-level data cache, it doesn't fit in a single virtual memory page, and it doesn't fit in a single mass storage cluster of blocks.

For example, an array of `doubles` is considered *tiny* only if it contains exactly one element, *small* if it has 2 to 8 elements, *medium* if it has 9 to 512 numbers, *large* if it has more than 512 of them.

Because there are very different hardware architectures, the given numbers are only an indication. Though, such numbers are rather realistic, and can be taken as serious criteria to develop software covering the main architectures in a rather efficient way.

it:Ottimizzare C++/Ciclo di vita dell'ottimizzazione¹¹

¹⁰ http://en.wikipedia.org/wiki/Object%20%28computer_science%29

¹¹ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FCiclo%20di%20vita%20dell%20%19ottimizzazione>

3 Writing efficient code

In this chapter, guidelines are presented to program in C++ avoiding inefficient operations and preparing the source code for a possible successive optimization stage, but without jeopardizing code safety or maintainability.

Such guidelines could give no performance advantage, but quite probably they don't give any disadvantage either, and therefore you can apply them without worrying of their impact on performance. You are advised to accustom yourself to follow always such guidelines, even in code portions that have no particular efficiency requirements.

1. Performance improving features¹
2. Performance worsening features²
3. Constructions and destructions³
4. Allocations and deallocations⁴
5. Memory access⁵
6. Thread usage⁶

it:Ottimizzare C++/Scrivere codice C++ efficiente⁷

1 <http://en.wikibooks.org/wiki/%2FPerformance%20improving%20features>
2 <http://en.wikibooks.org/wiki/%2FPerformance%20worsening%20features>
3 <http://en.wikibooks.org/wiki/%2FConstructions%20and%20destructions>
4 <http://en.wikibooks.org/wiki/%2FAllocations%20and%20deallocations>
5 <http://en.wikibooks.org/wiki/%2FMemory%20access>
6 <http://en.wikibooks.org/wiki/%2FThread%20usage>
7 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente>

4 Performance improving features

Some features of the C++ language, if properly used, allow to increase the speed of the resulting software.

In this section guidelines to exploit such features are presented.

4.0.1 The most efficient types

When defining an object to store an integer number, use the `int` or the `unsigned int` type, except when a longer type is needed; when defining an object to store a character, use the `char` type, except when the `wchar_t` type is needed; and when defining an object to store a floating point number, use the `double` type, except when the `long double` type is needed. If the resulting aggregate object is of medium or large size, replace all the integer types with the smallest integer type that is long enough to contain it, but without using bit-fields, and replace the floating point types with the `float` type, except when greater precision is needed.

The `int` and `unsigned int` types are by definition the most efficient ones on any platform.

The `double` type is two to three times less efficient than the `float` type, but it has greater precision.

Some processor types handle more efficiently `signed char` objects, while others handle more efficiently `unsigned char` objects. Therefore, both in C and in C++, the `char` type, different from `signed char` type, has been introduced as the most efficient character type for the target processor.

The `char` type can contain only small character sets; typically up to a maximum of 255 distinct characters. To handle bigger character sets, you should use the `wchar_t` type, although that is obviously less efficient.

In case of numbers contained in a medium or large aggregate object, or in a collection that will be probably be of medium or large size, it is better to minimize the size in bytes of the aggregate object or collection. This can be done by replacing the `ints` with `shorts` or `signed chars`, replacing the `unsigned ints` with `unsigned shorts` or `unsigned chars`, and replacing the `doubles` with `floats`. For example, to store an integer number with a range of 0 to 1000, you can use an `unsigned short`, while to store an integer number with a range of -100 to 100, you can use a `signed char`.

Bit-fields could contribute to minimize the size in bytes of the aggregate object, but their handling causes a slow down that could be counterproductive; therefore, postpone their introduction to the optimization stage.

4.0.2 Function-objects

Instead of passing a function pointer as an argument to a function, pass a function-object¹ (or, if using the C++0x² standard, a lambda expression).

For example, if you have the following array of structures:

```
struct S {  
    int a, b;  
};  
S arr[n_items];
```

... and you want to sort it by the `b` field, you could define the following comparison function:

```
bool compare(const S& s1, const S& s2) {  
    return s1.b < s2.b;  
}
```

... and pass it to the standard sort algorithm:

```
std::sort(arr, arr + n_items, compare);
```

However, it is probably more efficient to define the following function-object class (aka *functor*):

```
struct Comparator {  
    bool operator()(const S& s1, const S& s2) const {  
        return s1.b < s2.b;  
    }  
};
```

... and pass a temporary instance of it to the standard sort algorithm:

```
std::sort(arr, arr + n_items, Comparator());
```

Function-objects functions are usually expanded *inline*, and therefore they are just as efficient as in-place code, while functions passed by pointers are rarely inlined. Lambda expressions are implemented as function-objects, so they have the same performance.

4.0.3 `qsort` and `bsearch` functions

Instead of the `qsort` and `bsearch` C standard library functions, use the `std::sort` and `std::lower_bound` C++ standard library functions.

The former two functions require a pointer to function as argument, as the latter two may get a function-object argument (or, using the C++0x standard, a lambda expression). Pointers to function often are not expanded inline, and therefore they are less efficient than function-objects that are almost always inlined.

¹ <http://en.wikipedia.org/wiki/Function%20object>

² <http://en.wikipedia.org/wiki/C%2B%2B0x>

4.0.4 Encapsulated collections

Encapsulate in specific classes the collections that are accessible from several compilation units, to make easily interchangeable the implementation.

At design time, it is difficult to decide which data structure will have optimal performance in the actual use of the software. At optimization time, you can measure whether by changing the type of a container, for example passing from `std::vector` to `std::list`, you can improve the performance. Though, such a change of implementation causes the change of most source code that uses directly the collection whose type has been changed.

If a collection is private to one compilation unit, such change will impact only the source code contained in this unit, and therefore it is not necessary to encapsulate that collection. If instead that collection isn't private, that is it is directly accessible from other compilation units, in the future there could be a huge amount of code to change for such a collection type change. Therefore, to make feasible such optimization, you should encapsulate that collection in a class whose interface does not change when the container implementation is changed.

STL containers already pursue this principle, but some operations are still available only for some containers (like `operator[]`, existing for `std::vector`, but not for `std::list`).

4.0.5 STL containers usage

When using an STL container, if several equivalent expressions have the same performance, choose the more general expression.

For instance, call `a.empty()` instead of `a.size() == 0`, call `iter != a.end()` instead of `iter < a.end()`, and call `distance(iter1, iter2)` instead of `iter2 - iter1`. The former expressions are valid for every container kind, while the latter are valid only for some kinds, and the former are no less efficient than the latter.

Unfortunately, it is not always possible to write code that is equally correct and efficient for every type of container. Nevertheless, decreasing the number of statements that are dependent on the container type will decrease the number of statements have to be changed if the type of the container is later changed.

4.0.6 Choice of the default container

When choosing a variable-length container, if in doubt, choose a `vector`.

For set up to 8 elements, `vector` is the most efficient variable-length container for any operation.

For larger collections, other containers may become more efficient for certain operations, but `vector` remains the one that has the least space overhead (as long as there is no excess capacity), and the greatest locality of reference.

4.0.7 Inlined functions

If you use compilers that allow whole program optimization and the automatic inline expansion of functions, use such options, and do not declare any functions `inline`. If such compiler features are not available, declare, in header files only, functions `inline` that contain no more than three lines of code and have no loops.

Functions that are expanded inline avoid the overhead of function calls, which grows as the number of function arguments. In addition, since inline code is near caller code, it has better locality of reference. And lastly, as the intermediate code generated by the compiler for inlined functions is merged with the caller code, it can be more easily optimized by the compiler.

Expanding inline a tiny function, such as a function containing only a simple assignment or a simple `return` statement, can result in a decrease in the size of the generated machine code.

Conversely, every time a function containing substantial code is inlined the machine code is duplicated, and therefore the total size of the program is increased.

Inlined code is more difficult to profile. If a non-inlined function is a bottleneck, it can be found by the profiler. But if that function is inlined in all the places where it is called, its running time is scattered among many functions and the bottleneck situation cannot be detected by a profiler.

For functions containing substantial amounts of code, only the performance critical ones should be declared `inline` at the optimization stage.

4.0.8 Symbols representation

To represent internal symbols, use enumerations instead of strings.

For example, instead of the following code:

```
const char* directions[] = { "North", "South", "East", "West" };
```

use the following code:

```
enum directions { North, South, East, West };
```

An enumeration is implemented as an integer. Compared to an integer, a string occupies more space and is slower to copy and compare. (In addition, using strings instead of integers for representation of internal state may introduce string comparison errors in code that deals with multiple locales.)

4.0.9 `if` and `switch` statements

If you have to compare an integer value with a set of constant values, instead of a sequence of `if` statements, use a `switch` statement.

For example, instead of the following code:

```
if (a[i] == 1) f();
else if (a[i] == 2) g();
else if (a[i] == 5) h();
```


write the following code:

```
switch (a[i]) {
case 1: f(); break;
case 2: g(); break;
case 5: h(); break;
}
```

Compilers may exploit the regularity of `switch` statements to apply some optimizations, in particular if the guideline "Case values for `switch` statements" in this section is applied.

4.0.10 Case values of `switch` statements

As constants for `switch` statements cases, use compact sequences of values, that is sequences with no gaps or with few small gaps.

Optimizing compilers, when compiling a `switch` statement whose case values comprise most of the values in an integer interval, instead of generating a sequence of `if` statements, generate a *jump-table*, that is an array of the start addresses of the code for every case, and when executing the `switch` statement, they use the table to jump to the code associated to the case number.

For example, the following C++ code:

```
switch (i) {
case 10:
case 13:
    func_a();
    break;
case 11:
    func_b();
    break;
}
```

probably generates machine code corresponding to the following pseudo-code:

```
// N.B.: This is not C++ code
static address jump_table[] = { case_a, case_b, end, case_a };
unsigned int index = i - 10;
if (index > 3) goto end;
goto jump_table[index];
case_a: func_a(); goto end;
case_b: func_b();
end:
```

Instead, the following C++ code:

```
switch (i) {
case 100:
case 130:
    func_a();
    break;
case 110:
    func_b();
    break;
}
```

probably generates machine code corresponding to the following code:

```
if (i == 100) goto case_a;
if (i == 130) goto case_a;
if (i == 110) goto case_b;
goto end;
case_a: func_a(); goto end;
case_b: func_b();
end:
```

For so few cases, probably there is little difference between the two situations, but as the case count increases, the former code becomes more efficient, as it performs only one *computed goto* instead of a sequence of branches.

4.0.11 Cases order in switch statement

In switch statements, put the most typical cases before.

If the compiler does not use a *jump-table*, the cases are evaluated in order of appearance; therefore, fewer comparisons are performed for the more frequent cases.

4.0.12 Grouping function arguments

Instead of calling a function in a loop which uses more variables than there are registers, use a function that is passed a struct or object.

For example, instead of the following code:

```
for (int i = 0; i < 1000; ++i) {
    f(i, a1, a2, a3, a4, a5, a6, a7, a8);
}
```

write the following:

```
struct {
    int i;
    type a1, a2, a3, a4, a5, a6, a7, a8;
} s;
s.a1 = a1; s.a2 = a2; s.a3 = a3; s.a4 = a4;
s.a5 = a5; s.a6 = a6; s.a7 = a7; s.a8 = a8;
for (int i = 0; i < 1000; ++i) {
    s.i = i;
    f(s);
}
```

If all the function's arguments can be placed directly into a processor's registers, the arguments can be passed and manipulated quickly. If there are more arguments than registers, all the arguments or the arguments which couldn't be placed into registers may require being pushed onto a stack at the start of every function call and popped off a stack at the end of every function call, even if the arguments have not changed. If a structure or object is passed, a register may be used and after initialization of the structure or object, only the parts of the structure or object which have changed between successive calls must be updated.

4.0.13 Use of containers member functions

To search for an element in a container, use a member function of the container, instead of an STL algorithm.

If such a specific member function has been created, when a generic STL algorithm was already existing, it is only because such member functions is more efficient.

For example, to search a `std::set` object, you can use the `std::find` generic algorithm, or the `std::set::find` member function; but the former has linear complexity ($O(n)$), while the latter has logarithmic complexity ($O(\log(n))$).

4.0.14 Search in sorted sequences

To search a sorted sequence, use the `std::lower_bound`, `std::upper_bound`, `std::equal_range`, or `std::binary_search` generic algorithms.

Given that all the cited algorithms use a logarithmic complexity ($O(\log(n))$) binary search, they are faster than the `std::find` algorithm, that uses the linear complexity ($O(n)$) sequential scan.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Costrutti che migliorano le prestazioni³

4.0.15 `static` member functions

In every class, declare every member function that does not access the non-`static` members of the class as `static`.

In other words, declare all the member functions that you can as `static`.

In this way, the implicit `this` argument is not passed.

³ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FCostrutti%20che%20migliorano%20le%20prestazioni>

5 Performance worsening features

With respect to C language, C++ has some features, that worsen efficiency if used inappropriately.

Some of them are anyway rather efficient, and therefore you can use them liberally when they are useful, but you should also avoid to pay their cost when you don't need them.

Other features are instead quite inefficient, and so you should use them sparingly.

In this section, guidelines are presented to avoid the costs of C++ features that worsen performance.

5.0.16 The `throw` operator

Call the `throw` operator only when you want to notify a user of the failure of the current command.

The raising of an exception has a very high cost compared to a function call. It requires thousands of processor cycles. If you perform this operation only when a message is displayed on the user interface or written into a log file, you have the guarantee that it will not be performed too often without notice.

Instead, if exception raising is performed for algorithmic purposes, even if such operation is initially thought to be performed rarely, it may end up being performed too frequently.

5.0.17 `virtual` member functions

Define the destructor as `virtual` if and only if the class contains at least one other `virtual` member function or if the class might be derived from. Except for destructors, do not declare functions as `virtual` unless you intend to override them.

Classes that have some `virtual` member functions occupy more storage space than those classes without them. Instances of classes having at least one `virtual` member function occupy more space (typically, a pointer and possibly some padding) and their construction requires more time (typically, to set the pointer) than the instances of classes without `virtual` member functions.

In addition, every `virtual` member function has a slower call time than an identical non-`virtual` member function.

5.0.18 `virtual` inheritance

Use `virtual` inheritance only when several classes must share the representation of a common base class.

For example, consider the following class definitions:

```
class A { ... };  
class B1: public A { ... };  
class B2: public A { ... };  
class C: public B1, public B2 { ... };
```

With such definitions, every C class object contains two distinct class A objects, one inherited from the B1 base class, and the other from the B2 class.

This is not a problem if class A has no non-`static` member variables.

If instead such class A contains some member variables, and you mean that such member variables are unique for every class C instance, you must use `virtual` inheritance, in the following way:

```
class A { ... };  
class B1: virtual public A { ... };  
class B2: virtual public A { ... };  
class C: public B1, public B2 { ... };
```

This situation is the only case when `virtual` derivation is necessary.

The member functions of the class A are somewhat slower to call on a C class object if `virtual` inheritance is used.

5.0.19 Templates of polymorphic classes

Do not define templates of polymorphic classes.

In other words, don't use the "template" and the "virtual" keywords in the same class definition.

Class templates, every time they are instantiated, generate a copy of all the member functions used, and if such classes contain virtual functions, even their *vtable* and *RTTI* data is replicated. This data bloats the code.

5.0.20 Use of automatic deallocators

Use a memory manager based on garbage-collection¹ or a kind of reference-counted² smart-pointer³ (like `shared_ptr` in the Boost⁴ library) only if you can prove its expediency for the specific case.

Garbage collection, or automatic reclamation of unreferenced memory, allows the programmer to leave out calls for memory deallocation and prevents memory leaks⁵. However, one must implement garbage collection through a non-standard library since these features are not included in C++. In addition, garbage collection typically causes slower execution than that of explicit deallocation (when the `delete` operator is explicitly called). In addition, when the garbage collector runs the rest of the program is stopped, therefore increasing the variance of the time taken by the commands.

1 <http://en.wikipedia.org/wiki/Garbage%20collection>

2 <http://en.wikipedia.org/wiki/Reference%20counting>

3 <http://en.wikipedia.org/wiki/Smart%20pointer>

4 <http://www.boost.org/>

5 <http://en.wikipedia.org/wiki/Memory%20leak>

The C++98 standard library contains only one smart-pointer kind, `auto_ptr`, that is quite efficient. Other smart-pointers provided by non-standard libraries, like Boost, will be provided by the C++0x standard library. The smart-pointer is based on reference-count but is less efficient than a simple pointer. For example, `shared_ptr` is the standard Boost smart pointer. However, if the thread-safe version of Boost is used, the library has very poor performance for creation, destruction and copying of these pointers since it must guarantee mutual exclusion for these operations.

Usually, you should try to assign every dynamically allocated object to an owner at design time. When such an assignment is difficult, for example if several objects tend to bounce the responsibility to destroy the object, it becomes expedient to use a reference-counted smart-pointer to handle the object.

5.0.21 The `volatile` modifier

Define `volatile` only those variables that are changed asynchronously by hardware devices.

The usage of the `volatile` modifier prevents the compiler to allocate a variable in a register, even for a short time. This guarantees that all the devices *see* the same variable, but slows down much the operations that access this variable.

`volatile` does **not** guarantee that other threads will see the same values, as it does not force the compiler to generate the necessary memory barrier and lock instructions. Therefore read and write accesses to a volatile value may be made out of order even on the same CPU (important in the case of interrupts and signals) and especially out of order across the memory cache and bus to other CPUs. You **must** use a proper thread or atomic memory API or write machine code to guarantee the proper order of operations for safe threading.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Costrutti che peggiorano le prestazioni⁶

⁶ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FCostrutti%20che%20peggiorano%20le%20prestazioni>

6 Constructions and destructions

Construction and destruction of an object requires time, especially if the object owns other objects.

This section will provide guidelines to decrease the number of object constructions and corresponding destructions.

6.0.22 Variable scope

Declare variables as late as possible.

To do so, the programmer must declare all variables in the most local scope. By doing so, the variable is not constructed or destructed if that scope is never reached. To postpone the declaration as far as possible inside a scope causes that if before such declaration there is an early exit, using a `return` or `break` or `continue` statement, the object associated to the variable is not constructed nor destructed.

In addition, often at the beginning of a routine you haven't yet an appropriate value to initialize the object associated to the variable, and therefore you must initialize it with a default value, and then assign an appropriate value to it. Instead, if you declare such variable when you have available an appropriate value, you can initialize the object using such value, without needing a successive assignment, as advised by the guideline "Initializations" in this section.

6.0.23 Initializations

Use initializations instead of assignments. In particular, in constructors, use initialization lists.

For example, instead of writing:

```
string s;  
...  
s = "abc"
```

write:

```
string s("abc");
```

Even if an instance of a class is not explicitly initialized, it is anyway automatically initialized by the default constructor.

To call the default constructor followed by an assignment with a value may be less efficient than to call only a constructor with the same value.

6.0.24 Increment/decrement operators

Use prefix increment (++) or decrement (--) operators instead of the corresponding postfix operators, if the expression value is not used.

If the incremented object is a primitive type, there is no difference between prefix and postfix operators. However, if it is a composite object, the postfix operator causes the creation of a temporary object, while the prefix operator does not.

Because every object that is a primitive type may become a composite object in the future, it is better to use the prefix operator whenever possible, especially when writing generic (templated) code that operates on iterators.

Use the postfix operator only when the variable is in a larger expression and must be incremented only after the expression is evaluated.

6.0.25 Assignment composite operators

Use the assignment composite operators (like in `a += b`) instead of simple operators combined with assignment operators (like in `a = a + b`).

For example, instead of the following code:

```
string s1("abc");
string s2 = s1 + " " + s1;
```

write the following code:

```
string s1("abc");
string s2 = s1;
s2 += " ";
s2 += s1;
```

Typically a simple operator creates a temporary object. In the example, the operator `+` creates temporary strings, whose creation and destruction require much time.

On the contrary, the equivalent code using the `+=` operator do not create temporary objects.

6.0.26 Function argument passing

When you pass an object `x` of type `T` as argument to a function, use the following criterion:

- **If `x` is a input-only argument,**
 - **if `x` may be null,**
 - **pass it by pointer to constant (`const T* x`),**
 - **otherwise, if `T` is a fundamental type or an iterator or a function-object,**
 - **pass it by value (`T x`) or by constant value (`const T x`),**
 - **otherwise,**
 - **pass it by reference to constant (`const T& x`),**
 - **otherwise, i.e. if `x` is an output-only or input/output argument,**
 - **if `x` may be null,**

- **pass it by pointer to non-constant (`T* x`),**
- **otherwise,**
- **pass it by reference to non-constant (`T& x`).**

Pass by reference is more efficient than pass by pointer as it facilitates variable elimination by the compiler, and as the callee hasn't to check if the reference is valid or null; though, the pointer has the advantage of being able to represent a null value, and it is more efficient to pass just a pointer, than to pass a reference to a possibly dummy object and a boolean indicating whether the reference is valid.

For objects that may be contained in one or two registers, pass by value is more efficient or equally efficient than pass by reference, as such objects may be contained in registers and don't have indirection levels; therefore, this is the fastest way to pass surely tiny objects, as the fundamental types, the iterators and the function-objects. For objects that are larger than a couple of registers, pass by reference is more efficient than pass by value, as pass by value causes the copy into the stack of such objects.

A composite object that is fast to copy could be efficiently passed by value, but, except it is an iterator or a function-object, for which it is assumed the efficiency of the copy, this technique is risky, as in the future the object could become slower to copy. For example, if an object of class `Point` contains only two `float`s, it could be efficiently passed by value; but if in the future a third `float` is added, or if the two `float`s become two `doubles`, it could become more efficient pass by reference.

6.0.27 `explicit` declaration

Declare `explicit` all the constructors that may receive only one argument, except the copy constructors of concrete classes.

Non-`explicit` constructors may be called automatically by the compiler when it performs an automatic type conversion. The execution of such constructors may take much time.

If such conversion is made compulsorily explicit, and it new class name is not specified in the code, the compiler could choose another overloaded function, avoiding to call the costly constructor, or generate an error, so forcing the programmer to choose another way to avoid the constructor call.

For copy constructors of concrete classes a distinction must be made, to allow their pass by value. For abstract classes, even copy constructors may be declared `explicit`, as, by definitions, abstract classes cannot be instantiated, and so objects of such type should never be passed by value.

6.0.28 Conversion operators

Declare conversion operators only to keep compatibility with an obsolete library (in C++0x, declare them `explicit`).

Conversion operators allow implicit conversions, and so incur in the same problem of implicit constructors, described in the guideline "`explicit` declaration" in this section.

If such conversions are needed, provide instead an equivalent member function, as it may be called only explicitly.

The only acceptable remaining usage for conversion operators is when a new library must coexist with an older similar library. In such case, it may be convenient to have operators that convert

automatically objects from the old library into the corresponding types of the new library, and vice versa.

6.0.29 The *Pimpl* idiom

Use the *Pimpl*¹ idiom only when you want to make the rest of the program independent from the implementation of a class.

The *Pimpl* idiom (meaning **P**ointer to **impl**ementation) consists in storing in the object only a pointer to a data structure containing all the useful information about such object.

The main advantage of such idiom is that it speeds up incremental compilation of the code, that is it makes less likely that a small change in source code causes the need to recompile a large number of code lines.

Such idiom allows also to speed up some operations, like the `swap` of two objects, but in general it slows down every access to the object data because of the added indirection level, and causes an added memory allocation for every creation or copy of such object. Therefore it shouldn't be used for classes whose public member functions are called frequently.

6.0.30 Iterators and function objects

Ensure that custom iterators and function objects are tiny and do not allocate dynamic memory.

STL algorithms pass such objects by value. Therefore, if their copy is not extremely efficient, STL algorithms are slowed down.

If an iterator or function object for some reason needs elaborate internal state, allocate it dynamically and use a shared pointer. For example, say you want to implement an STL-compliant 32-bit Random Number Generator² on top of the Linux/OpenBSD `/dev/urandom` device:

```
#include <boost/shared_ptr.hpp>
#include <fstream>

class urandom32 {
    boost::shared_ptr<std::ifstream> device;

public:
    urandom32() : device(new std::ifstream("/dev/urandom")) { }

    uint32_t operator() ()
    {
        uint32_t r;
        device->read(reinterpret_cast<char *>(&r),
sizeof(uint32_t));
        return r;
    }
};
```

1 <http://en.wikipedia.org/wiki/Opaque%20pointer>

2 <http://www.sgi.com/tech/stl/RandomNumberGenerator.html>

In this case, the use of a pointer was actually necessary because the `ifstream` class is non-copyable: its copy constructor is declared `private`. This example uses the `boost::shared_ptr` smart pointer; for more speed, intrusive reference counting could be used.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Costruzioni e distruzioni³

³ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FCostruzioni%20e%20distruzioni>

7 Allocations and deallocations

Dynamic memory allocation and deallocation are very slow operations, compared with automatic memory allocation and deallocation. In other words, the heap is much slower than the stack.

In addition, such kind of allocation causes a per-allocation overhead, causes virtual memory fragmentation, and causes bad data locality of reference, with ensuing bad usage of both processor data cache and virtual memory space.

Dynamic memory allocation/deallocation was performed in C language using the `malloc` and `free` standard library functions. In C++, although such functions are still available, usually for such purpose the `new`, `new[]`, `delete`, and `delete[]` operators are used.

Obviously, a way to decrease the allocation count is to decrease the number of constructed objects, and therefore the section "Constructions and destructions" in this chapter is indirectly useful also for the purpose of this section.

Yet, here guidelines are presented to decrease the allocations count for a given number of `new` operator calls.

7.0.31 Fixed length arrays

If a static or non-large array has compile-time constant length, instead of a vector object, use a C-language array or an `array` object from the Boost¹ library.

`vectors` stores the data in a dynamically allocated buffer, while the other solutions allocate data inside the object itself. This avoids repeated allocations/deallocations of dynamic memory and favor data locality.

If the array is large, such advantages are diminished, and instead it becomes more important to avoid using too much stack space.

7.0.32 Allocating many small objects

If you have to allocate many objects of the same size, use a block allocator.

A *block allocator*² (aka *pool allocator*) allocates medium to large memory blocks, and provides the service to allocate/deallocate fixed size smaller blocks. It allows high allocation/deallocation speed, low memory fragmentation, an efficient usage of data caches and of the virtual memory.

1 <http://www.boost.org>

2 <http://en.wikipedia.org/wiki/Memory%20pool>

In particular, an allocator of this kind can greatly improve the performance of the `std::list`, `std::set`, `std::multiset`, `std::map`, and `std::multimap` standard containers.

If your standard library implementation does not already use a block allocator for such containers, you should get one and specify it as template parameter of the instances of such containers templates. Boost³ provides two customizable block allocators, `pool_allocator` and `fast_pool_allocator`⁴. Other pool allocator libraries can be found on the World Wide Web. Always measure first to find the fastest allocator for the job at hand.

7.0.33 Appending elements to a collection

When you have to append elements to a collection, use `push_back` to append a single element, use `insert` to append a sequence of elements, and use `back_inserter` to cause an STL algorithm append elements to a sequence.

The `push_back` functions guarantees an amortized linear time, as, in case of `vectors`, it increases exponentially the capacity.

The `back_inserter` class calls internally the `push_back` function.

The `insert` function allows to insert a whole sequence in an optimized way, and therefore a single call to it is faster than many calls to `push_back`.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Allocazioni e deallocazioni⁵

3 <http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FLibraries%2FBoost>
4 http://www.boost.org/doc/libs/1_38_0/libs/pool/doc/interfaces/pool_alloc.html
5 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FAllocazioni%20e%20deallocazioni>

8 Memory access

This section presents guidelines to improve main memory access performance, by exploiting the features of the processor caches and of the secondary memory swapping by the operating system virtual memory manager.

8.0.34 Memory access order

Access memory in increasing addresses order. In particular:

- scan arrays in increasing order;
- scan multidimensional arrays using the rightmost index for innermost loops;
- in class constructors and in assignment operators (`operator=`), access member variables in the order of declaration.

Data caches optimize memory access in increasing sequential order.

When a multidimensional array is scanned, the innermost loop should iterate on the last index, the innermost-but-one loop should iterate on the last-but-one index, and so on. In such a way, it is guaranteed that array cells are processed in the same order in which they are arranged in memory. For example, the following code is optimized:

```
float a[num_levels][num_rows][num_columns];
for (int lev = 0; lev < num_levels; ++lev) {
    for (int r = 0; r < num_rows; ++r) {
        for (int c = 0; c < num_columns; ++c) {
            a[lev][r][c] += 1;
        }
    }
}
```

8.0.35 Memory alignment

Keep the compiler default memory alignment.

Compilers use by default an alignment criterion for fundamental types, for which objects may have only memory addresses that are a multiple of particular factors. Such criterion guarantees top performance, but it may add *padding*s (or *holes*) between successive objects.

If it is necessary to avoid such padding for some structures, use the *pragma* directive only around such structure definitions.

8.0.36 Grouping functions in compilation units

Define in the same compilation unit all the member functions of a class, all the `friend` functions of such class, and all the member functions of `friend` classes of such class, except when the resulting file become unwieldy for its size.

In such a way, both the machine code resulting by the compilation of such functions and the static data defined in such classes and functions will have addresses near each other; in addition, even compilers that do not perform whole program optimization may optimize the calls among these functions.

8.0.37 Grouping variables in compilation units

Define every global variable in the compilation unit in which it is used more often.

In such a way, such variables will have addresses near to each other and to the static variables defined in such compilation units; in addition, even compilers that do not perform whole program optimization may optimize the access to such variables from the functions that use them more often.

8.0.38 Private functions and variables in compilation units

Declare in an anonymous namespace the variables and functions that are global to compilation unit, but not used by other compilation units.

In C language and also in C++, such variables and functions may be declared `static`. Though, in modern C++, the use of `static` global variables and functions is not recommended, and should be replaced by variables and functions declared in an anonymous namespace.

In both cases, the compiler is notified that such identifiers will never be used by other compilation units. This allows the compilers that do not perform whole program optimization to optimize the usage of such variables and functions.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Accesso alla memoria¹

¹ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FAccesso%20alla%20memoria>

9 Thread usage

9.0.39 Worker thread

Every time in an interactive application you must perform an operation that can take more than few seconds, assign this operation to a specific worker thread, having less than normal priority.

In such a way, the main thread is ready to handle other user commands. By assigning to the worker thread a less than normal priority, the user interface remains almost just as fast.

Strictly speaking, this guidelines does not improves the speed of the application, but only its responsiveness. Though, this is perceived by users as a speed improvement.

9.0.40 Multiple worker threads

In a multicore system, if you can split an operation in several threads, use as many worker threads as are the processor cores.

In such a way, every core can process a worker thread. If the worker threads are more than the cores, there would be a lot of thread switching, and this would slow down the operation. The main thread does not slow down the operation, as it is almost inactive.

9.0.41 Use of multi-threaded libraries

If you are developing a single-threaded application, don't use libraries designed only for multi-threaded applications.

The techniques to make thread-safe a library may have memory and time overhead. If you don't use threads, avoid to pay their cost.

9.0.42 Creation of multi-threaded libraries

If you are developing a library, handle correctly the case it is used by multi-threaded applications, but optimize also the case it is used by single-threaded applications.

The techniques to make thread-safe a library may have memory and time overhead. If the users of your library don't use threads, avoid forcing your users to pay the cost of threads.

9.0.43 Mutual exclusion

Use mutual exclusion primitives only when several threads access concurrently the same data, and at least one of the accesses is for writing.

Mutual exclusion primitives have a overhead.

If you are sure that in a given period no thread is writing in a memory area, there is no need to synchronize read accesses for such area.

it:Ottimizzare C++/Scrivere codice C++ efficiente/Usare thread¹

¹ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FScrivere%20codice%20C%2B%2B%20efficiente%2FUsare%20thread>

10 General optimization techniques

In this section we present some of most common techniques for algorithmic optimization. These techniques should be mostly independent from any specific programming language, software or hardware platform. When optimizing, always start by considering different algorithms before resorting to lower-level optimizations to retain generality, maintainability and portability of your code.

Some of the proposed techniques will have an implementation in C++.

1. Input/Output¹
2. Memoization²
3. Sorting³
4. Other techniques⁴

it:Ottimizzare C++/Tecniche generali di ottimizzazione⁵

1 Chapter 11 on page 37

2 Chapter 12 on page 45

3 Chapter 13 on page 49

4 Chapter 14 on page 53

5 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FTecniche%20generali%20di%20ottimizzazione>

11 Output

11.0.44 Binary format

Instead of storing data in text mode, store them in a binary format.

On average, binary numbers occupy less space than formatted numbers, and so it is faster to transfer them from memory to disk or vice versa. Also, if the data is transferred in the same format used by the processor there is no need of costly conversions from text format to binary format or vice versa.

Some disadvantages of using a binary format are that data is not human-readable and that the format may be dependent on the processor architecture.

11.0.45 Open files

Instead of opening and closing an often needed file every time you access it, open it only the first time you access it, and close it when you are finished using it.

To close and reopen a disk file takes a variable time, but about the same time to read 15 to 20 KB from the disk cache.

Therefore, if you need to access a file often, you can avoid this overhead by opening the file only one time before accessing it, keeping it open by hoisting its handle wrapper to an external scope, and closing it when you are done.

11.0.46 I/O buffers

Instead of doing many I/O operations on single small or tiny objects, do I/O operations on a 4 KB buffer containing many objects.

Even if the run-time support I/O operations are buffered, the overhead of many I/O functions costs more than copying the objects into a buffer.

Larger buffers do not have a good locality of reference.

11.0.47 Memory-mapped file

Except in a critical section of a real-time system, if you need to access most parts of a binary file in a non-sequential fashion, instead of accessing it repeatedly with *seek* operations, or loading it all in an application buffer, use a memory-mapped file¹, if your operating system provides such feature.

¹ <http://en.wikipedia.org/wiki/Memory-mapped%20file>

When you have to access most parts of a binary file in a non-sequential fashion, there are two standard alternative techniques:

- Open the file without reading its contents; and every time a data is demanded, jump to the data position using a file positioning operation (aka *seek*), and read that data from the file.
- Allocate a buffer as large as the whole file, open the file, read its contents into the buffer, close the file; and every time a data is demanded, search the buffer for it.

Using a memory-mapped file, with respect to the first technique, every positioning operation is replaced by a simple pointer assignment, and every read operation is replaced by a simple memory-to-memory copy. Even assuming that the data is already in disk cache, both memory-mapped files operations are much faster than the corresponding file operations, as the latter require as many system calls.

With respect to the technique of pre-loading the whole file into a buffer, using a memory-mapped file has the following advantages:

- When file reading system calls are used, data is usually transferred first into the disk cache and then in the process memory, while using a memory-mapped file the system buffer containing the data loaded from disk is directly accessed, thus saving both a copy operation and the disk cache space. The situation is analogous for output operations.
- When reading the whole file, the program is stuck for a significant time period, while using a memory-mapped file such time period is scattered through the processing, as long as the file is accessed.
- If some sessions need only a small part of the file, a memory-mapped file loads only those parts.
- If several processes have to load in memory the same file, the memory space is allocated for every process, while using a memory-mapped file the operating system keeps in memory a single copy of the data, shared by all the processes.
- When memory is scarce, the operating system has to write out to the swap disk area even the parts of the buffer that haven't been changed, while the unchanged pages of a memory-mapped file are just discarded.

Yet, usage of memory-mapped files is not appropriate in a critical portion of a real-time system, as access to data has a latency that depends on the fact that the data has already been loaded in system memory or is still only on disk.

Strictly speaking, this is a technique dependent on the software platform, as the memory-mapped file feature is not part of C++ standard library nor of all operating systems. Though, given that such feature exists in all the main operating systems that support virtual memory, this technique is of wide applicability.

Here are two classes that encapsulate the access to a file through a memory-mapped file, followed by a small program demonstrating the usage of such classes. They are usable both from Posix operating systems (like Unix, Linux, and Mac OS X) and from Microsoft Windows. The `MemoryFile` class allows both to write and to read a file, and also to change its size. The `InputMemoryFile` class allows only to read a file, but it is simpler and safer, and therefore it is recommended in case you don't need to change the file contents.

File "memory_file.hpp":

```
#ifndef MEMORY_FILE_HPP
#define MEMORY_FILE_HPP
```



```

#include <cstring> // for size_t

/*
  Read-only memory-mapped file wrapper.
  It handles only files that can be wholly loaded
  into the address space of the process.
  The constructor opens the file, the destructor closes it.
  The "data" function returns a pointer to the beginning of the file,
  if the file has been successfully opened, otherwise it returns 0.
  The "size" function returns the length of the file in bytes,
  if the file has been successfully opened, otherwise it returns 0.
*/
class InputMemoryFile {
public:
    InputMemoryFile(const char *pathname);
    ~InputMemoryFile();
    const char* data() const { return data_; }
    size_t size() const { return size_; }
private:
    const char* data_;
    size_t size_;
#ifdef __unix__
    int file_handle_;
#elif defined(_WIN32)
    typedef void* HANDLE;
    HANDLE file_handle_;
    HANDLE file_mapping_handle_;
#else
    #error Only Posix or Windows systems can use memory-mapped files.
#endif
};

/*
  Read/write memory-mapped file wrapper.
  It handles only files that can be wholly loaded
  into the address space of the process.
  The constructor opens the file, the destructor closes it.
  The "data" function returns a pointer to the beginning of the file,
  if the file has been successfully opened, otherwise it returns 0.
  The "size" function returns the initial length of the file in
  bytes,
  if the file has been successfully opened, otherwise it returns 0.
  Afterwards it returns the size the physical file will get if it is
  closed now.
  The "resize" function changes the number of bytes of the
  significant
  part of the file. The resulting size can be retrieved
  using the "size" function.
  The "reserve" grows the physical file to the specified number of
  bytes.
  The size of the resulting file can be retrieved using "capacity".
  Memory mapped files cannot be shrunk;
  a value smaller than the current capacity is ignored.
  The "capacity()" function return the size the physical file has at
  this time.
  The "flush" function ensure that the disk is updated
  with the data written in memory.
*/
class MemoryFile {
public:
    enum e_open_mode {
        if_exists_fail_if_not_exists_create,
        if_exists_keep_if_dont_exists_fail,
        if_exists_keep_if_dont_exists_create,
        if_exists_truncate_if_not_exists_fail,
        if_exists_truncate_if_not_exists_create,
    };
};

```

```
MemoryFile(const char *pathname, e_open_mode open_mode);
~MemoryFile();
char* data() { return data_; }
void resize(size_t new_size);
void reserve(size_t new_capacity);
size_t size() const { return size_; }
size_t capacity() const { return capacity_; }
bool flush();
private:
char* data_;
size_t size_;
size_t capacity_;
#ifdef __unix__
int file_handle_;
#elif defined(_WIN32)
typedef void * HANDLE;
HANDLE file_handle_;
HANDLE file_mapping_handle_;
#else
#error Only Posix or Windows systems can use memory-mapped files.
#endif
};
#endif
```

File "memory_file.cpp":

```
#include "memory_file.hpp"
#ifdef __unix__
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#elif defined(_WIN32)
#include <windows.h>
#endif

InputMemoryFile::InputMemoryFile(const char *pathname):
    data_(0),
    size_(0),
#ifdef __unix__
    file_handle_(-1)
{
    file_handle_ = ::open(pathname, O_RDONLY);
    if (file_handle_ == -1) return;
    struct stat sbuf;
    if (::fstat(file_handle_, &sbuf) == -1) return;
    data_ = static_cast<const char*> (::mmap(
        0, sbuf.st_size, PROT_READ, MAP_SHARED, file_handle_, 0));
    if (data_ == MAP_FAILED) data_ = 0;
    else size_ = sbuf.st_size;
#elif defined(_WIN32)
    file_handle_(INVALID_HANDLE_VALUE),
    file_mapping_handle_(INVALID_HANDLE_VALUE)
{
    file_handle_ = ::CreateFile(pathname, GENERIC_READ,
        FILE_SHARE_READ, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (file_handle_ == INVALID_HANDLE_VALUE) return;
    file_mapping_handle_ = ::CreateFileMapping(
        file_handle_, 0, PAGE_READONLY, 0, 0, 0);
    if (file_mapping_handle_ == INVALID_HANDLE_VALUE) return;
    data_ = static_cast<char*> (::MapViewOfFile(
        file_mapping_handle_, FILE_MAP_READ, 0, 0, 0));
    if (data_) size_ = ::GetFileSize(file_handle_, 0);
#endif
}

InputMemoryFile::~InputMemoryFile() {
#ifdef __unix__
```

```

        ::munmap(const_cast<char*>(data_), size_);
        ::close(file_handle_);
#elif defined(_WIN32)
        ::UnmapViewOfFile(data_);
        ::CloseHandle(file_mapping_handle_);
        ::CloseHandle(file_handle_);
#endif
}

#include <iostream>
MemoryFile::MemoryFile(const char *pathname, e_open_mode open_mode):
    data_(0),
    size_(0),
#ifdef __unix__
    file_handle_(-1)
{
    int posix_open_mode = O_RDWR;
    switch (open_mode)
    {
        case if_exists_fail_if_not_exists_create:
            posix_open_mode |= O_EXCL | O_CREAT;
            break;
        case if_exists_keep_if_dont_exists_fail:
            break;
        case if_exists_keep_if_dont_exists_create:
            posix_open_mode |= O_CREAT;
            break;
        case if_exists_truncate_if_not_exists_fail:
            posix_open_mode |= O_TRUNC;
            break;
        case if_exists_truncate_if_not_exists_create:
            posix_open_mode |= O_TRUNC | O_CREAT;
            break;
        default: return;
    }
    const size_t min_file_size = 4096;
    file_handle_ = ::open(pathname, posix_open_mode, S_IRUSR |
S_IWUSR | S_IRGRP | S_IROTH);
    if (file_handle_ == -1) return;
    struct stat sbuf;
    if (::fstat(file_handle_, &sbuf) == -1) return;
    size_t initial_file_size = sbuf.st_size;
    size_t adjusted_file_size = initial_file_size == 0 ?
min_file_size : initial_file_size;
    ::ftruncate(file_handle_, adjusted_file_size);
    data_ = static_cast<char*> (::mmap(
        0, adjusted_file_size, PROT_READ | PROT_WRITE, MAP_SHARED,
file_handle_, 0));
    if (data_ == MAP_FAILED) data_ = 0;
    else {
        size_ = initial_file_size;
        capacity_ = adjusted_file_size;
    }
}
#elif defined(_WIN32)
    file_handle_(INVALID_HANDLE_VALUE),
    file_mapping_handle_(INVALID_HANDLE_VALUE)
{
    int windows_open_mode;
    switch (open_mode)
    {
        case if_exists_fail_if_not_exists_create:
            windows_open_mode = CREATE_NEW;
            break;
        case if_exists_keep_if_dont_exists_fail:
            windows_open_mode = OPEN_EXISTING;
            break;
        case if_exists_keep_if_dont_exists_create:
            windows_open_mode = OPEN_ALWAYS;

```

```

        break;
    case if_exists_truncate_if_not_exists_fail:
        windows_open_mode = TRUNCATE_EXISTING;
        break;
    case if_exists_truncate_if_not_exists_create:
        windows_open_mode = CREATE_ALWAYS;
        break;
    default: return;
}
const size_t min_file_size = 4096;
file_handle_ = ::CreateFile(pathname, GENERIC_READ |
GENERIC_WRITE,
    0, 0, windows_open_mode, FILE_ATTRIBUTE_NORMAL, 0);
if (file_handle_ == INVALID_HANDLE_VALUE) return;
size_t initial_file_size = ::GetFileSize(file_handle_, 0);
size_t adjusted_file_size = initial_file_size == 0 ?
min_file_size : initial_file_size;
file_mapping_handle_ = ::CreateFileMapping(
    file_handle_, 0, PAGE_READWRITE, 0, adjusted_file_size, 0);
if (file_mapping_handle_ == INVALID_HANDLE_VALUE) return;
data_ = static_cast<char*>(::MapViewOfFile(
    file_mapping_handle_, FILE_MAP_WRITE, 0, 0, 0));
if (data_) {
    size_ = initial_file_size;
    capacity_ = adjusted_file_size;
}
#endif
}

void MemoryFile::resize(size_t new_size) {
    if (new_size > capacity_) reserve(new_size);
    size_ = new_size;
}

void MemoryFile::reserve(size_t new_capacity) {
    if (new_capacity <= capacity_) return;
#ifdef __unix__
    ::munmap(data_, size_);
    ::ftruncate(file_handle_, new_capacity);
    data_ = static_cast<char*>(::mmap(
        0, new_capacity, PROT_READ | PROT_WRITE, MAP_SHARED,
        file_handle_, 0));
    if (data_ == MAP_FAILED) data_ = 0;
    capacity_ = new_capacity;
#elif defined(_WIN32)
    ::UnmapViewOfFile(data_);
    ::CloseHandle(file_mapping_handle_);
    file_mapping_handle_ = ::CreateFileMapping(
        file_handle_, 0, PAGE_READWRITE, 0, new_capacity, 0);
    capacity_ = new_capacity;
    data_ = static_cast<char*>(::MapViewOfFile(
        file_mapping_handle_, FILE_MAP_WRITE, 0, 0, 0));
#endif
}

MemoryFile::~MemoryFile() {
#ifdef __unix__
    ::munmap(data_, size_);
    if (size_ != capacity_)
    {
        ::ftruncate(file_handle_, size_);
    }
    ::close(file_handle_);
#elif defined(_WIN32)
    ::UnmapViewOfFile(data_);
    ::CloseHandle(file_mapping_handle_);
    if (size_ != capacity_)
    {

```

```

        ::SetFilePointer(file_handle_, size_, 0, FILE_BEGIN);
        ::SetEndOfFile(file_handle_);
    }
    ::CloseHandle(file_handle_);
#endif
}

bool MemoryFile::flush() {
#ifdef __unix__
    return ::msync(data_, size_, MS_SYNC) == 0;
#elif defined(_WIN32)
    return ::FlushViewOfFile(data_, size_) != 0;
#endif
}

```

File "memory_file_test.cpp":

```

#include "memory_file.hpp"
#include <iostream>

bool CopyFile(const char* source, const char* dest, bool overwrite)
{
    InputMemoryFile source_mf(source);
    if (! source_mf.data()) return false;
    MemoryFile dest_mf(dest, overwrite ?
        MemoryFile::if_exists_truncate_if_not_exists_create :
        MemoryFile::if_exists_fail_if_not_exists_create);
    if (! dest_mf.data()) return false;
    dest_mf.resize(source_mf.size());
    if (source_mf.size() != dest_mf.size()) return false;
    std::copy(source_mf.data(), source_mf.data() + source_mf.size(),
        dest_mf.data());
    return true;
}

int main() {
    if (! CopyFile("memory_file_test.cpp", "copy.tmp", true)) {
        std::cerr << "Copy failed" << std::endl;
        return 1;
    }
}

```

it:Ottimizzare C++/Tecniche generali di ottimizzazione/Input/Output²

² <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FTecniche%20generali%20di%20ottimizzazione%2FInput%2FOutput>

12 Memoization

*Memoization*¹ techniques (aka *caching* techniques) are based on the principle that if you must repeatedly compute a pure function, that is a *referentially transparent*² function (aka mathematical function³), for the same argument, and if such computation requires significant time, you can save time by storing the result of the first evaluation and retrieve that result the other times.

12.0.48 Look-up table

If you often have to call a pure function that has a small integer interval as domain, pre-compute (at compile time or at program start-up time) all the values of the function for every value of the domain and put them in a static array called *lookup table*⁴. When you need the value of the function for a particular value of the domain, read the corresponding value of such array.

For example, to compute the square root of an integer between 0 and 9, the following function is faster:

```
double sqrt10(int i) {
    static double lookup_table[] = {
        0, 1, sqrt(2.), sqrt(3.), 2,
        sqrt(5.), sqrt(6.), sqrt(7.), sqrt(8.), 3,
    };
    assert(0 <= i && i < 10);
    return lookup_table[i];
}
```

Array access is very fast, mainly if the accessed cell is in processor data cache. Therefore, if the lookup table is not large, almost surely its access is faster than the function to evaluate.

If the lookup table is large, it may be no more efficient, for the memory footprint, for the time to pre-compute all the values, if it doesn't fit the processor data cache. But if the function to evaluate is slow, it is called many times and you can afford to use much memory, consider using a lookup table up to several hundreds of kilobytes. It is rarely efficient to exceed one megabyte.

12.0.49 One-place cache

If you often have to call a pure function with the same arguments, the first time the function is called save the arguments and the result in static variables. When the function is called

1 <http://en.wikipedia.org/wiki/Memoization>
2 <http://en.wikipedia.org/wiki/Referential%20transparency%20%28computer%20science%29>
3 <http://en.wikipedia.org/wiki/Function%20%28mathematics%29>
4 <http://en.wikipedia.org/wiki/Lookup%20table>

again, compare the new arguments with the saved ones; if they match, return the saved result, otherwise compute the result and store the new arguments and the new result.

For example, instead of the following function:

```
double f(double x, double y) {  
    return sqrt(x * x + y * y);  
}
```

you can use this function:

```
double f(double x, double y) {  
    static double prev_x = 0;  
    static double prev_y = 0;  
    static double result = 0;  
    if (x == prev_x && y == prev_y) {  
        return result;  
    }  
    prev_x = x;  
    prev_y = y;  
    result = sqrt(x * x + y * y);  
    return result;  
}
```

Notice that, for faster processing it isn't necessary that the function be called with the same arguments for the entire program session. It is enough that it is called some times with the same arguments, then some other times with other arguments. In such cases, the computation is performed only when the arguments change.

Obviously, instead of increasing the speed, this technique may decrease it if the function is called with almost always changing arguments or if the comparison between the new arguments and the old ones requires more time than the computation of the function itself.

Notice that if you use static variables this function is not thread-safe and cannot be recursive. If this function must be called concurrently by several threads, it is necessary to replace the static variables with variables that are distinct for every thread⁵.

Notice also that in the example it is assumed that the function has zero value when both arguments are zero. Failing this, you should choose another solution, such as one of the following:

- Initialize the variable *result* with the value that corresponds to all-zero arguments.
- Initialize the variables *prev_x* and *prev_y* with values that will never be passed as arguments.
- Add a static flag indicating whether the static variables keep valid values and check that flag at every call.

12.0.50 N-places cache

If you often have to call a pure function with arguments that in most cases belong to a small domain, use a static *map* (aka *dictionary*) that is initially empty. When the function is called, search the map for the function argument. If you find it, return the associated value, otherwise compute the result and insert the pair argument-result into the map.

⁵ <http://en.wikipedia.org/wiki/Thread-local%20storage>

Here is an example in which the map is implemented using an array. The same function was used for the example of the guideline "One-place cache" in this section:

```
double f(double x, double y) {
    static const int n_buckets = 8; // should be a power of 2
    static struct {
        double x; double y; double result;
    } cache[n_buckets];
    static int last_read_i = 0;
    static int last_written_i = 0;
    int i = last_read_i;
    do {
        if (cache[i].x == x && cache[i].y == y) {
            return cache[i].result;
        }
        i = (i + 1) % n_buckets;
    } while (i != last_read_i);
    last_read_i = last_written_i = (last_written_i + 1) % n_buckets;
    cache[last_written_i].x = x;
    cache[last_written_i].y = y;
    cache[last_written_i].result = sqrt(x * x + y * y);
    return cache[last_written_i].result;
}
```

Some functions, although they have a theoretically large domain, are called most often with few distinct arguments.

For example, a word processor may have many installed fonts, but in a typical document only a few fonts are used for most characters. A rendering function that has to handle the font of every character of the document will be called typically with few distinct values. For such cases, an N-places cache is preferable to a one-place cache, as in the example.

The remarks about static variables, in guideline "One-place cache" in this section, apply also to this case.

For small caches (in the example, having 8 places) the most efficient algorithm is a sequential scan on an array. To implement a larger cache, though, a search tree or a hash table could be more efficient. In addition, the cache of the example has fixed size, but it could be expedient to have a variable-size cache.

Usually, the last read element is the most likely for the next call. Therefore, as in the example, it may be expedient to save its position and to begin the search from that position.

If the cache does not expand itself indefinitely, there is the problem choosing the element to replace. Obviously, it would be better to replace the element that is the least likely to be requested by the next call. In the example, it is assumed that, among the elements in the cache, the first inserted element is the least probable for the next call. Therefore, the write scans cyclically through the array. Often, a better criterion is to replace the least recently read element instead of the least recently written element. To implement such criterion, a more complex algorithm is required.

it:Ottimizzare C++/Tecniche generali di ottimizzazione/Caching⁶

⁶ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FTecniche%20generali%20di%20ottimizzazione%2FCaching>

13 Sorting

The C++ Standard Template Library (STL) provides the template function `sort` that implements a comparison sort¹ algorithm. Because `sort` is templated, it can be used for various types of sequences holding any type of key, as long as the keys are comparable (implement the `<` operator). A good compiler can generate code optimized for the various kinds of sequence/key combinations.

The reference implementation of the STL uses the introsort² algorithm (since the 2000 release; the GNU C++ library uses the reference implementation). This algorithm is a very fast combination of quicksort³ and heapsort⁴ with a specially designed selection algorithm.

The `sort` template function is not guaranteed to be stable⁵. When a stable sort is required, use the `stable_sort` template function instead.

This section suggests alternatives to the `sort` and `stable_sort` template functions that may be faster in specific cases.

13.1 Sorting with small ranges of keys

To sort a data set according an integer key having a small range, use the *counting sort*⁶ algorithm.

The *counting sort* algorithm has $O(N+M)$ complexity, where N is the number of elements to sort and M is the range of the sort keys, that is the difference between the highest key and the lowest key. In case N elements are to be sorted whose key is an integer number belonging to an interval containing no more than two times N values (i.e when $M \leq 2 * N$ holds), this algorithm may be quite faster than `sort`. In some cases it is faster even with larger ranges.

This algorithm may be used also for a partial ordering; for example, if the keys are integers between zero and one billion, you can still sort them using only the most significant byte, so to get an order for which the formula $a_n < a_{n+1} + 256 * 256 * 256$ holds.

1 <http://en.wikibooks.org/wiki/%3A%3AComparison%20sort>
2 <http://en.wikibooks.org/wiki/%3A%3AIntrosort>
3 <http://en.wikibooks.org/wiki/%3A%3AQuicksort>
4 <http://en.wikibooks.org/wiki/%3A%3AHeapsort>
5 <http://en.wikibooks.org/wiki/%3A%3AStable%20sort>
6 <http://en.wikibooks.org/wiki/Algorithm%20implementation%2FSorting%2FCounting%20sort%23C%2B%2B>

13.1.1 Example: sorting 8-bit integers

Say you want to sort an array of arbitrary `char` elements. These take on values in the range `0..CHAR_MAX` (inclusive), accounting for `CHAR_MAX+1` different values. `CHAR_MAX` is defined in the header `<climits>`.

```
#include <climits>

void count_sort(char *a, char *const end)
{
    size_t freq[CHAR_MAX+1] = {0};
    char *p;

    for (p = a; p < end; ++p)
        freq[*p] += 1;

    char c;
    for (c = 0, p = a; c < UINT8_MAX; ++c)
        while (freq[c]-- > 0)
            *p++ = c;
}
```

The `counting_sort` function implements the pigeonhole sort⁷ algorithm. It takes a pointer to the first element of the input array and a pointer that points *one element beyond* the end of the array. Why? Because we don't have to stop here.

We can generalize `counting_sort` to a template function that also works for `string`, `vector<char>` and other sequence types, without loss of efficiency. When doing so, we need to work with iterators rather than pointers.

```
#include <climits>

template <typename OutputIter>
void counting_sort(OutputIter const &begin, OutputIter const &end)
{
    size_t freq[CHAR_MAX+1] = {0};
    OutputIter it;

    for (it = begin; it < end; ++it)
        freq[*it] += 1;

    char c;
    for (c = 0, it = begin; c < CHAR_MAX; ++c)
        while (freq[c]-- > 0)
            *it++ = c;
}
```

13.2 Partial sorting

13.2.1 Partitioning

If you have to split a sequence according a criterion, use a partitioning algorithm, instead of a sorting one.

⁷ <http://en.wikibooks.org/wiki/%3Aw%3APigeonhole%20sort>

In STL there is the `std::partition` algorithm, that is faster than the `std::sort` algorithm, as it has $O(N)$ complexity, instead of $O(N \log(N))$.

13.2.2 Stable partitioning and sorting

If you have to partition or sort a sequence for which equivalent entities may be swapped, don't use a stable algorithm.

In STL there is the `std::stable_partition` partitioning algorithm, that is slightly slower than the `std::partition` algorithm; and there is the `std::stable_sort` sorting algorithm, that is slightly slower than the `std::sort` algorithm.

13.2.3 Order partitioning

If you have to pick out the first N elements from a sequence, or the N^{th} element in a sequence, use an order partitioning algorithm, instead of a sorting one.

In STL there is the `std::nth_element` algorithm, that, although slightly slower than the `std::stable_partition` algorithm, is quite faster than the `std::sort` algorithm, as it has $O(N)$ complexity, instead of $O(N \log(N))$.

13.2.4 Sorting only the first N elements

If you have to sort the first N elements of a much longer sequence, use an order statistic algorithm, instead of a sorting one.

In STL there are the `std::partial_sort` and `std::partial_sort_copy` algorithms, that, although slower than the `std::nth_element` algorithm, are so much faster than the `std::sort` algorithm as the partial sequence to sort is shorter than the whole sequence.

it:Ottimizzare C++/Tecniche generali di ottimizzazione/Ordinamento⁸

⁸ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FTecniche%20generali%20di%20ottimizzazione%2FOrdinamento>

14 Other techniques

14.0.5 Query cursor

Instead of defining a function that returns a collection (aka *snapshot*), define a function that returns an iterator (aka *cursor* or *dynaset*), with which you can generate or possibly change the required data.

This technique is particularly useful for database queries, but is applicable also to internal data structures.

Let's assume you have a collection (or a set of collections) encapsulated in a class. Such class exposes one or more member functions to extract (or filter) a subset from such collection.

A way to get it is to construct a new collection, to copy the extracted data into it, and to return such collection. In the database jargon, such collection is called *snapshot*.

This technique is effective but inefficient, as the allocation and copy of the snapshot takes a lot of time and a lot of storage space. In addition, it has the shortcoming that, until all the data has been extracted, you cannot proceed to process the already extracted data.

Here is an equivalent but more efficient technique.

The query function returns an iterator. In database jargon, such iterator is called *cursor* or *dynaset*. The caller uses such iterator to extract, one at a time, the data filtered, and possibly to change them.

Notice that this solution is not exactly equivalent, as if during the iterator use the collection is changed by another function call, possibly coming from another thread, it may happen that the iterator is invalidated, or just that the filtered collection do not corresponds to the specified criteria. Therefore, you can apply this technique only when you are sure that the underlying collection is not changed in any way, except by the iterator itself, during the whole life of the iterator.

This technique is independent of the programming language, as the iterator concept is an abstract design pattern.

14.0.6 Binary search

If you have to do many searches in a rarely changed collection, instead of using a search tree or a hash table, you can get a speed up if you put the data in an array, sort the array, and do binary searches¹ on it.

A binary search on an array has logarithmic complexity, like search trees, but has the advantage of compactness and locality of reference typical of arrays.

¹ <http://en.wikipedia.org/wiki/Binary%20search>

If the array is changed, this algorithm may still be competitive, as long as the changes are much less frequent than searches.

If every collection change consists in very few insertions or changes or deletions of elements, it is better to shift the array at every change operation. Instead, if a collection change is more bulky, it is better to recreate and sort the whole array.

In C++, if the array length is not a compile-time constant, use a `vector`.

14.0.7 Singly-linked lists

If for a list you don't need bidirectional iterators, you don't need to insert elements at the end or before the current element, and you don't need to know how many elements there are in the list, use a singly-linked list², instead of a doubly-linked list³.

Such container, although it has many shortcomings, occupies less space and it is faster.

Typically, the heading of a doubly-linked list contains a pointer to the head of the list, a pointer to the back, and the counter of elements, while the heading of a singly-linked list contains only a pointer to the head of the list. In addition, typically, every node of a doubly-linked list contains a pointer to the previous node and a pointer to the next node, while every node of a singly-linked list contains only a pointer to the next node. At last, every element insertion into a doubly-linked list must update four pointers and increment a counter, while every element insertion into a singly-linked list must only update two pointers.

In the C++ standard library, the `std::list` container is implemented by a doubly-linked list. The `slist` container, non-standard but available in various libraries, and the `forward_list` container, that will be in C++0x standard library, are implemented by singly-linked lists.

it:Ottimizzare C++/Tecniche generali di ottimizzazione/Altre tecniche⁴

2 <http://en.wikipedia.org/wiki/Linked%20list%23Singly-linked%20list>

3 <http://en.wikipedia.org/wiki/Linked%20list%23Doubly-linked%20list>

4 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FTecniche%20generali%20di%20ottimizzazione%2FAltre%20tecniche>

15 Code optimization

In this chapter some techniques, specific for C++ language, are proposed. They are to be applied only in bottlenecks, as, although they may speed up execution, they also make more complex and less maintainable the source code.

In addition, such guidelines in some cases could worsen the performance instead of improving it, and therefore their effect should be always measured before releasing them.

The optimization techniques are grouped according their goal.

1. Allocations and deallocations¹
2. Run-time support²
3. Instruction count³
4. Constructions and destructions⁴
5. Pipeline⁵
6. Memory access⁶
7. Faster operations⁷
8. Compile time optimization⁸

it:Ottimizzare C++/Ottimizzazione del codice C++⁹

1 Chapter 16 on page 57

2 Chapter 17 on page 61

3 Chapter 18 on page 63

4 Chapter 19 on page 67

5 Chapter 20 on page 71

6 Chapter 21 on page 75

7 Chapter 22 on page 79

8 <http://en.wikibooks.org/wiki/Optimizing%20C%2B%2B%2FCode%20optimization%2FCompile%20time%20optimization>

9 <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B>

16 Allocations and deallocations

Even using a very efficient allocator, the allocation and deallocation operations take a significant time, and often the allocator is not very efficient.

In this section some techniques are described to decrease the total number of memory allocations, and their corresponding deallocations. They are to be applied only in bottlenecks, that is after having measured that the large number of allocations has a significant impact on performance.

16.0.8 The `alloca` function

In non-recursive functions, to allocate variable-size but not large memory space, use the `alloca` function.

It is very efficient, as it allocates space on the stack.

It is a non-standard function, but it is available with many compilers for several operating systems.

It may be used even to allocate an array of objects having constructors, as long as the *placement-new* operator is called on the allocated space, but it shouldn't be used for an array of objects having a destructor or which, directly or indirectly, contain member variables having a destructor, as such destructors would never be called.

Though, it is rather dangerous, as, if called too many times or with a too big value, it overflows the stack, and, if called for objects having a destructor, it causes resource leaks. Therefore this function is to be used sparingly.

16.0.9 Move allocations and deallocations

Move before bottlenecks memory allocations, and after bottlenecks the matching deallocations.

Variable length dynamic memory management is much slower than automatic memory management.

Analogous optimization is to be done for operations causing allocations indirectly, as the copy of objects which, directly or indirectly, own dynamic memory.

16.0.10 The `reserve` function

Before adding elements to a `vector` or to a `string` object, call its member function `reserve` with a size big enough for most cases.

If objects are repeatedly added to a `vector` or `string` object, several costly reallocations are performed. To avoid such reallocations, it is enough to initially allocate the required space.

16.0.11 Keep vectors capacity

To empty a `vector<T> x` object without deallocating its memory, use the statement `x.resize(0);`; to empty it and deallocate its memory, use the statement `vector<T> ().swap(x);`.

To empty a `vector` object, there also exists the `clear()` member function, but, the C++ standard does not specify whether or not this function preserves the allocated capacity of the `vector`.

If you are repeatedly filling and emptying a `vector` object, and thus you want to avoid frequent reallocations, perform the emptying by calling the `resize` member function, which, according to the standard, preserves the capacity of the object. If instead you have finished using a large `vector` object, and you may not use it again or you are going to use it with substantially fewer elements, you should free the object's memory by calling the `swap` function on a new empty temporary `vector` object.

16.0.12 swap function overload

For every copyable concrete class T which, directly or indirectly, owns some dynamic memory, redefine the appropriate swap functions.

In particular, add to the class `public` member function having the following signature:

```
void swap(T& throw());
```

and add the following non-member function in the same namespace that contains the class `T`:

```
void swap(T& lhs, T& rhs) { lhs.swap(rhs); }
```

and, if the class is not a class template, add also the following non-member function in the same file that contains the class `T` definition:

```
namespace std { template<> swap(T& lhs, T& rhs) { lhs.swap(rhs); } }
```

In the standard library, the `swap` function is called frequently by many algorithms. Such function has a generic implementation and specialized implementations for various types of the standard library.

If objects of a non-standard class are used in a standard library algorithm that calls `swap` on them, and the `swap` function is not overloaded, the generic implementation is used.

The generic implementation of the `swap` function causes the creation and destruction of a temporary object and the execution of two object assignments. Such operation take much time if applied to objects that own dynamic memory, as such memory is reallocated three times.

The ownership of dynamic memory may be even only indirect. For example, if a member variable is a `string` or a `vector`, or is an object that contains a `string` or `vector` object, the memory owned by these objects is reallocated every time the object that contains them is copied. Therefore, even in these cases the `swap` function is to be overloaded.

If the object doesn't own dynamic memory, the copy of the object is much faster, and however it is not noticeably slower than using other techniques, and so no `swap` overload is needed.

If the class is not copyable or abstract, the `swap` function must never be called on object of such type, and therefore also in these cases no `swap` function is to be redefined.

To speed up the function `swap`, you have to specialize it for your class. There are two possible ways to do that: in the same namespace of the class (that may be the global one) as an overload, or in the namespace `std` as a specialization of the standard template. It is advisable to define it in both ways, as, first, if it is a class template only the first way is possible, and then some compilers do not accept or accept with a warning a definition only in the first way.

The implementations of such functions must access all the members of the object, and therefore they need to call a member function, that by convention is called again `swap`, that does the actual work.

Such work consists in swapping all the non-static members of the two objects, typically by calling the `swap` function on them, without qualifying its namespace.

To put the function `std::swap` into the current scope, the function must begin with the statement:

```
using std::swap;
```

it:Ottimizzare C++/Ottimizzazione del codice C++/Allocazione e deallocazione¹

¹ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FAllocazione%20e%20deallocazione>

17 Run-time support

C++ run-time support routines obviously have a significant cost, because otherwise such behavior would have been inlined. Here techniques are presented to avoid the language features that cause an implicit call to costly run-time support routines.

17.0.13 The `typeid` operator

Instead of using the `typeid` operator, use a virtual function.

Such operator may take more time than a virtual function call.

17.0.14 The `dynamic_cast` operator

Instead of the `dynamic_cast` operator, use the `typeid` operator, or, better, a virtual function call.

Such operator may take a time noticeably longer than a virtual function call, and longer also than the `typeid` operator.

17.0.15 Empty exception specification

Use the empty exception specification (that is, append `throw()` to the declaration) for the functions you are sure will never throw exceptions.

Some compilers use such information to simplify the bookkeeping needed to handle exceptions.

17.0.16 The `try/catch` statement

For every bottleneck, move before the bottleneck the `try` keywords, and after the bottleneck the matching `catch` clauses.

In other words, hoist `try/catch` statements out of bottlenecks.

The execution of a `try/catch` statement sometimes is free of charge, but other times causes a slowdown. Avoid the repeated execution of such block inside bottlenecks.

17.0.17 Floating point vs integer operations

If the target processor does not contain a floating point unit, replace floating point functions, constants and variables with the corresponding integer functions, constants and variables; if the target processor contains only a single precision floating point unit, replace double functions, constants and variables with their `float` correspondents.

Present processors for desktop or server computers contain dedicated hardware for floating point arithmetic, both at single and at double precision, and therefore such operations are almost as fast as their integer correspondents.

Instead, some processors for embedded systems do not contain dedicated hardware for floating point arithmetic, or contain hardware able to handle only single precision numbers. Therefore, in such systems, the operation that cannot be performed by hardware are emulated by very slow library functions. In such case, it is much more efficient to use integer arithmetic, or, if available in hardware, single precision floating point arithmetic.

To handle fractional numbers by using integer operations, every number is to be meant as multiplied by a scale factor. To do that, every number is multiplied by such factor at input, and is divided by the same factor at output, or vice versa.

17.0.18 Number to string conversion

Use optimized functions to convert numbers to strings.

The standard functions to convert an integer number to a string or a floating point number to string are rather inefficient. To speed up such operations, use non-standard optimized function, possibly written by yourself.

17.0.19 Use of `cstdio` functions

To perform input/output operations, instead of using the C++ streams, use the old C functions, declared in the `cstdio` header.

C++ I/O primitives have been designed mainly for type safety and for customization rather than for performance, and many library implementation of them turn out to be rather inefficient. In particular, the C language I/O functions `fread` and `fwrite` are more efficient than the `fstream` `read` and `write` member functions.

If you have to use C++ streams, use `"\n"` instead of `std::endl` since `std::endl` also flushes the stream.

it:Ottimizzare C++/Ottimizzazione del codice C++/Supporto run-time¹

¹ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FSupporto%20run-time>

18 Instruction count

Even the language features that generate inlined code may have a significant cost, as such instructions are anyway to be executed. In this section some techniques are presented to decrease the total number of machine instructions that the processor will have to execute to perform a given operation.

18.0.20 Cases order in `switch` statement

In `switch` statements, sort the cases by decreasing probability.

In the guideline "Cases order in `switch` statement" in section 3.1, it was already suggested to put before the most typical cases, that is those that were presumed to be more probable. As further optimization, you can count, in typical runs, the actual number of times every case is chosen, and sort the cases from the most frequent to the less frequent.

18.0.21 Template integer parameters

If an integer value is a constant in the application code, but is a variable in library code, make it a template parameter.

Let's assume you are writing the following library function, in which both `x` and `y` do not have a known value when the library is developed:

```
int f1(int x, int y) { return x * y; }
```

Such function may be called from the following application code, in which `x` does not have a constant value, but `y` is the constant 4:

```
int a = f1(b, 4);
```

If, when you write the library, you know that the caller will surely pass a constant for the argument `y`, you can transform your function into the following function template:

```
template <int Y> int f2(int x) { return x * Y; }
```

Such function may be called from the following application code:

```
int a = f2<4>(b);
```

Such a call instantiates automatically the following function:

```
int f2(int x) { return x * 4; }
```

The latter function is faster than the former function `f1`, for the following reasons:

- Only one argument is passed to the function (x) instead of two (x and y).
- The multiplication by an integer constant (4) is always faster than a multiplication by an integer variable (y).
- As the constant value (4) is a power of two, the compiler, instead of performing an integer multiplication, performs a bit shift.

In general, the integer template parameters are constants for those who instantiate the template and therefore for the compiler, and constants are handled more efficiently than variables. In addition, some operations involving constants are pre-computed at compilation-time.

If, instead of a normal function, you already have a function template, it is enough to add a further parameter to that template.

18.0.22 The Curiously Recurring Template Pattern

If you have to write a library abstract base class such that in every algorithm in the application code only one class derived from such base class will be used, use the *Curiously Recurring Template Pattern*¹.

Let's assume you are writing the following library base class:

```
class Base {
public:
    void g() { f(); }
private:
    virtual void f() = 0;
};
```

In this class, the function `g` performs an algorithm that calls the function `f` as an abstract operation for the algorithm. In design patterns terminology, `g` is a *template method*² design pattern. The purpose of such class is to allow to write the following application code:

```
class Derived1: public Base {
private:
    virtual void f() { ... }
};
...
Base* p1 = new Derived1;
p1->g();
```

In such a case, it is possible to transform the previous library code into the following:

```
template <class Derived> class Base {
public:
    void g() { f(); }
private:
    void f() { static_cast<Derived*>(this)->f(); }
};
```

As a consequence, the application code will become the following:

```
class Derived1: public Base<Derived1> {
```

1 <http://en.wikipedia.org/wiki/Curiously%20recurring%20template%20pattern>
2 <http://en.wikipedia.org/wiki/Template%20method%20pattern>

```
private:
    void f() { ... }
};
...
Derived1* p1 = new Derived1;
p1->g();
```

In such a way, the call to `f` in the function `Base<Derived1>::g` is statically bound to the member function `Derived1::f`, that is the call to such function is no more `virtual`, and can be inlined.

Though, let's assume you want to add the following definition:

```
class Derived2: public Base<Derived2> {
protected:
    void f() { ... }
};
```

With this technique it wouldn't be possible to define a pointer or a reference to a base class that is common to both `Derived1` and `Derived2`, as such base classes are two unrelated types; as a consequence, this technique is not applicable when you want to allow the application code to define a container of arbitrary objects derived from the class `Base`.

Other limitations are:

- `Base` is necessarily an abstract type;
- an object of type `Derived1` cannot be converted into an object of type `Derived2` or vice versa;
- for every derivation of `Base`, all the machine code generated for `Base` is duplicated.

18.0.23 The *Strategy* design pattern

If an object that implements the *Strategy*³ design pattern (aka *Policy*) is a constant in every algorithm of the application code, eliminate such an object, make `static` all its members, and add its class as a template parameter.

Let's assume you are writing the following library code, that implements the *Strategy* design pattern:

```
class C;
class Strategy {
public:
    virtual bool is_valid(const C& c) const = 0;
    virtual void run(C& c) const = 0;
};

class C {
public:
    void set_strategy(const Strategy& s) { s_ = s; }
    void f() { if (s_.is_valid(*this)) s_.run(*this); }
private:
    Strategy s_;
};
```

This library code has the purpose to allow the following application code:

```
class MyStrategy: public Strategy {
public:
```

³ <http://en.wikipedia.org/wiki/Strategy%20pattern>

```

    virtual bool is_valid(const C& c) const { ... }
    virtual void run(C& c) const { ... }
};
...
MyStrategy s; // Object representing my strategy.
C c; // Object containing an algorithm with customizable strategy.
c.set_strategy(s); // Assignment of the custom strategy.
c.f(); // Execution of the algorithm with assigned strategy.

```

In such a case, it's possible to convert the previous library code into the following:

```

template <class Strategy>
class C {
public:
    void f() {
        if (Strategy::is_valid(*this)) Strategy::run(*this);
    }
};

```

As a consequence, the application code will become the following:

```

class MyStrategy {
public:
    static bool is_valid(const C<MyStrategy>& c) { ... }
    static void run(C<MyStrategy>& c) { ... }
};
...

C<MyStrategy> c; // Object with statically assigned strategy.
c.f(); // Execution with statically assigned strategy.

```

In such a way, the object-strategy is avoided, and the member functions `MyStrategy::is_valid` and `MyStrategy::run` are statically bound, that is calls to virtual functions are avoided.

Though, such solution does not allow to choose the strategy at run-time, and of course neither to change it during the object life. In addition, the algorithm code is duplicated for every instantiation of its class.

18.0.24 Bitwise operators

If you have to perform boolean operations on a set of bits, put those bits in an unsigned int object, and use bitwise operators on it.

The bitwise operators (&, |, ^, <<, and >>) are translated in single fast instructions, and operate on all the bits of a register in a single instruction.

it:Ottimizzare C++/Ottimizzazione del codice C++/Numero di istruzioni⁴

⁴ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FNumero%20di%20istruzioni>

19 Constructions and destructions

Often it happens that, while processing an expression, a temporary object is created, which is destroyed at the end of that expression. If such object is of a fundamental type, almost always the compiler succeeds in avoiding its creation, and anyway the creation and destruction of an object of a fundamental type are quite fast. Instead, if the object is of a composite type, its creation and destruction have an unlimited cost, as they cause the call of a constructor and the call of the destructor, that may take any time.

In this section some techniques are presented to avoid that composite temporary objects are created, and therefore that their constructors and destructors are called.

19.0.25 Functions return value

For non-inlined functions, try to declare a return type for which an object copy moves no more than 8 bytes. If unfeasible, at least construct the result object in the `return` statement.

While compiling a non-inlined function, the compiler cannot know if the return value will be used, and therefore it must generate it anyway. To generate and assign an object whose copy moves no more than 8 bytes has little or no cost, but to generate and assign more complex object takes time. If the temporary object owns resources, the taken time is enormously bigger, but even without allocations, the taken time grows with the number of machine words used by such object.

However, if the object to return is constructed in the `return` instructions themselves, therefore without assigning such value to a variable, the language standard guarantees an optimization called *Return Value Optimization*, that prevents the creation of temporaries.

Some compilers succeeds to avoid creating temporaries, even when the returned object is associated to a local variable (with the so-called *Named Return Value Optimization*), but this is not generally guaranteed and has anyway some limitations. C++ FAQ¹

To check whether one of the above optimizations is applied, increment a static container in every constructor, destructor, and in assignment operators of the returned object class. In case no optimization is applied, resort to one of the following alternative techniques:

- Make `void` the function return type, and add to it a passed-by-reference argument, acting as return value.
- Transform the function into a constructor of the return type, taking the same function arguments.
- Make the function return an object of an auxiliary type, that steals the resources from the return object and passes them to the destination object, without copying their contents.

¹ <http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.9>

- Use an *expression template*², that is an advanced technique, part of the programming paradigm called *template metaprogramming*³.
- If using the C++0x standard, use an *rvalue reference*.

19.0.26 Moving declarations outside loops

If a variable is declared in the body of a loop, and an assignment to it costs less than a construction plus a destruction, move that declaration before the loop.

If the variable is declared in the body of a loop, the associated object is constructed and destructed at every iteration, while if it is outside the loop, such object is constructed and destructed only once, but is presumably assigned one more time in the body of the loop.

Though, in many cases, an assignment costs exactly as much as a pair construction+destruction, and thus in such cases there is no gain in moving the declaration outside the loop and adding an assignment inside.

19.0.27 Assignment operator

In an assignment operator overload (operator=), if you are sure that it will never throw exceptions, copy every member variable, instead of using the *copy&swap* idiom.

The most efficient way to copy an object is to imitate an appropriate initialization list of a copy constructor, that is, first, to call the analogous member functions of the base classes, and then to copy every member variable, in declaration order.

Unfortunately, such technique is not *exception-safe*, that is if during this operation an exception is thrown, the destructors of some already constructed sub-objects could never be called. Therefore, if there is the chance that during the copy an exception is thrown, you must use an *exception-safe* technique, although it won't have optimal performance.

The most elegant *exception-safe* assignment technique is the one called *copy&swap*. It is exemplified by the following code, in which C represents the name of the class, and C a member function to define:

```
C& C::operator=(C new_value) {
    swap(new_value);
    return *this;
}
```

19.0.28 Overload to avoid conversions

To avoid costly conversions, define overloaded functions for the most typical argument types.

Let's assume you wrote the following function:

2 <http://en.wikibooks.org/wiki/More%20C%2B%2B%20Idioms%2FExpression-template>
 3 <http://en.wikipedia.org/wiki/Template%20metaprogramming>

```
int f(const std::string& s) { return s[0]; }
```

whose purpose is to allow to write the following code:

```
std::string s("abc");
int n = f(s);
```

But it can be used also by the following code:

```
int n = f(string("abc"));
```

And, thanks to the implicit conversion from `char*` to `std::string`, it can be used also by the following code:

```
int n = f("abc");
```

Both the last two calls to the `f` function are inefficient, as they create a temporary non-empty `std::string` object.

To keep the efficiency of the first example call, you have to define also the following function overload:

```
int f(const char* s) { return s[0]; }
```

In general, if a function is called by passing to it an argument of an unexpected type but that can be implicitly converted to an expected type, a temporary of the expected type is created.

To avoid such temporary object, you have to define an overload of the original function that takes an argument of the type of the actual passed object, thus avoiding the need of a conversion.

it:Ottimizzare C++/Ottimizzazione del codice C++/Costruzioni e distruzioni⁴

⁴ <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FCostruzioni%20e%20distruzioni>

20 Pipeline

The conditional jump machine language instructions (aka *branches*), may be generated by many C++ language features, among which there are the `if-else`, `for`, `while`, `do-while`, and `switch-case` statements, and by boolean and conditional expressions operators.

Modern processors handle branches efficiently only if they can predict them. In case of prediction error, the steps already done by the pipeline¹ on the following instructions are useless and the processor must restart from the branch destination instruction.

The branch prediction is based on the previous iterations on the same instruction. If the branches follow a regular pattern, the prediction are successful.

The best cases are those in which a branch instruction has always the same effect; in such cases, the prediction is almost always correct. The worst case is that in which the branch instruction has a random outcome, with about a 50% probability to jump; in such case, the prediction in the average is correct half of the times, but it is not impossible that it is always wrong.

In bottlenecks, the hard-to-predict branches should be avoided. If a branch is predicted very badly, even replacing it with a rather slow sequence of instructions may result in a speed up.

In this section, some techniques are presented to replace branches with equivalent instructions.

20.0.29 Integer interval check

If you have to check whether an integer number `i` is between two integer numbers `min_i` and `max_i` included, and you are sure that `min_i <= max_i`, use the following expression:

```
unsigned(i - min_i) <= unsigned(max_i - min_i)
```

In the given conditions, the above formula is equivalent to the following, more intuitive formula:

```
min_i <= i && i <= max_i
```

The former formula performs two differences and one comparison, while the latter formula performs no difference and two comparisons. For pipelined processors, comparisons are slower than differences, because they imply a branch.

In addition, if `min_i` is a constant expression with zero value, the two differences disappear.

In particular, to check whether an integer `i` is a valid index for an array of `size` elements, that is to perform array bounds checking, use the following expression:

¹ <http://en.wikibooks.org/wiki/Microprocessor%20Design%2FPipelined%20Processors>

```
unsigned(i) < unsigned(size)
```

Obviously, if the expressions are already of an unsigned type, conversions are unneeded.

20.0.30 The binary look-up table

Instead of a conditional expression in which both cases are constants, use a look-up table with two-places.

If you have a statement like the following, where `c` and `d` represent constant expressions, and `b` represents a boolean expression:

```
a = b ? c : d;
```

that is equivalent to the following code:

```
if (b) a = c;
else a = d;
```

try to replace it with the following code, equivalent but perhaps faster:

```
static const type lookup_table[] = { d, c };
a = lookup_table[b];
```

The conditional expression is compiled into a branch. If such a branch is not well predicted, it takes longer than the lookup-table.

This technique may be applied also to a sequence of conditional expressions. For example, instead of the following code:

```
a = b1 ? c : b2 ? d : b3 ? e : f;
```

that is equivalent to the following code:

```
if (b1) a = c;
else if (b2) a = d;
else if (b3) a = e;
else a = f;
```

try to see if the following code is faster:

```
static const type lookup_table[] = { f, e, d, d, c, c, c, c };
a = lookup_table[b1 * 4 + b2 * 2 + b3];
```

20.0.31 Early address calculation

Try to calculate the value of a pointer or iterator somewhat before when you need to access the referenced object.

For example, in a loop, the following statement:

```
a = **p;
```

may be a bit less efficient than the following:

```
a = *p++;
```

In the first case, the value of the pointer or iterator is calculated just before it is used to access the referenced object, while in the second case it is computed in the previous iteration. In a pipelined processor, in the second case, the increment of the pointer may be performed simultaneously with the access of the referenced object, while in the first case the two operations must be serialized.

it:Ottimizzare C++/Ottimizzazione del codice C++/Pipeline²

² <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FPipeline>

21 Memory access

When the application accesses main memory, it implicitly uses both the various processor caches and the disk swapping mechanism by the virtual memory manager of the operating system.

Both the processor caches and the virtual memory manager process data block-wise, and therefore the software is faster if the few memory blocks contain the code and the data used by a single command. The principle that the data and the code processed by a command should reside in near regions of memory is called *locality of reference*¹.

This principle becomes even more important for performance in multi-threaded applications on multi-core systems, as if several threads running on different cores access the same cache block, the contention causes a performance degradation.

In this section techniques are proposed to optimize usage of the processor caches and of the virtual memory, by incrementing the locality of reference of code and data.

21.0.32 Nearing the code

Put near in the same compilation unit all the function definitions belonging to the same bottleneck.

In such a way, the machine code generated by compiling such functions will have near addresses, and so greater code locality of reference.

Another positive consequence is that the local static data declared and used by such functions will have near addresses, and so greater data locality of reference.

21.0.33 unions

In medium or large arrays or collections, use unions.

`unions` allow to save memory space in variable type structures, and therefore to make them more compact.

Though, don't use them for small or tiny objects, as there are no significant space gains, and with some compilers the objects put in a `union` are not kept in processor registers.

¹ <http://en.wikipedia.org/wiki/Locality%20of%20reference>

21.0.34 Bit-fields

If a medium or large object contains several integer numbers with a small range, transform them in bit-fields.

Bit-fields decrease the object size.

For example, instead of the following structure:

```
struct {
    bool b;
    unsigned short ui1, ui2, ui3; // range: [0, 1000]
};
```

that takes up 8 bytes, you can define the following structure:

```
struct {
    unsigned b: 1;
    unsigned ui1: 10, ui2: 10, ui3: 10; // range: [0, 1000]
};
```

that takes up only $(1 + 10 + 10 + 10 = 31 \text{ bits}, 31 \leq 32)$ 4 bytes.

For another example, instead of the following array:

```
signed char a[5]; // range: [-20, +20]
```

that takes up 5 bytes, you can define the following structure:

```
struct {
    signed a1: 6, a2: 6, a3: 6, a4: 6, a5: 6; // range: [-20, +20]
};
```

that takes up only $(6 + 6 + 6 + 6 + 6 = 30 \text{ bits}, 30 \leq 32)$ 4 bytes.

Though, there is a performance penalty in packing and unpacking the field. In addition, in the last example, the field can no more be accessed by index.

21.0.35 Template code independent of parameters

If in a class template a non-trivial member function does not depend on any template parameter, define a non-member function having the same body, and replace the original function body with a call to the new function.

Let's assume you wrote the following code:

```
template <typename T>
class C {
public:
    C(): x_(0) { }
    int f(int i) { body(); return i; }
private:
    T x_;
};
```

Try to replace the above code with the following:

```
template <typename T>
class C {
public:
    C(): x_(0) { }
    void f(int i) { return f_(i); }
private:
    T x_;
};

void f_(int i) { body(); return i; }
```

For every instantiation of a class template that uses a function of that class template, the whole code of the function is instantiated. If a function in that class template do not depend on any template parameters, at every instantiation the function machine code is duplicated. Such code replication bloats the program.

In a class template or in a function template, a big function could have a large part that do not depend on any template parameters. In such a case, first, factor out such a code portion as a distinct function, and then apply this guideline.

it:Ottimizzare C++/Ottimizzazione del codice C++/Accesso alla memoria²

² <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FAccesso%20alla%20memoria>

22 Faster operations

Some elementary operations, even being conceptually as simple as others, are much faster for the processor. A clever programmer can choose the faster instructions for the job.

Though, every optimizing compiler is already able to choose the fastest instructions for the target processor, and so some techniques are useless with some compilers.

In addition, some techniques may even worsen performance on some processors.

In this section some techniques are presented that may improve performance on some compiler/processor combinations.

22.0.36 Structure fields order

Arrange the member variables of classes and structures in such a way that the most used variables are in the first 128 bytes, and then sorted from the longest object to the shortest.

If in the following structure the `msg` member is used only for error messages, while the other members are used for computations:

```
struct {
    char msg[400];
    double d;
    int i;
};
```

you can speed up the computation by replacing the structure with the following one:

```
struct {
    double d;
    int i;
    char msg[400];
};
```

On some processors, the addressing of a member is more efficient if its distance from the beginning of the structure is less than 128 bytes.

In the first example, to address the `d` and `i` fields using a pointer to the beginning of the structure, an offset of at least 400 bytes is required.

Instead, in the second example, containing the same fields in a different order, the offsets to address `d` and `i` are of few bytes, and this allows to use more compact instructions.

Now, let's assume you wrote the following structure:

```
struct {
    bool b;
    double d;
};
```

```
    short s;  
    int i;  
};
```

Because of fields alignment, it typically occupies 1 (bool) + 7 (padding) + 8 (double) + 2 (short) + 2 (padding) + 4 (int) = 24 bytes.

The following structure is obtained from the previous one by sorting the fields from the longest to the shortest:

```
struct {  
    double d;  
    int i;  
    short s;  
    bool b;  
};
```

It typically occupies 8 (double) + 4 (int) + 2 (short) + 1 (bool) + 1 (padding) = 16 bytes. The sorting minimized the paddings (or holes) caused by the alignment requirements, and so generates a more compact structure.

22.0.37 Floating point to integer conversion

Exploit non-standard routines to round floating point numbers to integer numbers.

The C++ language do not provide a primitive operation to round floating point numbers. The simplest technique to convert a floating point number x to the nearest integer number n is the following statement:

```
n = int(floor(x + 0.5f));
```

Using such a technique, if x is exactly equidistant between two integers, n will be the upper integer (for example, 0.5 generates 1, 1.5 generates 2, -0.5 generates 0, and -1.5 generates -1).

Unfortunately, on some processors (in particular, the Pentium family), such expression is compiled in a very slow machine code. Some processors have specific instructions to round numbers.

In particular, the Pentium family has the instruction `fistp`, that, used as in the following code, gives much faster, albeit not exactly equivalent, code:

```
#if defined(__unix__) || defined(__GNUC__)  
    // For 32-bit Linux, with Gnu/AT&T syntax  
    __asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );  
#else  
    // For 32-bit Windows, with Intel/MASM syntax  
    __asm fld qword ptr x;  
    __asm fistp dword ptr n;  
#endif
```

The above code rounds x to the nearest integer, but if x is exactly equidistant between to integers, n will be the nearest even integer (for example, 0.5 generates 0, 1.5 generates 2, -0.5 generates 0, and -1.5 generates -2).

If this result is tolerable or even desired, and you are allowed to use embedded assembly, then use this code. Obviously, it is not portable to other processor families.

22.0.38 Integer numbers bit twiddling

Twiddle the bits of integer numbers exploiting your knowledge of their representation.

A collection of hacks of this kind is [here](#)¹. Some of these tricks are actually already used by some compilers, others are useful to solve rare problems, others are useful only on some platforms.

22.0.39 Floating point numbers bit twiddling

Twiddle the bits of floating point numbers exploiting your knowledge of their representation.

For the most common operation, compilers generate already optimized code, but some less common operation may become slightly faster if the bits are manipulated using bitwise integer operators.

One of such operations is the multiplication or the division by a power of two. To perform such operation, it is enough to add or subtract the exponent of the power of two to the exponent part of the floating point number.

For example, given a variable `f` of type `float`, conforming to IEEE 754 format, and given an integer positive expression `n`, instead of the following statement:

```
f *= pow(2, n);
```

you can use the following code:

```
if (*(int*)&f & 0x7FFFFFFF) { // if f==0 do nothing
    *(int*)&f += n << 23; // add n to the exponent
}
```

22.0.40 Array cells size

Ensure that the size (resulting from the `sizeof` operator) of non-large cells of arrays or of vectors be a power of two, and that the size of large cells of arrays or of vectors be not a power of two.

The direct access to an array cell is performed by multiplying the index by the cell size, that is a constant. If the second factor of this multiplication is a power of two, such an operation is much faster, as it is performed as a bit shift. Analogously, in multidimensional arrays, all the sizes, except at most the first one, should be powers of two.

This sizing is obtained by adding unused fields to structures and unused cells to arrays. For example, if every cell is a 3-tuple of `float` objects, it is enough to add a fourth *dummy* `float` object to every cell.

Though, when accessing the cells of a multidimensional array in which the last dimension is an enough large power of two, you can drop into the *data cache contention* phenomenon (aka *data cache conflict*), that may slow down the computation by a factor of 10 or more. This phenomenon happens only when the array cells exceed a certain size, that depends on the data cache, but is about 1 to 8 KB. Therefore, in case an algorithm has to process an array whose cells have or could have as size

¹ <http://www-graphics.stanford.edu/~seander/bithacks.html>

a power of two greater or equal to 1024 bytes, first, you should detect if the data cache contention happens, e in such a case you should avoid such phenomenon.

For example, a matrix of 100 x 512 `float` objects is an array of 100 arrays of 512 `floats`. Every cell of the first-level array has a size of $512 \times 4 = 2048$ bytes, and therefore it is at risk of data cache contention.

To detect the contention, it is enough to add an elementary cell (a `float`) to every to every last-level array, but keeping to process the same cells than before, and measure whether the processing time decrease substantially (by at least 20%). In such a case, you have to ensure that such improvement be stabilized. For that goal, you can employ one of the following techniques:

- Add one or more unused cells at the end of every last-level array. For example, the array `double a[100][1024]` could become `double a[100][1026]`, even if the computation will process such an array up to the previous sizes.
- Keep the array sizes, but partition it in rectangular blocks, and process all the cells in one block at a time.

22.0.41 Prefix vs. Postfix Operators

Prefer prefix operators over postfix operators.

When dealing with primitive types, the prefix and postfix arithmetic operations are likely to have identical performance. With objects, however, postfix operators can cause the object to create a copy of itself to preserve its initial state (to be returned as a result of the operation), as well as causing the side-effect of the operation. Consider the following example:

```
class IntegerIncreaser
{
    int m_Value;

public:
    /* Postfix operator. */
    IntegerIncreaser operator++ (int) {
        IntegerIncreaser tmp (*this);

        ++m_Value;
        return tmp;
    };

    /* Prefix operator. */
    void operator++ () {
        ++m_Value;
    };
};
```

Because the postfix operators are required to return an unaltered version of the value being incremented (or decremented) — regardless of whether the result is actually being used — they will most likely make a copy. STL iterators (for example) are more efficient when altered with the prefix operators.

22.0.42 Explicit inlining

If you don't use the compiler options of whole program optimization and to allow the compiler to inline any function, try to move to the header files the functions called in bottlenecks, and declare them `inline`.

As explained in the guideline "Inlined functions" in section 3.1, every inlined function is faster, but many inlined functions slow down the whole program.

Try to declare `inline` a couple of functions at a time, as long as you get significant speed improvements (at least 10%) in a single command.

22.0.43 Operations with powers of two

If you have to choose an integer constant by which you have to multiply or divide often, choose a power of two.

The multiplication, division, and modulo operations between integer numbers are much faster if the second operand is a constant power of two, as in such case they are implemented as bit shifts or bit maskings.

22.0.44 Integer division by a constant

When you divide an integer (that is known to be positive or zero) by a constant, convert the integer to `unsigned`.

If `s` is a signed integer, `u` is an unsigned integer, and `C` is a constant integer expression (positive or negative), the operation `s / C` is slower than `u / C`, and `s % C` is slower than `u % C`. This is most significant when `C` is a power of two, but in all cases, the sign must be taken into account during division.

The conversion from signed to unsigned, however, is free of charge, as it is only a reinterpretation of the same bits. Therefore, if `s` is a signed integer that you *know* to be positive or zero, you can speed up its division using the following (equivalent) expressions: `(unsigned)s / C` and `(unsigned)s % C`.

22.0.45 Processors with reduced data bus

If the data bus of the target processor is smaller than the processor registers, if possible, use integer types not larger than the data bus for all the variables except for function parameters and for the most used local variables.

The types `int` and `unsigned int` are the most efficient, after they have been loaded in processor registers. Though, with some processor families, they could not be the most efficient type to access in memory.

For example, there are processors having 16-bit registers, but an 8-bit data bus, and other processors having 32-bit registers, but 16-bit data bus. For processors having the data bus smaller than the internal registers, usually the types `int` and `unsigned int` match the size of the registers.

For such systems, loading and storing in memory an `int` object takes a longer time than that taken by an integer not larger than the data bus.

The function arguments and the most used local variables are usually allocated in registers, and therefore do not cause memory access.

22.0.46 Rearrange an array of structures as several arrays

Instead of processing a single array of aggregate objects, process in parallel two or more arrays having the same length.

For example, instead of the following code:

```
const int n = 10000;
struct { double a, b, c; } s[n];
for (int i = 0; i < n; ++i) {
    s[i].a = s[i].b + s[i].c;
}
```

the following code may be faster:

```
const int n = 10000;
double a[n], b[n], c[n];
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

Using this rearrangement, "a", "b", and "c" may be processed by array processing instructions that are significantly faster than scalar instructions. This optimization may have null or adverse results on some (simpler) architectures.

it:Ottimizzare C++/Ottimizzazione del codice C++/Operazioni veloci²

² <http://it.wikibooks.org/wiki/Ottimizzare%20C%2B%2B%2FOttimizzazione%20del%20codice%20C%2B%2B%2FOperazioni%20veloci>

23 Tools

A list of tools to access the CPU hardware counters:

- OProfile¹ (Linux)
- PAPI² (Linux, various Unices)

1 <http://oprofile.sourceforge.net/news/>
2 <http://icl.cs.utk.edu/papi/>

24 Further reading

- Calling conventions for different C++ compilers and operating systems¹
- Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms²
- "Technical Report on C++ Performance"³ by Dave Abrahams et. al. 2003
- "Code optimization tips for 8-bit microcontrollers"⁴ by Steve Kolokowsky 2009
- "Getting the Least Out of Your C Compiler"⁵ by Jakob Engblom 2001 lists "some of the more common mistakes and how to avoid them, ... An important conclusion is that code that is easy for a human to understand is usually also compiler friendly, contrary to hacker tradition."
- Efficient C++: Performance Programming Techniques; Dov Bulka, David Mayhew⁶
- Inside the C++ Object Model; Stanley B. Lippman⁷

1 http://www.agner.org/optimize/calling_conventions.pdf

2 http://www.agner.org/optimize/optimizing_cpp.pdf

3 <http://www.research.att.com/~bs/performanceTR.pdf>

4 <http://www.edn.com/article/CA6643507.html>

5 <http://user.it.uu.se/~jakob/publications/engblom-esc-sf-2001.pdf>

6 <http://www.amazon.ca/Efficient-C-Performance-Programming-Techniques/dp/0201379503>

7 <http://www.amazon.com/Inside-Object-Model-Stanley-Lippman/dp/0201834545>

25 Contributors

Edits	User
8	lexec1 ¹
5	Auremel ²
132	Carlo.milanesi ³
1	Cmbay ⁴
6	Cyberbisson ⁵
1	Darklama ⁶
17	Dirk Hünninger ⁷
1	Dogthehellrider ⁸
2	Doomdayx ⁹
3	Fishpi ¹⁰
1	Jfmantis ¹¹
1	Mtrenkmann ¹²
2	Panic2k4 ¹³
8	Qwertyus ¹⁴
1	Ramac ¹⁵
2	Ropieur ¹⁶
1	Sigma 7 ¹⁷
3	Whiteknight ¹⁸
1	Zlynx ¹⁹

-
- 1 <http://en.wikibooks.org/w/index.php?title=User:lexec1>
 - 2 <http://en.wikibooks.org/w/index.php?title=User:Auremel>
 - 3 <http://en.wikibooks.org/w/index.php?title=User:Carlo.milanesi>
 - 4 <http://en.wikibooks.org/w/index.php?title=User:Cmbay>
 - 5 <http://en.wikibooks.org/w/index.php?title=User:Cyberbisson>
 - 6 <http://en.wikibooks.org/w/index.php?title=User:Darklama>
 - 7 http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger
 - 8 <http://en.wikibooks.org/w/index.php?title=User:Dogthehellrider>
 - 9 <http://en.wikibooks.org/w/index.php?title=User:Doomdayx>
 - 10 <http://en.wikibooks.org/w/index.php?title=User:Fishpi>
 - 11 <http://en.wikibooks.org/w/index.php?title=User:Jfmantis>
 - 12 <http://en.wikibooks.org/w/index.php?title=User:Mtrenkmann>
 - 13 <http://en.wikibooks.org/w/index.php?title=User:Panic2k4>
 - 14 <http://en.wikibooks.org/w/index.php?title=User:Qwertyus>
 - 15 <http://en.wikibooks.org/w/index.php?title=User:Ramac>
 - 16 <http://en.wikibooks.org/w/index.php?title=User:Ropieur>
 - 17 http://en.wikibooks.org/w/index.php?title=User:Sigma_7
 - 18 <http://en.wikibooks.org/w/index.php?title=User:Whiteknight>
 - 19 <http://en.wikibooks.org/w/index.php?title=User:Zlynx>

List of Figures

- GFDL: Gnu Free Documentation License. <http://www.gnu.org/licenses/fdl.html>
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. <http://creativecommons.org/licenses/by-sa/3.0/>
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. <http://creativecommons.org/licenses/by-sa/2.5/>
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. <http://creativecommons.org/licenses/by-sa/2.0/>
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. <http://creativecommons.org/licenses/by-sa/1.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/deed.en>
- cc-by-2.5: Creative Commons Attribution 2.5 License. <http://creativecommons.org/licenses/by/2.5/deed.en>
- cc-by-3.0: Creative Commons Attribution 3.0 License. <http://creativecommons.org/licenses/by/3.0/deed.en>
- GPL: GNU General Public License. <http://www.gnu.org/licenses/gpl-2.0.txt>
- LGPL: GNU Lesser General Public License. <http://www.gnu.org/licenses/lgpl.html>
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. <http://artlibre.org/licence/lal/de>
- CFR: Copyright free use.

- EPL: Eclipse Public License. <http://www.eclipse.org/org/documents/epl-v10.php>

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses²⁰. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrowser.

²⁰ Chapter 26 on page 95

26 Licenses

26.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assure copyright for the work, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those

activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable and exclusive provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work that the User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a form that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of This License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright

holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR

LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty, and each file should have at least the "copyright" line and a pointer to where the full notice is found.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History"). To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties; any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest on adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- * A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- * C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- * D. Preserve all the copyright notices of the Document.
- * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- * H. Include an unaltered copy of this License.
- * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions if they were used. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retittle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

<program> Copyright (C) <year> <name of author>. This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lpl.html>.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation, with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

26.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

26.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code

under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms

that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.