Optimizing C++/Print Version

Optimizing C++/Cover

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, optimization generally does not mean writing optimally performing software; rather, it means incrementally changing (refactoring) 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.

Optimization life cycle

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

1. **Design**. 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 structure 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**. Then the resulting software is tested, to increase the probability that it doesn't have crippling defects.
4. **Optimization** (aka *Tuning*). After having completed the development of a correctly working application, the optimization stage begins, having the following sub-stages:
    1. **Performance testing**. 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.
    2. **Profiling** (aka *Performance analysis*). For every command with inadequate performance, a profiler is used to determine which portions of code are the so-called *bottlenecks* for that command. Bottlenecks are the portions of code where a disproportionately large amount of time is spent or memory space allocated.
    3. **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* sorting algorithm is chosen instead of the *selection sort* algorithm. If this optimization makes the program fast enough, the optimization stage is complete.
    4. **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 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.
    5. **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.
6. **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".

Conventions

By **object**, 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 fits in a single mass storage cluster of blocks.

For example, an array of `double`s 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.

Writing Efficient Code

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.

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.

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 `int`s with `short`s or `signed char`s, replacing the `unsigned int`s with `unsigned short`s or `unsigned char`s, and replacing the `double`s with `float`s. 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.

Function-objects

Instead of passing a function pointer as an argument to a function, pass a

function-object (or, if using the C++0x 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.

`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.

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`).

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.

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.

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 be declared `inline` at the optimization stage.

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.)

`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.

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.

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.

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 then 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.

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))).

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.

`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.

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.

The `throw` operator

Call the `throw` operator only when you want 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 is 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 to be performed too frequently.

`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 constructors and 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.

`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.

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.

Use of automatic deallocators

Use a memory manager based on garbage-collection or a kind of reference-counted smart-pointer (like `shared_ptr` in the Boost (http://www.boost.org/) 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 leaks. 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.

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.

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.

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.

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.

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.

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 (templatized) 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.

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.

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 `double`s, it could become more efficient pass by reference.

`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.

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.

The *Pimpl* idiom

Use the *Pimpl* 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.

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 of 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 Generator (http://www.sgi.com/tech/stl/RandomNumberGenerator.html) 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
                return r;
        }
};
```

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.

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.

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 Boost (http://www.boost.org) library.

`vector`s 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.

Allocating many small objects

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

A *block allocator* (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.

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. Boost provides two customizable block allocators, `pool_allocator` and `fast_pool_allocator` (http://www.boost.org/doc/libs/1_38_0/libs/pool/doc/interfaces/pool_alloc.html) . 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.

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 `vector`s, 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`.

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.

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;
        }
    }
}
```

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 *paddings* (or *holes*) between successive objects.

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

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.

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.

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.

Thread usage

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.

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.

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.

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.

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.

General Optimization Techniques

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/Output
2. Memoization
3. Sorting
4. Other techniques

Output

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.

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.

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.

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 file, if your operating system provides such feature.

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_;
#if defined(__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 clo
  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 phisical file to the specified number of bytes
  The size of the resulting file can be retrieved using "capacity".
  Memory mapped files cannot be shrinked;
  a value smaller than the current capacity is ignored.
  The "capacity()" function return the size the physical file has at thi
  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_;
#if defined(__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"
#if defined(__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),
#if defined(__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() {
#if defined(__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),
#if defined(__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 |
    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 :
    ::ftruncate(file_handle_, adjusted_file_size);
    data_ = static_cast<char*>(::mmap(
        0, adjusted_file_size, PROT_READ | PROT_WRITE, MAP_SHARED, file_
    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 :
    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;
#if defined(__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
    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() {
```

```
#if defined(__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() {
#if defined(__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;
    }
}
```

Memoization

Memoization techniques (aka *caching* techniques) are based on the principle that if you must repeatedly compute a pure function, that is a *referentially transparent* function (aka mathematical function), 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.

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*. 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.

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 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 thread.

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.

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.

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.

Sorting

The C++ Standard Template Library (STL) provides the template function `sort` that implements a comparison sort algorithm. Because `sort` is templatized, 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 introsort algorithm (since the 2000 release; the GNU C++ library uses the reference implementation). This algorithm is a very fast combination of quicksort and heapsort with a specially designed selection algorithm.

The `sort` template function is not guaranteed to be stable. 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.

Sorting with small ranges of keys

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

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 sort 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;
}
```

Partial sorting

Partitioning

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

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)).

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.

Order partitioning

If you have to pick out the first N elements from a sequence, or the Nth 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 then the `std::sort` algorithm, as it has O(N) complexity, instead of O(N log(N)).

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.

Other techniques

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.

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 searches 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.

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`.

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 list, instead of a doubly-linked list.

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.

Code Optimization

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 deallocations
2. Run-time support
3. Instruction count
4. Constructions and destructions
5. Pipeline
6. Memory access
7. Faster operations

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.

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.

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.

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.

Keep `vector`s 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 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.

`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, an 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;
```

Run-time support

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

The `typeid` operator

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

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

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.

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.

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 as a slowdown. Avoid the repeated execution of such block inside bottlenecks.

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.

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.

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.

Instruction count

Even the language features that generate inlined code may have a significant cost, as such instruction 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.

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.

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.

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*.

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* 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> {
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.

The *Strategy* design pattern

If an object that implements the *Strategy* 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&) const = 0;
    virtual void run(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:
    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.

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.

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.

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++ FAQ (http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.9)

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

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.

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;
}
```

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:

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

whose purpose is to allows 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.

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 pipeline 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.

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:

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

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

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];
```

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.

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*.

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.

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.

`union`s

In medium or large arrays or collections, use `union`s.

`union`s 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.

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 bits, 31 <= 32) 4 bytes.

For another example, instead of the following array:

```
unsigned 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 bits, 30 <= 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.

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.

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.

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.

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.

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 (http://www-graphics.stanford.edu/~seander/bithacks.html) . 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.

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
}
```

Array cells size

Ensure that the size (resulting from the `sizeof` operator) of non-large cells of arrays or of `vector`s be a power of two, and that the size of large cells of arrays or of `vector`s 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 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 `float`s. Every cell of the first-level array has a size of 512 x 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.

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.

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.

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.

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`.

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.

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.

GNU Free Documentation License

2. VERBATIM COPYING

3. COPYING IN QUANTITY

5. COMBINING DOCUMENTS

How to use this License for your documents