1.
Optimizing software in C++
An optimization guide for Windows, Linux and Mac platforms

By Agner Fog

Contents

1 Introduction

This manual is for advanced programmers and software developers who want to make their software faster. It is assumed that the reader has a good knowledge of the C++ programming language and a basic understanding of how compilers work. The C++ language is chosen as the basis for this manual for reasons explained on page 6 below.

This manual is based mainly on my study of how compilers and microprocessors work. The recommendations are based on the x86 family of microprocessors from Intel and AMD including the 64-bit versions. The x86 processors are used in the most common platforms with Windows, Linux, BSD and Mac OS X operating systems, though these operating systems can also be used with other microprocessors. Many of the advices may apply to other platforms and other compiled programming languages as well.

This is the first in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.

2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.

3. The microarchitecture of Intel and AMD CPU's: An optimization guide for assembly programmers and compiler makers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel and AMD CPU's.

5. Calling conventions for different C++ compilers and operating systems.

The latest versions of these manuals are always available from www.agner.org/optimize.

Those who are satisfied with making software in a high-level language need only to read this first manual. The subsequent manuals are for those who want to go deeper into the technical details of instruction timing, assembly language programming, compiler technology, and microprocessor microarchitecture. A higher level of optimization can sometimes be obtained by the use of assembly language for CPU-intensive code, as described in the subsequent manuals.

Please note that my optimization manuals are used by thousands of people. I simply don't have the time to answer questions from everybody. So please don't send your programming questions to me. You will not get any answer. Beginners are advised to seek information elsewhere and get a good deal of programming experience before trying the techniques in the present manual. There are various discussion forums on the Internet where you can get answers to your programming questions if you cannot find the answers in the relevant books and manuals.

I want to thank the many people who have sent me corrections and suggestions for my optimization manuals. I am always happy to receive new relevant information.

1.1 The costs of optimizing

University courses in programming nowadays stress the importance of structured and object-oriented programming, modularity, reusability and systematization of the software development process. These requirements are often conflicting with the requirements of optimizing the software for speed or size.

Today, it is not uncommon for software teachers to recommend that no function or method should be longer than a few lines. A few decades ago, the recommendation was the opposite: Don't put something in a separate subroutine if it is only called once. The reasons for this shift in software writing style are that software projects have become bigger and more complex, that there is more focus on the costs of software development, and that computers have become more powerful.

The high priority of structured software development and the low priority of program efficiency is reflected, first and foremost, in the choice of programming language and interface frameworks. This is often a disadvantage for the end user who has to invest in ever more powerful computers to keep up with the ever bigger software packages and who is still frustrated by unacceptably long response times, even for simple tasks.

Sometimes it is necessary to compromise on the advanced principles of software development in order to make software packages faster and smaller. This manual discusses how to make a sensible balance between these considerations. It is discussed how to identify and isolate the most critical part of a program and concentrate the optimization effort on that particular part. It is discussed how to overcome the dangers of a relatively primitive programming style that doesn't automatically check for array bounds violations, invalid pointers, etc. And it is discussed which of the advanced programming constructs are costly and which are cheap, in relation to execution time.

2 Choosing the optimal platform

2.1 Choice of hardware platform

The choice of hardware platform has become less important than it used to be. The distinctions between RISC and CISC processors, between PC's and mainframes, and between simple processors and vector processors are becoming increasingly blurred as the standard PC processors with CISC instruction sets have got RISC kernels, vector processing instructions, multiple kernels, and a processing speed exceeding that of yesterday's big mainframe computers.

Today, the choice of hardware platform for a given task is often determined by considerations such as price, compatibility, second source, and the availability of good development tools, rather than by the processing power. Connecting several standard PC's in a network may be both cheaper and more efficient than investing in a big mainframe computer. Big supercomputers with massively parallel vector processing capabilities still have a niche in scientific computing, but for most purposes the standard PC processors are preferred because of their superior performance/price ratio.

The CISC instruction set (called x86) of the standard PC processors is certainly not optimal from a technological point of view. This instruction set is maintained for the sake of backwards compatibility with a lineage of software that dates back to around 1980 where RAM memory and disk space were scarce resources. However, the CISC instruction set is better than its reputation. The compactness of the code makes caching more efficient today where cache size is a limited resource. The CISC instruction set may actually be better than RISC in situations where code caching is critical. The worst problem of the x86 instruction set is the scarcity of registers. This problem has been alleviated in the new 64-bit extension to the x86 instruction set where the number of registers has been doubled.

The choice of platform is obviously influenced by the requirements of the task in question. For example, a heavy graphics application is preferably implemented on a platform with a graphics coprocessor or graphics accelerator card.

Thin clients that depend on network resources are not recommended for critical applications because the response times for network resources cannot be controlled.

This manual is based on the standard PC platform with an Intel or AMD processor and a Windows, Linux, BSD operating system running in 32-bit or 64-bit mode or an Intel-based Mac computer. Many of the advices given here may apply to other platforms as well, but the examples have been tested only on PC platforms.

2.2 Choice of microprocessor

The benchmark performance of competing brands of microprocessors are very similar thanks to heavy competition. The Intel Pentium 4 and other processors with the Intel NetBurst architecture are good for memory-intensive applications but poor for CPU-intensive applications. Intel Pentium M, Intel Core Duo and AMD Athlon and Opteron processors are better for CPU-intensive applications. The Intel Core 2 processor gives the best performance for code that uses vector operations.

Processors with multiple kernels are advantageous for applications that can be divided into multiple threads that run in parallel.

Some systems have a graphics processing unit, usually on a graphics card. Such units can be used as coprocessors to take care of some of the heavy graphics calculations. In some cases it is possible to utilize the computational power of the graphics processing unit for other purposes than it is intended for. Some systems also have a physics processing unit intended for calculating the movements of objects in computer games. Such a coprocessor might also be used for other purposes. The use of coprocessors is beyond the scope of this manual.

2.3 Choice of operating system

Modern microprocessors in the x86 family can run in both 16-bit and 32-bit mode. The newest processors can also run in 64-bit mode.

16-bit mode is used in the old operating systems DOS and Windows 3.x. These systems require segmentation of the memory if the size of program or data exceeds 64 Kbytes. The modern microprocessors are not optimized for 16-bit mode and some operating systems are not backwards compatible with 16-bit programs. It is not recommended to make 16-bit programs, except for small embedded systems.

At the time of writing (summer 2006), the 32-bit systems are dominating. 64-bit processors are only slowly penetrating the market and 64-bit operating systems are rare. There is no heavy marketing of 64-bit software yet. There is no doubt, however, that the 64-bit systems will dominate in the future.

The 64-bit systems can be expected to give better performance than 32-bit systems for applications that are CPU-intensive or memory-intensive.

A software developer may choose to make time-critical software in two versions. A 32-bit version for the sake of compatibility with existing systems and a 64-bit version for best performance.

The Windows and Linux operating systems give almost identical performance for 32-bit software because the two operating systems are using the same function calling conventions. FreeBSD, Open BSD and Intel-based Mac OS are identical to Linux in almost all respects relevant to software optimization. Everything that is said here about Linux also applies to these systems.

64 bit systems are more efficient than 32 bit systems for the following reasons:

- The number of registers is doubled. This makes it possible to store intermediate data and local variables in registers rather than in memory.
- The size of the integer registers is extended to 64 bits. This is only an advantage in applications that can take advantage of 64-bit integers.
- Function parameters are transferred in registers rather than on the stack. This makes function calls more efficient.
- The allocation and deallocation of big memory blocks is more efficient.
- The SSE2 instruction set is supported on all 64-bit CPU's and operating systems.

A disadvantage of 64-bit systems is that pointers, references, and stack entries use 64 bits rather than 32 bits. This makes data caching slightly less efficient. The 64-bit systems offer a choice between a small memory model where the size of code and static data is limited to 2 Gbytes and a large memory model without this limitation. I cannot think of any use for the less efficient large memory model because the size of dynamic and automatic data is not limited to 2 Gbytes anyway.

The similarity between the operating systems disappears when running in 64-bit mode because the function calling conventions are different. 64-bit Windows allows only four function parameters to be transferred in registers, whereas 64-bit Linux allows up to fourteen parameters to be transferred in registers (6 integer and 8 floating point). There are also other details that make function calling more efficient in 64-bit Linux than in 64-bit Windows (See page 38 and manual 5: "Calling conventions for different C++ compilers and operating systems"). An application with many function calls may run slightly faster in 64-bit Linux than in 64-bit Windows. The disadvantage of 64-bit Windows may be mitigated by making critical functions inline or static or by using a compiler that can do whole program optimization.

2.4 Choice of programming language

Before starting a new software project, it is important to decide which programming language is best suited for the project at hand. Low-level languages are good for optimizing execution speed or program size, while high-level languages are good for making clear and well-structured code and for fast and easy development of user interfaces and interfaces to network resources, databases, etc.

The efficiency of the final application depends on the way the programming language is implemented. The highest efficiency is obtained when the code is compiled and distributed as binary executable code. Most implementations of C++, Pascal and Fortran are based on compilers.

Several other programming languages are implemented with interpretation. The program code is distributed as it is and interpreted line by line when it is run. Examples include JavaScript, PHP, ASP and UNIX shell script. Interpreted code is very inefficient because the body of a loop is interpreted again and again for every iteration of the loop.

Some implementations use just-in-time compilation. The program code is distributed and stored as it is, and is compiled when it is executed. An example is Perl.

Several modern programming languages use an intermediate code (byte code). The source code is compiled into an intermediate code, which is the code that is distributed. The intermediate code cannot be executed as it is, but must go through a second step of

interpretation or compilation before it can run. Most implementations of Java are based on an interpreter which interprets the intermediate code by emulating the so-called Java virtual machine. C#, managed C++, and other languages in Microsoft's .NET framework are based on just-in-time compilation of an intermediate code.

The reason for using an intermediate code is that it is intended to be platform-independent and compact. The biggest disadvantage of using an intermediate code is that the user must install a large runtime framework for interpreting or compiling the intermediate code. This framework typically uses much more resources than the code itself.

The history of programming languages and their implementations reveal a zigzag course that reflects the conflicting considerations of efficiency, platform independence, and easy development. For example, the first PC's had an interpreter for Basic. A compiler for Basic soon became available because the interpreted version of Basic was too slow. Today, the most popular version of Basic is Visual Basic .NET, which is implemented with an intermediate code and just-in-time compilation. Some early implementations of Pascal used an intermediate code like the one that is used for Java today. But this language gained remarkably in popularity when a genuine compiler became available.

It should be clear from this discussion that the choice of programming language is a compromise between efficiency, portability and development time. Interpreted languages such as Java are out of the question when efficiency is important. A language based on intermediate code and just-in-time compilation may be a viable compromise when portability and ease of development are more important than speed. This includes languages such as C# and Visual Basic .NET. However, these languages have the disadvantage of a very large runtime framework that must be loaded every time the program is run. The time it takes to load the framework and compile the program are often much more than the time it takes to execute the program, and the runtime framework may use more resources than the program itself when running. Programs using such a framework sometimes have unacceptably long response times for simple tasks like pressing a button or moving the mouse. The .NET framework should definitely be avoided when speed is critical.

The fastest execution is no doubt obtained with a fully compiled code. Compiled languages include C++, Pascal, Fortran and several other less well-known languages. My preference is for C++ for several reasons. C++ is supported by some very good compilers and optimized function libraries. C++ is an advanced high-level language with a wealth of advanced features rarely found in other languages. But the C++ language also includes the low-level C language as a subset, giving access to low-level optimizations. Most C++ compilers are able to generate an assembly language output, which is useful for checking how well the compiler optimizes a piece of code. Furthermore, most C++ compilers allow assembly-like intrinsic functions, inline assembly or easy linking to assembly language modules when the highest level of optimization is needed. The C++ language is portable in the sense that C++ compilers exist for all major platforms. Pascal has many of the advantages of C++ but is not quite as versatile. Fortran is also quite efficient, but the syntax is very old-fashioned.

Development in C++ is quite efficient thanks to the availability of powerful development tools. One popular development tool is Microsoft Visual Studio. This tool can make two different implementations of C++, directly compiled code and intermediate code for the common language runtime of the .NET framework. Obviously, the directly compiled version is preferred.

An important disadvantage of C++ relates to security. There are no checks for array bounds violation, integer overflow, and invalid pointers. The absence of such checks makes the code execute faster than other languages that do have such checks. But it is the responsibility of the programmer to make explicit checks for such errors in cases where they cannot be ruled out by the program logic. Some guidelines are provided below, on page 11.

C++ is definitely the preferred programming language when the optimization of performance has high priority. The gain in performance over other programming languages can be quite substantial. This gain in performance can easily justify a possible minor increase in development time when performance is important to the end user.

There may be situations where a high level framework based on intermediate code is needed for other reasons, but part of the code still needs careful optimization. A mixed implementation can be a viable solution in such cases. The most critical part of the code can be implemented in compiled C++ or assembly language and the rest of the code, including user interface etc., can be implemented in the high level framework. The optimized part of the code can possibly be compiled as a dynamic link library (DLL) which is called by the rest of the code. This is not an optimal solution because the high level framework still consumes a lot of resources, and the transitions between the two kinds of code gives an extra overhead which consumes CPU time. But this solution can still give a considerable improvement in performance if the time-critical part of the code can be completely contained in a DLL.

2.5 Choice of compiler

There are several different C++ compilers to choose between. It is difficult to predict which compiler will do the best job optimizing a particular piece of code. Each compiler does some things very smart and other things very stupid. Some common compilers are mentioned below.

Microsoft Visual Studio

This is a very user friendly compiler with many facilities, but also very expensive. A limited "express" edition is available for free. Visual Studio can build code for the .NET framework as well as directly compiled code. (Compile without the Common Language Runtime, CLR, to produce binary code). Supports 32-bit and 64-bit Windows. The integrated development environment (IDE) supports multiple programming languages, profiling and debugging. A command-line version of the C++ compiler is available for free in the Microsoft platform software development kit (PSDK). Optimizes reasonably well.

Borland C++ builder

Has an IDE with many of the same features as the Microsoft compiler. Supports only 32-bit Windows. Does not support the newest instruction sets. Does not optimize as good as the Microsoft, Intel and Gnu compilers.

Intel C++ compiler

This compiler does not have its own IDE. It is intended as a plug-in to Microsoft Visual Studio when compiling for Windows and to Eclipse when compiling for Linux. It works with both old and new versions of Microsoft C++ compilers. It can also be used as a stand alone compiler when called from a command line or a make utility. It supports 32-bit and 64-bit Windows and 32-bit and 64-bit Linux, as well as Intel-based Mac OS and Itanium systems.

The Intel compiler has a number of important optimization features:

- Very good support for vector operations using the single-instruction-multiple-data capabilities of the latest instruction sets for Intel and AMD microprocessors. This compiler can change simple code to vector code automatically (see page 87).

- Very good support for parallel processing on systems with multiple processors or multi-kernel processors. Can do automatic parallelization or explicit parallelization using the OpenMP directives.

- Supports CPU dispatch to make multiple code versions for different CPU's.
- Comes with an optimized math function library.
- Excellent support for inline assembly on all platforms and the possibility of using the same inline assembly syntax in both Windows and Linux.

The Intel compiler obviously optimizes the code for best performance on Intel microprocessors, but it also works very well with AMD processors.

The Intel compiler is a good choice for code that can benefit from its many optimization features and for code that is ported to multiple operating systems.

Gnu

This is a good and free open source compiler. It comes with most distributions of Linux, 32-bit and 64-bit. Available for many platforms, including 32-bit Windows but not 64-bit Windows. Optimizes very well.

Digital Mars

This is a cheap open source compiler for 32-bit Windows, including an IDE. Does not optimize well.

Open Watcom

Another open source compiler for 32-bit Windows. Does not, by default, conform to the standard calling conventions. Optimizes reasonably well.

Codeplay VectorC

A commercial compiler for 32-bit Windows. Integrates into the Microsoft Visual Studio IDE. Has not been updated since 2004. Can do automatic vectorization. Optimizes moderately well. Supports three different object file formats.

Comments

All of these compilers can be used as command-line versions without an IDE. The command line versions are either free or available as noncommercial or trial versions.

Object files and library files produced by the Intel compiler are fully compatible with Microsoft or Gnu compilers. Object files produced by the Digital Mars and Codeplay compilers are mostly compatible with the Microsoft compiler. Otherwise, the compilers are not compatible on the object file level.

My recommendation for good code performance is to use the Gnu or Intel compiler for Linux applications and the Intel or Microsoft compiler for Windows applications. Use the Intel compiler if your application can benefit from automatic vectorization (see page 86), automatic parallelization (see page 83) or if you want automatic CPU dispatch (see page 105).

The choice of compiler may in some cases be determined by the requirements of compatibility with legacy code, specific preferences for the IDE, for debugging facilities, easy GUI development, database integration, web application integration, mixed language programming, etc. In cases where the chosen compiler doesn't provide the best optimization it may be useful to make the most critical modules with a different compiler. Object files generated by the Intel compiler can in most cases be linked into projects made with Microsoft or Gnu compilers without problems if the necessary library files are also included. Combining Intel and Borland compilers is more difficult. The functions must have `extern "C"` declaration and the object files need to be converted to OMF format. Alternatively, make a DLL with the best compiler and call it from a project built with another compiler.

2.6 Choice of user interface framework

Most of the code in a typical software project goes to the user interface. Applications that are not computationally intensive may very well spend more CPU time on the user interface than on the essential task of the program.

Application programmers rarely program their own graphical user interfaces from scratch. This would not only be a waste of the programmers' time, but also inconvenient to the end user. Menus, buttons, dialog boxes, etc. should be as standardized as possible for usability reasons. The programmer can use standard user interface elements that come with the operating system or libraries that come with compilers and development tools.

A popular user interface library for Windows and C++ is Microsoft Foundation Classes (MFC). A competing product is Borland's now discontinued Object Windows Library (OWL). Several graphical interface frameworks are available for Linux systems. The user interface library can be linked either as a runtime DLL or a static library. A runtime DLL takes more memory resources than a static library, except when several applications use the same DLL at the same time.

A user interface library may be bigger than the application itself and take more time to load. A light-weight alternative is the Windows Template Library (WTL). A WTL application is generally faster and more compact than an MFC application. The development time for WTL applications can be expected to be higher due to poor documentation and lack of advanced development tools.

The simplest possible user interface is obtained by dropping the graphical user interface and use a console mode program. The inputs for a console mode program are typically specified on a command line or an input file. The output goes to the console or to an output file. A console mode program is fast, compact, and simple to develop. It is easy to port to different platforms because it doesn't depend on system-specific graphical interface calls. The usability may be poor because it lacks the self-explaining menus of a graphical user interface. A console mode program is useful for calling from other applications such as a make utility.

The conclusion is that the choice of user interface framework must be a compromise between development time, usability, program compactness, and execution time. No universal solution is best for all applications.

2.7 Overcoming the drawbacks of the C++ language

While C++ has many advantages when it comes to optimization, it does have some disadvantages that make developers choose other programming languages. This section discusses how to overcome these disadvantages when C++ is chosen for the sake of optimization.

Portability

C++ is fully portable in the sense that the syntax is fully standardized and supported on all major platforms. However, C++ is also a language that allows direct access to hardware interfaces and system calls. These are of course system-specific. In order to facilitate porting between platforms, it is recommended to put the user interface and other system-specific parts of the code in a separate module, and put the task-specific part of the code, which supposedly is system-independent, in another module.

The size of integers and other hardware-related details depend on the hardware platform and operating system. See page 21 for details.

Development time

Some developers feel that a particular programming language and development tool is faster to use than others. While some of the difference is simply a matter of habit, it is true that some development tools have powerful facilities that do much of the trivial programming work automatically. The development time and maintainability of C++ projects can be improved by consistent modularity and reusable classes.

Security

The most serious problem with the C++ language relates to security. Standard C++ implementations have no checking for array bounds violations and invalid pointers. This is a frequent source of errors in C++ programs and also a possible point of attack for hackers. It is necessary to adhere to certain programming principles in order to prevent such errors in programs where security matters.

Problems with invalid pointers can be avoided by using references instead of pointers, by initializing pointers to zero, by setting pointers to zero whenever the objects they point to become invalid, and by avoiding pointer arithmetics and pointer type casting. Linked lists and other data structures that typically use pointers may be replaced by more efficient container class templates, see page 121. Avoid the function `scanf`.

Violation of array bounds is probably the most common of all errors in C++ programs. Writing past the end of an array can cause other variables to be overwritten. And even worse, it can overwrite the return address of the function in which the array is defined. This can cause all kinds of strange and unexpected behaviors. A good way to prevent such errors is to replace arrays by well-tested container classes. A container class for an array with bounds-checking is provided on page 121. The standard template library (STL) is a useful source of such container classes. Unfortunately, many standard container classes use dynamic memory allocation, which is quite inefficient. See page and 75 and 121 for examples of how to avoid dynamic memory allocation.

Arrays are often used as buffers for storing text or input data. It is important to check the length of any data before copying it into a buffer, especially if it comes from an unverified source. Storing text strings in character arrays is efficient, but unsafe if the length of the string is not checked.

You may deviate from these security advices in critical parts of the code where speed is important. This can be permissible if the unsafe code is limited to well-tested functions, classes, templates or modules with a well-defined interface to the rest of the program.

3 Finding the biggest time consumers

3.1 How much is a clock cycle?

In this manual, I am using CPU clock cycles rather than seconds or microseconds as a time measure. This is because computers have very different speeds. If I write that something takes 10 µs today, then it may take only 5 µs on the next generation of computers and my manual will soon be obsolete. But if I write that something takes 10 clock cycles then it will still take 10 clock cycles even if the CPU clock frequency is doubled.

The length of a clock cycle is the reciprocal of the clock frequency. For example, if the clock frequency is 2 GHz then the length of a clock cycle is

$$\frac{1}{2\text{GHz}} = 0.5\text{ns}.$$

A clock cycle on one computer is not always comparable to a clock cycle on another computer. The Pentium 4 (NetBurst) CPU is designed for a higher clock frequency than other CPU's, but it uses more clock cycles than other CPU's for executing the same piece of code in general.

Assume that a loop in a program repeats 1000 times and that there are 100 floating point operations (addition, multiplication, etc.) inside the loop. If each floating point operation takes 5 clock cycles, then we can roughly estimate that the loop will take 1000 * 100 * 5 * 0.5 ns = 250 µs on a 2 GHz CPU. Should we try to optimize this loop? Certainly not! 250 µs is less than 1/50 of the time it takes to refresh the screen. There is no way the user can see the delay. But if the loop is inside another loop that also repeats 1000 times then we have an estimated calculation time of 250 ms. This delay is just long enough to be noticeable but not long enough to be annoying. We may decide to do some measurements to see if our estimate is correct or if the calculation time is actually more than 250 ms. If the response time is so long that the user actually has to wait for a result then we will consider if there is something that can be improved.

3.2 Use a profiler to find hot spots

Before you start to optimize anything, you have to identify the critical parts of the program. In some programs, more than 99% of the time is spent in the innermost loop doing mathematical calculations. In other programs, 99% of the time is spent on reading and writing data files while less than 1% goes to doing something on these data. It is very important to optimize the parts of the code that matters rather than the parts of the code that use only a small fraction of the total time. Optimizing less critical parts of the code will not only be a waste of time, it also makes the code less clear and more difficult to debug and maintain.

Some compiler packages include a profiler that can tell how many times each function is called and how much time it uses. Some profilers can give even more information than that. Intel's VTune and AMD's CodeAnalyst are tools that can find hot spots and identify problems such as cache misses, unaligned data, floating point exceptions, branch mispredictions, etc.

Profilers are not always reliable. They may give misleading results if the total execution time is low or if the program spends time waiting for input. It may be necessary to make a test program that works on predefined test data without waiting for user input in order to get reliable profiling results.

If you don't have a profiler and if it is not obvious which part of the code is most critical, then set up a number of counter variables that are incremented at different places in the code to see which part is executed most times.

If the program has a critical hot spot then you may isolate this hot spot in a separate subroutine or module that can be tested separately and optimized for speed, while the rest of the program can be optimized for clarity and maintainability. Most profilers are intended for testing a whole program in order to find hot spots. They are less useful for analyzing a small part of the code, once the hot spot has been identified. I am providing some test tools that are designed specifically for testing small parts of a program. See page 119 for details.

A profiler is most useful for finding problems that relate to CPU-intensive code. But many programs use more time loading files or accessing network and other resources than doing arithmetic operations. The most common time-consumers are discussed in the following sections.

3.3 Program installation

The time it takes to install a program package is not traditionally considered a software optimization issue. But it is certainly something that can steal the user's time. The time it takes to install a software package and make it work cannot be ignored if the goal of software optimization is to save time for the user. With the high complexity of modern software, it is not unusual for the installation process to take more than an hour. Neither is it unusual that a user has to reinstall a software package several times in order to find and resolve compatibility problems.

Software developers should take installation time and compatibility problems into account when deciding whether to base a software package on a complex framework requiring many files to be installed.

The installation process should use standardized installation tools. It should be possible to select all installation options at the start so that the rest of the installation process can proceed unattended. Uninstallation should also proceed in a standardized manner.

3.4 Program loading

It often takes more time to load a program than to execute it. The loading time can be annoyingly high for programs that are based on big frameworks and intermediate code, as is commonly the case with programs written in Java, C#, Visual Basic, etc.

But program loading can be a time-consumer even for programs implemented in compiled C++. This typically happens if the program uses a lot of runtime DLL's (dynamically linked libraries, also called shared objects), resource files, configuration files, help files and databases. The operating system may not load all the modules of a big program when the program starts up. Some modules may be loaded only when they are needed, or they may be swapped to the hard disk if the RAM size is insufficient.

The user expects immediate responses to simple actions like a key press or mouse move. It is unacceptable to the user if such a response is delayed for several seconds because it requires the loading of modules or resource files from disc. Memory-hungry applications force the operating system to swap memory to disc. Memory swapping is a frequent cause of unacceptably long response times to simple things like a mouse move or key press.

The best way to make program loading fast is to use static linking of function libraries instead of dynamic link libraries. Many function libraries are available in both static and dynamic versions. Static linking of function libraries have some important advantages over dynamic linking:

1. Static linking includes only the part of the library that is actually needed by the application, while dynamic linking makes the entire library load into memory.

2. All the executable code is included in the same `.exe` file when static linking is used. Dynamic linking makes it necessary to load several files when the program is started.

3. It takes longer time to call a function in a dynamic link library than in a static link library because of extra jumping and because the code cache becomes more fragmented.

4. Installing a second application that uses a newer version of the same library can change the behavior of the first application if dynamic linking is used, but not if static linking is used.

The advantages of dynamic linking are:

1. Multiple applications running simultaneously can share the same DLL without the need to load more than one instance of the DLL into memory.

2. A DLL can be updated to a new version without the need to update the program that calls it.

Avoid an excessive number of DLL's, configuration files, resource files, help files etc. scattered around on the hard disk. A few files, preferably in the same directory as the `.exe` file, is acceptable.

3.5 File access

Reading or writing a file on a hard disk often takes much more time than processing the data in the file. Sequential forward access to a file is faster than random access. Reading or writing big blocks is faster than reading or writing a small bit at a time. Do not read or write less than a few kilobytes at a time.

You may mirror the entire file in a memory buffer and read or write it in one operation rather than reading or writing small bits in a non-sequential manner.

It is usually much faster to access a file that has been accessed recently than to access it the first time. This is because the file has been copied to the disk cache.

Files on removable media such as floppy disks and USB sticks may not be cached. This can have quite dramatic consequences. I once made a Windows program that wrote a file by calling `WritePrivateProfileString`, which opens and closes the file for each line. This worked sufficiently fast on a hard disk because of disk caching, but it took several minutes to write the file to a floppy disk.

A big file containing numerical data is more compact and efficient if the data are stored in binary form than if the data are stored in ASCII form. A disadvantage of binary data storage is that it is not human readable.

Optimizing file access is more important than optimizing CPU use in programs that have many file input/output operations. It can be advantageous to put file access in a separate thread if there is other work that the processor can do while waiting for disc operations to finish.

3.6 System database

It can take several seconds to access the system database in Windows. It is more efficient to store application-specific information in a separate file than in the big registration database in the Windows system. Note that the system may store the information in the database anyway if you are using functions such as `GetPrivateProfileString` and `WritePrivateProfileString` to read and write configuration files.

3.7 Other system resources

Writes to the screen, printer, etc. should preferably be done in big blocks rather than a small piece at a time because each call to a driver involves a large overhead of switching to protected mode and back again.

Accessing system devices and using advanced facilities of the operating system can be time consuming because it may involve the loading of several drivers, configuration files and system modules.

3.8 Network access

Some application programs use internet or intranet for automatic updates, remote help files, data base access, etc. The problem here is that access times cannot be controlled. The network access may be fast in a test situation but slow or completely absent in a use situation.

These problems should be taken into account when deciding whether to store help files and other resources locally or remotely. If frequent updates are necessary then it may be optimal to mirror the remote data locally.

Some applications have automatic updates of program files, data files, etc. The downloading of updates should proceed in a separate thread with low priority so that it doesn't disturb the user. The installation of downloaded program file updates should be postponed until the program is restarted anyway.

3.9 Memory access

Accessing data from RAM memory can take quite a long time compared to the time it takes to do calculations on the data. This is the reason why all modern computers have memory caches. Typically, there is a level-1 data cache of 8 - 64 Kbytes and a level-2 cache of 256 Kbytes to 2 Mbytes.

If the combined size of all data in a program is bigger than the level-2 cache and the data are scattered around in memory or accessed in a non-sequential manner then it is likely that memory access is the biggest time-consumer in the program. Reading or writing to a variable in memory takes only 2-3 clock cycles if it is cached, but several hundred clock cycles if it is not cached. See page 18 about data storage and page 72 about memory caching.

3.10 Context switches

A context switch is a switch between different tasks in a multitasking environment, between different threads in a multithreaded program, or between different parts of a big program. Frequent context switches can reduce the performance because the contents of data cache, code cache, branch target buffer, branch pattern history, etc. may have to be renewed.

Context switches are more frequent if the time slices allocated to each task or thread are smaller. The lengths of the time slices is determined by the operating system, not by the application program.

The number of context switches is smaller in a computer with multiple CPU's or a CPU with multiple cores.

3.11 Dependence chains

Modern microprocessors can do out-of-order execution. This means that if a piece of software specifies the calculation of A and then B, and the calculation of A is slow, then the microprocessor can begin the calculation of B before the calculation of A is finished. Obviously, this is only possible if the value of A is not needed for the calculation of B.

In order to take advantage of out-of-order execution, you have to avoid long dependence chains. A dependence chain is a series of calculations, where each calculation depends on the result of the preceding one. This prevents the CPU from doing multiple calculations simultaneously or out of order. See page 84 for examples of how to break a dependence chain.

3.12 Execution unit throughput

There is an important distinction between the latency and the throughput of an execution unit. For example, it takes five clock cycles to do a floating point addition on a Pentium 4. But it is possible to start a new floating point addition every clock cycle. This means that if each addition depends on the result of the preceding addition then you will have only one addition every five clock cycles. But if all the additions are independent then you can have one addition every clock cycle.

The highest performance that can possibly be obtained in a computationally intensive program is achieved when none of the time-consumers mentioned in the above sections are dominating and there are no long dependence-chains. In this case, the performance is limited by the throughput of the execution units rather than by the latency or by memory access.

The execution core of modern microprocessors is split between several execution units. Typically, there are two or more integer units, one floating point addition unit, and one floating point multiplication unit. This means that it is possible to do an integer addition, a floating point addition, and a floating point multiplication at the same time.

A code that does floating point calculations should therefore preferably have a balanced mix of additions and multiplications. Subtractions use the same unit as additions. Divisions take longer time and use the multiplication unit. It is possible to do integer operations in-between the floating point operations without reducing the performance because the integer operations use a different execution unit. For example, a loop that does floating point calculations will typically use integer operations for incrementing a loop counter, comparing the loop counter with its limit, etc. In most cases, you can assume that these integer operations do not add to the total computation time.

4 Performance and usability

A better performing software product is one that saves time for the user. Time is a precious resource for many computer users and much time is wasted on software that is slow, difficult to use, incompatible or error prone. All these problems are usability issues, and I believe that software performance should be seen in the broader perspective of usability. A list of literature on usability is given on page 132.

This is not a manual on usability, but I think that it is necessary here to draw the attention of software programmers to some of the most common obstacles to efficient use of software. The following list points out some typical sources of frustration and waste of time for software users as well as important usability problems that software developers should be aware of.

- Big runtime frameworks. The .NET framework and the Java virtual machine are frameworks that typically take much more resources than the programs they are running. Such frameworks are frequent sources of resource problems and compatibility problems and they waste a lot of time both during installation of the framework itself, during installation of the program that runs under the framework, during start of the program, and while the program is running. The main reason why such runtime frameworks are used at all is for the sake of cross-platform portability. Unfortunately, the cross-platform compatibility is not always as good as expected. I believe that the portability could be achieved more efficiently by better standardization of programming languages, operating systems, and API's.

- Memory swapping. Software developers typically have more powerful computers with more RAM than end users have. The developers may therefore fail to see the excessive memory swapping and other resource problems that cause the resource-

hungry applications to perform poorly for the end user.

- Installation problems. The procedures for installation and uninstallation of programs should be standardized and done by the operating system rather than by individual installation tools.

- Automatic updates. Automatic updating of software can cause problems if the network is unstable or if the new version causes problem that were not present in the old version. Updating mechanisms often disturb the users with nagging pop-up messages saying please install this important new update or even telling the user to restart the computer while he or she is busy concentrating on important work. The updating mechanism should never interrupt the user but only show a discrete icon signaling the availability of an update, or update automatically when the computer is restarted anyway. Software distributors are often abusing the update mechanism to advertise new versions of their software. This is annoying to the user.

- Compatibility problems. All software should be tested on different platforms, different screen resolutions, different system color settings and different user access rights. Software should use standard API calls rather than self-styled hacks and direct hardware access. Available protocols and standardized file formats should be used. Web systems should be tested in different browsers, different platforms, different screen resolutions, etc. Accessibility guidelines should be obeyed (See literature page 132).

- Copy protection. Some copy protection schemes are based on hacks that violate or circumvent operating system standards. Such schemes are frequent sources of compatibility problems and system breakdown. Many copy protection schemes are based on hardware identification. Such schemes cause problems when the hardware is updated. Most copy protection schemes are annoying to the user and prevent legitimate backup copying without effectively preventing illegitimate copying. The benefits of a copy protection scheme should be weighed against the costs in terms of usability problems and necessary support.

- Hardware updating. The change of a hard disk or other hardware often requires that all software be reinstalled and user settings are lost. It is not unusual for the reinstallation work to take a whole workday or more. Current operating systems need better support for hard disk copying.

- Security. The vulnerability of software with network access to virus attacks and other abuse is extremely costly to many users.

- Background services. Many services that run in the background are unnecessary for the user and a waste of resources. Consider running the services only when activated by the user.

- Take user feedback seriously. User complaints should be regarded as a valuable source of information about bugs, compatibility problems, usability problems and desired new features. User feedback should be handled in a systematic manner to make sure the information is utilized appropriately. Users should get a reply about investigation of the problems and planned solutions. Patches should be easily available from a website.

5 Choosing the optimal algorithm

The first thing to do when you want to optimize a piece of CPU-intensive software is to find the best algorithm. The choice of algorithm is very important for tasks such as sorting,

searching, or mathematical calculations. In such cases, you can obtain much more by choosing the best algorithm than by optimizing the first algorithm that comes to mind. In some cases you may have to test several different algorithms in order to find the one that works best on a typical set of test data.

The discussion of different algorithms is beyond the scope of this manual. You have to consult the general literature on algorithms for standard tasks such as sorting and searching, or the specific literature for more complicated mathematical tasks.

Before you start to code, you may consider whether others have done the job before you. Optimized function libraries for many standard tasks are available from a number of sources. For example, the "Intel Math Kernel Library" contains many functions for common mathematical calculations including linear algebra and statistics, and the "Intel Integrated Performance Primitives" library contains many functions for audio and video processing, signal processing, data compression and cryptography (www.intel.com).

6 The efficiency of different C++ constructs

Most programmers have little or no idea how a piece of program code is translated into machine code and how the microprocessor handles this code. For example, many programmers do not know that double precision calculations are just as fast as single precision. And who would know that a template class is more efficient than a polymorphous class?

This chapter is aiming at explaining the relative efficiency of different C++ language elements in order to help the programmer choosing the most efficient alternative. The theoretical background is further explained in the other volumes in this series of manuals.

6.1 Different kinds of variable storage

Variables and objects are stored in different parts of the memory, depending on how they are declared in a C++ program. This has influence on the efficiency of the data cache (see page 72). Data caching is poor if data are scattered randomly around in the memory. It is therefore important to understand how variables are stored. The storage principles are the same for simple variables, arrays and objects.

Storage on the stack

Variables declared with the keyword `auto` are stored on the stack. The keyword `auto` is practically newer used because automatic storage is the default for all variables and objects that are declared inside any function.

The stack is a part of memory that is organized in a first-in-last-out fashion. It is used for storing function return addresses (i.e. where the function was called from), function parameters, local variables, and for saving registers that have to be restored before the function returns. Every time a function is called, it allocates the required amount of space on the stack for all these purposes. This memory space is freed when the function returns. The next time a function is called, it can use the same space for the parameters of the new function.

The stack is the most efficient place to store data because the same range of memory addresses is reused again and again. If there are no big arrays, then it is almost certain that this part of the memory is mirrored in the level-1 data cache, where it is accessed quite fast.

The lesson we can learn from this is that all variables and objects should preferably be declared inside the function in which they are used.

It is possible to make the scope of a variable even smaller by declaring it inside a {} bracket. However, most compilers do not free the memory used by a variable until the function returns even though it could free the memory when exiting the {} bracket in which the variable is declared.

Global or static storage

Variables that are declared outside of any function are called global variables. They can be accessed from any function. Global variables are stored in a static part of the memory. The static memory is also used for variables declared with the `static` keyword, for floating point constants, string constants, array initializer lists, `switch` statement jump tables, and virtual function tables.

The static data area is usually divided into three parts: one for constants that are never modified by the program, one for initialized variables that may be modified by the program, and one for uninitialized variables that may be modified by the program.

The advantage of static data is that they can be initialized to desired values before the program starts. The disadvantage is that the memory space is occupied throughout the whole program execution, even if the variable is only used in a small part of the program. This makes data caching less efficient.

Do not make variables global if you can avoid it. Global variables may be needed for communication between different threads, but that's about the only situation where they are unavoidable. It may be useful to make a variable global if it is accessed by several different functions and you want to avoid the overhead of transferring the variable as function parameter. But it may be a better solution to make the functions that access the save variable members of the same class and store the shared variable inside the class. Which solution you prefer is a matter of programming style.

It is often preferable to make a lookup-table static. Example:

```
// Example 6.1
float SomeFunction (int x) {
   static float list[] = {1.1, 0.3, -2.0, 4.4, 2.5};
   return list[x];
}
```

The advantage of using `static` here is that the list does not need to be initialized when the function is called. The values are simply put there when the program is loaded into memory. If the word `static` is removed from the above example, then all five values have to be put into the list every time the function is called. This is done by copying the entire list from static memory to stack memory. Copying constant data from static memory to the stack is a waste of time in most cases, but it may be optimal in special cases where the data are used may times in a loop where almost the entire level-1 cache is used in a number of arrays that you want to keep together on the stack.

String constants and floating point constants are stored in static memory. Example:

```
// Example 6.2
a = b * 3.5;
c = d + 3.5;
```

Here, the constant `3.5` will be stored in static memory. Most compilers will recognize that the two constants are identical so that only one constant needs to be stored. All identical constants in the entire program will be joined together in order to minimize the amount of cache space used for constants.

Integer constants are usually included as part of the instruction code. You can assume that there are no caching problems for integer constants.

Register storage

A limited number of variables can be stored in registers instead of main memory. A register is a small piece of memory inside the CPU used for temporary storage. Variables that are stored in registers are accessed very fast. All optimizing compilers will automatically choose the most often used variables in a function for register storage.

The number of registers is very limited. There are approximately six integer registers available for general purposes in 32-bit operating systems and fourteen integer registers in 64-bit systems.

Floating point variables use a different kind of registers. There are eight floating point registers available in 32-bit operating systems and sixteen in 64-bit operating systems. Some compilers have difficulties making floating point register variables in 32-bit mode without the SSE2 instruction set.

Volatile

The `volatile` keyword specifies that a variable cannot be stored in a register, not even temporarily. This is necessary for variables that are accessed by more than one thread. Volatile storage prevents the compiler from doing any kind of optimization on the variable. It is sometimes used for turning off optimization of a particular variable.

Thread-local storage

Most compilers can make thread-local storage of static and global variables by using the keyword `__thread` or `__declspec(thread)`. Such variables have one instance for each thread. Thread-local storage is inefficient because it is accessed through a pointer stored in a thread environment block. Thread-local storage should be avoided, if possible, and replaced by storage on the stack (see above, p. 18). Variables stored on the stack always belong to the thread in which they are created.

Far

Systems with segmented memory, such as DOS and 16-bit Windows, allow variables to be stored in a far data segment by using the keyword `far` (arrays can also be `huge`). Far storage, far pointers, and far procedures are inefficient. If a program has too much data for one segment then it is recommended to use a different operating systems that allows bigger segments (32-bit or 64-bit systems).

Dynamic memory allocation

Dynamic memory allocation is done with the operators `new` and `delete` or with the functions `malloc` and `free`. These operators and functions consume a significant amount of time. A part of memory called the heap is reserved for dynamic allocation. The heap can easily become fragmented when objects of different sizes are allocated and deallocated in random order. The heap manager can spend a lot of time cleaning up spaces that are no longer used and searching for vacant spaces. This is called garbage collection. Objects that are allocated in sequence are not necessarily stored sequentially in memory. They may be scattered around at different places when the heap has become fragmented. This makes data caching inefficient.

Dynamic memory allocation also tends to make the code more complicated and error-prone. The program has to keep pointers to all allocated objects and keep track of when they are no longer used. It is important that all allocated objects are also deallocated in all possible cases of program flow. Failure to do so is a common source of error known as memory leak. An even worse kind of error is to access an object after it has been deallocated. The program logic may need extra overhead to prevent such errors.

See page 75 for a further discussion of the advantages and drawbacks of using dynamic memory allocation.

Some programming languages, such as Java, use dynamic memory allocation for all objects. This is of course inefficient.

Variables declared inside a class

Variables declared inside a class are stored in the order in which they appear in the class declaration. The type of storage is determined where the object of the class is declared. An object of a class, structure or union can use any of the storage methods mentioned above. An object cannot be stored in a register except in the simplest cases, but its data members can be copied into registers.

A class member variable with the `static` modifier will be stored in static memory and will have one and only one instance. Non-static members of the same class will be stored with each instance of the class.

Storing variables in a class or structure is a good way of making sure that variables that are used in the same part of the program are also stored near each other. See page 39 for the pros and cons of using classes.

6.2 Integers variables and operators

Integer sizes

Integers can be different sizes, and they can be signed or unsigned. The following table summarizes the different integer types available.

declaration	size, bits	minimum value	maximum value	comments
`char`	8	-128	127	
`short int`	16	-32768	32767	in 16-bit systems: `int`
`int`	32	-2^{31}	$2^{31}-1$	in 16-bit systems: `long int`
`__int64`	64	-2^{63}	$2^{63}-1$	in Windows: `__int64` in 32-bit Linux: `long long` in 64-bit Linux: `long int`
`unsigned char`	8	0	255	
`unsigned short int`	16	0	65535	in 16-bit systems: `unsigned int`
`unsigned int`	32	0	$2^{32}-1$	in 16-bit systems: `unsigned long`
`unsigned __int64`	64	0	$2^{64}-1$	Windows: `unsigned __int64` 32-bit Linux: `unsigned long long` 64-bit Linux: `unsigned long int`

Table 6.1. Sizes of different integer types

Unfortunately, the way of declaring an integer of a specific size is different for different platforms. Some compilers support the keywords `__int8`, `__int16`, `__int32` and `__int64` for integers of a specific size. On other systems, it is customary to define types like `INT32` in a header file.

Integer operations are fast in most cases, regardless of the size. However, it is inefficient to use an integer size that is larger than the largest available register size. In other words, it is inefficient to use 32-bit integers in 16-bit systems or 64-bit integers in 32-bit systems, especially if the code involves multiplication or division.

The compiler will always select the most efficient integer size if you declare an `int`, without specifying the size. Integers of smaller sizes (`char`, `short int`) are only slightly less efficient. In most cases, the compiler will convert these types to integers of the default size when doing calculations, and then use only the lower 8 or 16 bits of the result. You can assume that the type conversion takes zero or one clock cycle. In 64-bit systems, there is only a minimal difference between the efficiency of 32-bit integers and 64-bit integers, as long as you are not doing divisions.

It is recommended to use the default integer size in cases where the size doesn't matter, such as simple variables, loop counters, etc. In large arrays, it may be preferred to use the smallest integer size that is big enough for the specific purpose in order to make better use of the data cache. Bit-fields of sizes other than 8, 16, 32 and 64 bits are less efficient.

In 64-bit systems, you may use 64-bit integers if the application can make use of the extra bits. There is no automatic overflow check for integer operations.

Integer operators

Integer operations are generally very fast. Simple integer operations such as addition, subtraction, comparison, bit operations and shift operations take only one clock cycle on most microprocessors.

Multiplication and division take longer time. Integer multiplication takes 11 clock cycles on Pentium 4 processors, and 3 - 4 clock cycles on most other microprocessors. Integer division takes 40 - 80 clock cycles, depending on the microprocessor. Integer division is faster the smaller the integer size on AMD processors, but not on Intel processors. Details about instruction latencies are listed in manual 4: "Instruction tables". Tips about how to speed up multiplications and divisions are given on page 110 and 112, respectively.

Increment and decrement operators

The pre-increment operator `++i` and the post-increment operator `i++` are as fast as additions. When used simply to increment a variable, it makes no difference whether you use pre-increment or post-increment. The effect is simply identical. For example, `for (i=0; i<n; i++)` is the same as `for (i=0; i<n; ++i)`. But when the result of the expression is used, then there may be a difference in efficiency. For example, `x = array[i++]` is more efficient than `x = array[++i]` because in the latter case, the calculation of the address of the array element has to wait for the new value of `i` which will delay the availability of `x` for approximately two clock cycles. Obviously, the initial value of `i` must be adjusted if you change pre-increment to post-increment.

There are also situations where pre-increment is more efficient than post-increment. For example, in the case `a = ++b;` the compiler will recognize that the values of `a` and `b` are the same after this statement so that it can use the same register for both, while the expression `a = b++;` will make the values of `a` and `b` different so that they cannot use the same register.

Everything that is said here about increment operators also applies to decrement operators.

6.3 Floating point variables and operators

Modern microprocessors in the x86 family have two different types of floating point registers and correspondingly two different types of floating point instructions. Each type has advantages and disadvantages.

The original method of doing floating point operations involves eight floating point registers organized as a register stack. These registers have long double precision (80 bits). The advantages of using the register stack are:

- All calculations are done with long double precision.
- Conversions between different precisions take no extra time.
- There are intrinsic instructions for mathematical functions such as logarithms and trigonometric functions.
- The code is compact and takes little space in the code cache.

The register stack also has disadvantages:

- It is difficult for the compiler to make register variables because of the way the register stack is organized.
- Floating point comparisons are slow unless the Pentium-II or later instruction set is enabled.
- Conversions between integers and floating point numbers is inefficient.
- Division, square root and mathematical functions take more time to calculate when long double precision is used.

A newer method of doing floating point operations involves eight or sixteen so-called XMM registers which can be used for multiple purposes. Floating point operations are done with single or double precision, and intermediate results are always calculated with the same precision as the operands. The advantages of using the XMM registers are:

- It is easy to make floating point register variables.
- Vector operations are available for doing parallel calculations on vectors of two double-precision or four single-precision variables (see page 86).

Disadvantages are:

- Long double precision is not supported.
- The calculation of expressions where operands have mixed precision require precision conversion instructions which can be quite time-consuming (see page 114).
- Mathematical functions must use a function library, but this is often faster than the intrinsic hardware functions.

The floating point stack registers are available in all systems that have floating point capabilities (except in device drivers for 64-bit Windows). The XMM registers are available in 64-bit systems and in 32-bit systems when the SSE2 or later instruction set is enabled (single precision requires only SSE). See page 105 for how to test for the availability of the SSE or SSE2 instruction set.

Most compilers will use the XMM registers for floating point calculations whenever they are available, i.e. in 64-bit mode or when the SSE2 instruction set is enabled. Few compilers are able to mix the two types of floating point operations and choose the type that is optimal for each calculation.

In most cases, double precision calculations take no more time than single precision. When the floating point registers are used, there is simply no difference in speed between single and double precision. Long double precision takes only slightly more time. Single precision division, square root and mathematical functions are calculated faster than double precision when the XMM registers are used, while the speed of addition, subtraction, multiplication, etc. is still the same regardless of precision.

You may use double precision without worrying too much about the costs if it is good for the application. You may use single precision if you have big arrays and want to get as much data as possible into the data cache. Single precision is good if you can take advantage of vector operations in the XMM registers, as explained on page 86.

Floating point addition takes 3 - 6 clock cycles, depending on the microprocessor. Multiplication takes 4 - 8 clock cycles. Division takes 32 - 45 clock cycles. Floating point comparisons are inefficient when the floating point stack registers are used. Conversions of float or double to integer takes a long time when the floating point stack registers are used.

Do not mix single and double precision when the XMM registers are used. See page 114.

Avoid conversions between integers and floating point variables, if possible. See page 115.

Mathematical applications that generate floating point underflow can benefit from setting denormal numbers to zero:

```
// Example 6.3
#include <xmmintrin.h>
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
```

See page 119 and 102 for more information about mathematical functions.

6.4 Enums

An `enum` is simply an integer in disguise. Enums are exactly as efficient as integers.

6.5 Booleans

The order of Boolean operands

The operands of the Boolean operators `&&` and `||` are evaluated in the following way. If the first operand of `&&` is false, then the second operand is not evaluated at all because the result is known to be false regardless of the value of the second operand. Likewise, if the first operand of `||` is true, then the second operand is not evaluated, because the result is known to be true anyway.

It may be advantageous to put the operand that is most often true last in an `&&` expression, or first in an `||` expression. Assume, for example, that `a` is true 50% of the time and `b` is true 10% of the time. The expression `a && b` needs to evaluate `b` when `a` is true, which is 50% of the cases. The equivalent expression `b && a` needs to evaluate `a` only when `b` is true, which is only 10% of the time. This is faster if `a` and `b` take the same time to evaluate and are equally likely to be predicted by the branch prediction mechanism. See page 33 for an explanation of branch prediction.

If one operand is more predictable than the other, then put the most predictable operand first.

If one operand is faster to calculate than the other then put the operand that is calculated the fastest first.

However, you must be careful when swapping the order of Boolean operands. You cannot swap the operands if the evaluation of the operands has side effects or if the first operand determines whether the second operand is valid. For example:

```
// Example 6.4
unsigned int i;  const int ARRAYSIZE = 100;  float list[ARRAYSIZE];
if (i < ARRAYSIZE && list[i] > 1.0) { ...
```

Here, you cannot swap the order of the operands because the expression `list[i]` is invalid when `i` is not less than `ARRAYSIZE`. Another example:

```
// Example 6.5
if (handle != INVALID_HANDLE_VALUE && WriteFile(handle, ...)) { ...
```

Here you cannot swap the order of the Boolean operands because you should not call `WriteFile` if the handle is invalid.

Boolean variables are overdetermined

Boolean variables are stored as 8-bit integers with the value 0 for false and 1 for true.

Boolean variables are overdetermined in the sense that all operators that have Boolean variables as input check if the inputs have any other value than 0 or 1, but operators that have Booleans as output can produce no other value than 0 or 1. This makes operations with Boolean variables as input less efficient than necessary. Take the example:

```
// Example 6.6a
bool a, b, c, d;
c = a && b;
d = a || b;
```

This is typically implemented by the compiler in the following way:

```
bool a, b, c, d;
if (a != 0) {
   if (b != 0) {
      c = 1;
   }
   else {
      goto CFALSE;
   }
}
else {
   CFALSE:
   c = 0;
}
if (a == 0) {
   if (b == 0) {
      d = 0;
   }
   else {
      goto DTRUE;
   }
}
```

```
else {
   DTRUE:
   d = 1;
}
```

This is of course far from optimal. The branches may take a long time in case of mispredictions (see page 33). The Boolean operations can be made much more efficient if it is known with certainty that the operands have no other values than 0 and 1. The reason why the compiler doesn't make such an assumption is that the variables might have other values if they are uninitialized or come from unknown sources. The above code can be optimized if `a` and `b` have been initialized to valid values or if they come from operators that produce Boolean output. The optimized code looks like this:

```
// Example 6.6b
char a = 0, b = 0, c, d;
c = a & b;
d = a | b;
```

Here, I have used `char` (or `int`) instead of `bool` in order to make it possible to use the bitwise operators (`&` and `|`) instead of the Boolean operators (`&&` and `||`). The bitwise operators are single instructions that take only one clock cycle. The OR operator (`|`) works even if `a` and `b` have other values than 0 or 1. The AND operator (`&`) and the EXCLUSIVE OR operator (`^`) may give inconsistent results if the operands have other values than 0 and 1.

Note that there are a few pitfalls here. You cannot use `~` for NOT. Instead, you can make a Boolean NOT on a variable which is known to be 0 or 1 by XOR'ing it with 1:

```
// Example 6.7a
bool a, b;
b = !a;
```

can be optimized to:

```
// Example 6.7b
char a = 0, b;
b = a ^ 1;
```

You cannot replace `a && b` with `a & b` if `b` is an expression that should not be evaluated if `a` is false. Likewise, you cannot replace `a || b` with `a | b` if `b` is an expression that should not be evaluated if `a` is true.

The trick of using bitwise operators is more advantageous if the operands are variables than if the operands are comparisons, etc. For example:

```
// Example 6.8
bool a; float x, y, z;
a = x > y && z != 0;
```

This is optimal in most cases. Don't change `&&` to `&` unless you expect the `&&` expression to generate many branch mispredictions.

Boolean vector operations

An integer may be used as a Boolean vector. For example, if `a` and `b` are 32-bit integers, then the expression `y = a & b;` will make 32 AND-operations in just one clock cycle. The operators `&`, `|`, `^`, `~` are useful for Boolean vector operations.

6.6 Pointers and references

Pointers versus references

Pointers and references are equally efficient because they are in fact doing the same thing. Example:

```
// Example 6.9
void FuncA (int * p) {
   *p = *p + 2;
}

void FuncB (int & r) {
   r = r + 2;
}
```

These two functions are doing the same thing and if you look at the code generated by the compiler you will notice that the code is exactly identical for the two functions. The difference is simply a matter of programming style. The advantages of using pointers rather than references are:

- When you look at the function bodies above, it is clear that `p` is a pointer, but it is not clear whether `r` is a reference or a simple variable. Using pointers makes it more clear to the reader what is happening.

- It is possible to do things with pointers that are impossible with references. You can change what a pointer points to and you can do arithmetic operations with pointers.

The advantages of using references rather than pointers are:

- The syntax is simpler when using references.

- References are safe to use because they always point to a valid address, while pointers can be invalid and cause fatal errors if they are uninitialized, if pointer arithmetic calculations go outside the bounds of valid addresses, or if pointers are type-casted to a wrong type.

- References are useful for copy constructors and overloaded operators.

- Function parameters that are declared as constant references accept expressions as arguments while pointers and non-constant references require a variable.

Efficiency

Accessing a variable or object through a pointer or reference may be just as fast as accessing it directly. The reason for this efficiency lies in the way microprocessors are constructed. All non-static variables and objects declared inside a function are stored on the stack and are in fact addressed relative to the stack pointer. Likewise, all non-static variables and objects declared in a class are accessed through the implicit pointer known in C++ as '`this`'. We can therefore conclude that most variables in a well-structured C++ program are in fact accessed through pointers in one way or another. Therefore, micro-processors have to be designed so as to make pointers efficient, and that's what they are.

However, there are disadvantages of using pointers and references. Most importantly, it requires an extra register to hold the value of the pointer or reference. Registers is a scarce resource, especially in 32-bit mode. If there are not enough registers then the pointer has to be loaded from memory each time it is used and this will make the program slower. Another disadvantage is that the value of the pointer is needed a few clock cycles before the time the variable pointed to can be accessed.

Pointer arithmetic

A pointer is in fact an integer that holds a memory address. Pointer arithmetic operations are therefore as fast as integer arithmetic operations. When an integer is added to a pointer then its value is multiplied by the size of the object pointed to. For example:

```
// Example 6.10
struct abc {int a; int b; int c;};
abc * p; int i;
p = p + i;
```

Here, the value that is added to `p` is not `i` but `i*12`, because the size of `abc` is 12 bytes. The time it takes to add `i` to `p` is therefore equal to the time it takes to make a multiplication and an addition. If the size of `abc` is a power of 2 then the multiplication can be replaced by a shift operation which is much faster. In the above example, the size of `abc` can be increased to 16 bytes by adding one more integer to the structure.

Incrementing or decrementing a pointer does not require a multiplication but only an addition. Comparing two pointers requires only an integer comparison, which is fast. Calculating the difference between two pointers requires a division, which is slow unless the size of the type of object pointed to is a power of 2 (See page 112 about division).

The object pointed to can be accessed approximately two clock cycles after the value of the pointer has been calculated. Therefore, it is recommended to calculate the value of a pointer well before the pointer is used. For example, `x = *(p++)` is more efficient than `x = *(++p)` because in the latter case the reading of `x` must wait until a few clock cycles after the pointer `p` has been incremented, while in the former case `x` can be read before `p` is incremented. See page 22 for more discussion of the increment and decrement operators.

6.7 Function pointers

Calling a function through a function pointer typically takes a few clock cycles more than calling the function directly if the target address can be predicted. The target address is predicted if the value of the function pointer is the same as last time the statement was executed. If the value of the function pointer has changed then the target address is likely to be mispredicted, which causes a long delay. See page 33 about branch prediction. A Pentium M processor may be able to predict the target if the changes of the function pointer follows a simple regular pattern, while Pentium 4 and AMD processors are sure to make a misprediction every time the function pointer has changed.

6.8 Member pointers

In simple cases, a data member pointer simply stores the offset of a data member relative to the beginning of the object, and a member function pointer is simply the address of the member function. But there are special cases such as multiple inheritance where a much more complicated implementation is needed. These complicated cases should definitely be avoided.

A compiler has to use the most complicated implementation of member pointers if it has incomplete information about the class that the member pointer refers to. For example:

```
// Example 6.11
class c1;
int c1::*MemberPointer;
```

Here, the compiler has no information about the class `c1` other than its name at the time `MemberPointer` is declared. Therefore, it has to assume the worst possible case and make a complicated implementation of the member pointer. This can be avoided by making

the full declaration of `c1` before `MemberPointer` is declared. Avoid multiple inheritance, virtual functions, and other complications that make member pointers less efficient.

Most C++ compilers have various options to control the way member pointers are implemented. Use the option that gives the simplest possible implementation if possible, and make sure you are using the same compiler option for all modules that use the same member pointer.

6.9 Arrays

An array is implemented simply by storing the elements consecutively in memory. No information about the dimensions of the array is stored. This makes the use of arrays in C and C++ faster than in other programming languages, but also less safe. An efficient way of overcoming this safety problem is explained on page 121.

An array initialized by a list should preferably be static, as explained on page 19. An array can be initialized to zero by using `memset`:

```
// Example 6.12
float list[100];
memset(list, 0, sizeof(list));
```

A multidimensional array should be organized so that the last index changes fastest:

```
// Example 6.13
const int rows = 20, columns = 50;
float matrix[rows][columns];
int i, j;  float x;
   for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
         matrix[i][j] += x;
```

This makes sure that the elements are accessed sequentially. The opposite order of the two loops would make the access non-sequential which makes the data caching less efficient.

The size of all but the first dimension should preferably be a power of 2 if the rows are indexed in a non-sequential order in order to make the address calculation more efficient:

```
// Example 6.14
int FuncRow(int);  int FuncCol(int);
const int rows = 20, columns = 32;
float matrix[rows][columns];
int i;  float x;
for (i = 0; i < 100; i++)
   matrix[FuncRow(i)][FuncCol(i)] += x;
```

Here, the code must compute `(FuncRow(i)*columns + FuncCol(i)) * sizeof(float)` in order to find the address of the matrix element. The multiplication by `columns` in this case is faster when columns is a power of two. In the preceding example, this is not an issue because an optimizing compiler can see that the rows are accessed consecutively and can calculate the address of each row by adding the length of a row to the address of the preceding row.

The same advice applies to arrays of structure or class objects. The size (in bytes) of the objects should preferably be a power of 2 if the elements are accessed in a non-sequential order.

The advice of making the number of columns a power of 2 does not always apply to arrays that are bigger than the level-1 data cache and accessed non-sequentially because it may cause cache contentions. See page 72 for a discussion of this problem.

6.10 Type conversions

The C++ syntax has several different ways of doing type conversions:

```
// Example 6.15
int i;  float f;
f = i;                        // Implicit type conversion
f = (float)i;                 // C-style type casting
f = float(i);                 // Constructor-style type casting
f = static_cast<float>(i);    // C++ casting operator
```

These different methods have exactly the same effect. Which method you use is a matter of programming style. The time consumption of different type conversions is discussed below.

Signed / unsigned conversion

```
// Example 6.16
int i;
if ((unsigned int)i < 10) { ...
```

Conversions between signed and unsigned integers simply makes the compiler interpret the bits of the integer in a different way. There is no checking for overflow, and the code takes no extra time. These conversions can be used freely without any cost in performance.

Integer size conversion

```
// Example 6.17
int i;  short int s;
i = s;
```

An integer is converted to a longer size by extending the sign-bit if the integer is signed, or by extending with zero-bits if unsigned. This typically takes one clock cycle if the source is an arithmetic expression. The size conversion often takes no extra time if it is done in connection with reading the value from a variable in memory, as in example 6.18.

```
// Example 6.18
short int a[100];  int i, sum = 0;
for (i=0; i<100; i++) sum += a[i];
```

Converting an integer to a smaller size is done simply by ignoring the higher bits. There is no check for overflow. Example:

```
// Example 6.19
int i;  short int s;
s = (short int)i;
```

This conversion takes no extra time. It simply stores the lower 16 bits of the 32-bit integer.

Floating point precision conversion

Conversions between `float`, `double` and `long double` take no extra time when the floating point register stack is used. It takes between 2 and 15 clock cycles (depending on the processor) when the XMM registers are used. See page 23 for an explanation of register stack versus XMM registers. Example:

```
// Example 6.20
```

```
float a; double b;
a += b;
```

In this example, the conversion is costly if XMM registers are used. `a` and `b` should be of the same type to avoid this. See page 114 for further discussion.

Integer to float conversion

Conversion of a signed integer to a `float` or `double` takes 4 - 16 clock cycles, depending on the processor and the type of registers used. Conversion of an unsigned integer takes longer time. It is faster to first convert the unsigned integer to a signed integer if there is no risk of overflow:

```
// Example 6.21
unsigned int u;  double d;
d = (double)(signed int)u;  // Faster, but risk of overflow
```

Integer to float conversions can sometimes be avoided by replacing an integer variable by a float variable. Example:

```
// Example 6.22a
float a[100];  int i;
for (i = 0;  i < 100;  i++) a[i] = 2 * i;
```

The conversion of `i` to float in this example can be avoided by making an additional floating point variable:

```
// Example 6.22b
float a[100];  int i;  float i2;
for (i = 0, i2 = 0;  i < 100;  i++, i2 += 2.0f) a[i] = i2;
```

Float to integer conversion

Conversion of a floating point number to an integer takes a very long time unless the SSE2 or later instruction set is enabled. Typically, the conversion takes 50 - 100 clock cycles. The reason is that the C/C++ standard specifies truncation so the floating point rounding mode has to be changed to truncation and back again.

If there are floating point-to-integer conversions in the critical part of a code then it is important to do something about it. Possible solutions are:

1. Avoid the conversions by using different types of variables.

2. Move the conversions out of the innermost loop by storing intermediate results as floating point.

3. Use 64-bit mode or enable the SSE2 instruction set (requires a microprocessor that supports this).

4. Use rounding instead of truncation and make a round function using assembly language. See page 115 for details about rounding.

Pointer type conversion

A pointer can be converted to a pointer of a different type. Likewise, a pointer can be converted to an integer, or an integer can be converted to a pointer. It is important that the integer has enough bits for holding the pointer.

These conversions do not produce any extra code. It is simply a matter of interpreting the same bits in a different way or bypassing syntax checks.

These conversions are not safe, of course. It is the responsibility of the programmer to make sure the result is valid.

Re-interpreting the type of an object

It is possible to make the compiler treat a variable or object as if it had a different type by type-casting its address:

```
// Example 6.23
float x;
*(int*)&x |= 0x80000000;   // Set sign bit of x
```

The syntax may seem a little odd here. The address of `x` is type-casted to a pointer to an integer, and this pointer is then de-referenced in order to access `x` as an integer. The compiler does not produce any extra code for actually making a pointer. The pointer is simply optimized away and the result is that `x` is treated as an integer. This kind of trick is unsafe if the object is treated as if it is bigger than it actually is. This code works only when `int` and `float` have the same size (32 bits).

The above example sets the sign bit of `x` by using the `|` operator which otherwise can only be applied to integers. It is faster than `x = -abs(x);`. The same effect can be obtained by making a union containing a `float` and an `int`.

Const cast

The `const_cast` operator is used for relieving the `const` restriction from a pointer. It has some syntax checking and is therefore more safe than the C-style type-casting without adding any extra code. Example:

```
// Example 6.24
class c1 {
   const int x;       // constant data
   public:
   c1() : x(0) {};    // constructor initializes x to 0
   void xplus2() {    // this function can modify x
      *const_cast<int*>(&x) += 2;}  // add 2 to x
};
```

The effect of the `const_cast` operator here is to remove the `const` restriction on `x`. It is a way of relieving a syntax restriction, but it doesn't generate any extra code and doesn't take any extra time. This is a useful way of making sure than one function can modify `x`, while other functions can not.

Static cast

The `static_cast` operator does the same as the C-style type-casting. It is used, for example, to convert `float` to `int`.

Reinterpret cast

The `reinterpret_cast` operator is used for pointer conversions. It does the same as C-style type-casting with a little more syntax check. It does not produce any extra code.

Dynamic cast

The `dynamic_cast` operator is used for converting a pointer to one class to a pointer to another class. It makes a runtime check that the conversion is valid. For example, when a pointer to a base class is converted to a pointer to a derived class, it checks whether the original pointer actually points to an object of the derived class. This check makes

`dynamic_cast` more time-consuming than a simple type casting, but also safer. It may catch programming errors that would otherwise go undetected.

Converting class objects

Conversions involving class objects (rather than pointers to objects) are possible only if the programmer has defined a constructor, an overloaded assignment operator, or an overloaded type casting operator that specifies how to do the conversion. The constructor or overloaded operator is as efficient as a member function.

6.11 Branches and switch statements

The high speed of modern microprocessors is obtained by using a pipeline where instructions are fetched and decoded in several stages before they are executed. However, the pipeline structure has one big problem. Whenever the code has a branch (e.g. an if-else structure), the microprocessor doesn't know in advance which of the two branches to feed into the pipeline. If the wrong branch is fed into the pipeline then the error is not detected until 10 - 20 clock cycles later and the work it has done by fetching, decoding and perhaps speculatively executing instructions during this time has been wasted. The consequence is that the microprocessor wastes several clock cycles whenever it feeds a branch into the pipeline and later discovers that it has chosen the wrong branch.

Microprocessor designers have gone to great lengths to reduce this problem. The most important method that is used is branch prediction. Modern microprocessors are using advanced algorithms to predict which way a branch will go based on the past history of that branch and other nearby branches. The algorithms used for branch prediction are different for each type of microprocessor. These algorithms are described in detail in manual 3: "The microarchitecture of Intel and AMD CPU's".

A branch instruction takes typically 0 - 2 clock cycles in the case that the microprocessor has made the right prediction. The time it takes to recover from a branch misprediction is approximately 12 clock cycles for AMD and Pentium M microprocessors, and more than 25 clock cycles for Pentium 4. This is called the branch misprediction penalty.

Branches are relatively cheap if they are predicted most of the time, but expensive if they are often mispredicted. A branch that always goes the same way is predicted well, of course. A branch that goes one way most of the time and rarely the other way is mispredicted only when it goes the other way. A branch that goes many times one way, then many times the other way is mispredicted only when it changes. A branch that follows a simple periodic pattern can also be predicted quite well if it is inside a loop with few or no other branches. A simple periodic pattern can be, for example, to go one way two times and the other way three times. Then again two times the first way and three times the other way, etc. The worst case is a branch that goes randomly one way or the other with a 50-50 chance of going either way. Such a branch will be mispredicted 50% of the time.

A for-loop or while-loop is also a kind of branch. After each iteration it decides whether to repeat or to exit the loop. The loop-branch is usually predicted well if the repeat count is small and always the same. The maximum loop count that can be predicted perfectly is 64 for Pentium M, 17 for Pentium 4, and 9 for AMD Athlon and Opteron. Nested loops are predicted well only on Pentium M. A loop that contains several branches is not predicted well on Pentium 4. A loop that contains several branches is predicted well on AMD processors only if the branches inside the loop always go the same way.

A switch statements is a kind of branch that can go more than two ways. On AMD and Pentium 4 processors, a switch statement is simply predicted to go the same way as last time it was executed. It is therefore certain to be mispredicted whenever it goes another way than last time. The Pentium M processor is sometimes able to predict a switch statement if it follows a simple periodic pattern or if it is correlated with preceding branches.

In some cases it is possible to replace a poorly predictable branch by a table lookup. For example:

```
// Example 6.25a
float a;  bool b;
a = b ? 1.5f : 2.6f;
```

The `?:` operator here is a branch. If it is poorly predictable then replace it by a table lookup:

```
// Example 6.25b
float a;  bool b = 0;
static const float lookup[2] = {2.6f, 1.5f};
a = lookup[b];
```

If a `bool` is used as an array index then it is important to make sure it is initialized or comes from a reliable source so that it can have no other values than 0 or 1. See page 25.

The number of branches should preferably be kept small in the critical part of a program, especially if the branches are poorly predictable. It may be useful to roll out a loop if this can eliminate branches, as explained in the next paragraph.

The target of branches and function calls are saved in a special cache called the branch target buffer. Contentions in the branch target buffer can occur if a program has many branches or function calls. The consequence of such contentions is that branches can be mispredicted even if they otherwise would be predicted well. Even function calls can be mispredicted for this reason. The branch target buffer is particularly small in the Pentium M processor. A program with many branches and function calls in the critical part of the code can therefore suffer from mispredictions, especially on a Pentium M processor.

Manual 3: "The microarchitecture of Intel and AMD CPU's" gives more details on branch predictions in the different microprocessors.

6.12 Loops

The efficiency of a loop depends on how well the microprocessor can predict the loop control branch. See the preceding paragraph and manual 3: "The microarchitecture of Intel and AMD CPU's" for an explanation of branch prediction. A loop with a small and fixed repeat count and no branches inside can be predicted perfectly. As explained above, the maximum loop count that can be predicted is 64 for Pentium M, 17 for Pentium 4, and 9 for AMD Athlon and Opteron. Nested loops are predicted well only on Pentium M. A loop with a high repeat count is mispredicted only when it exits. For example, if a loop repeats a thousand times then the loop control branch is mispredicted only one time in thousand so the misprediction penalty is only a negligible contribution to the total execution time.

Loop unrolling

In some cases it can be an advantage to unroll a loop. Example:

```
// Example 6.26a
int i;
for (i = 0; i < 20; i++) {
   if (i % 2 == 0) {
      FuncA(i);
   }
   else {
      FuncB(i);
   }
   FuncC(i);
}
```

This loop repeats 20 times and calls alternately `FuncA` and `FuncB`, then `FuncC`. Unrolling the loop by two gives:

```
// Example 6.26b
int i;
for (i = 0; i < 20; i += 2) {
   FuncA(i);
   FuncC(i);
   FuncB(i+1);
   FuncC(i+1);
}
```

This has three advantages:

1. The `i<20` loop control branch is executed 10 times rather than 20.

2. The fact that the repeat count has been reduced from 20 to 10 means that it can be predicted perfectly on a Pentium 4.

3. The `if` branch is eliminated.

Loop unrolling also has disadvantages:

1. The unrolled loop may take up more space in the code cache.

2. The Core2 processor performs better on very small loops (less than 65 bytes of code).

3. If the repeat count is odd and you unroll by two then there is an extra iteration that has to be done outside the loop. In general, you have this problem when the repeat count is not certain to be divisible by the unroll factor.

Loop unrolling should only be used if there are specific advantages that can be obtained. If a loop contains floating point calculations and the loop counter is an integer, then you can generally assume that the overall computation time is determined by the floating point code rather than by the loop control branch. There is nothing to gain by unrolling the loop in this case.

Compilers will usually unroll a loop automatically if this appears to be profitable (see page 56). The programmer does not have to unroll a loop manually unless there is a specific advantage to obtain, such as eliminating the `if`-branch in example 6.26b.

The loop control condition

The most efficient loop control condition is a simple integer counter. A microprocessor with out-of-order capabilities (see page 84) will be able to evaluate the loop control statement several iterations ahead.

It is less efficient if the loop control branch depends on the calculations inside the loop. The following example converts a zero-terminated ASCII string to lower case:

```
// Example 6.27a
char string[100], *p = string;
while (*p != 0) *(p++) |= 0x20;
```

If the length of the string is already known then it is more efficient to use a loop counter:

```
// Example 6.27b
char string[100], *p = string;  int i, StringLength;
```

```
for (i = StringLength; i > 0; i--) *(p++) |= 0x20;
```

A common situation where the loop control branch depends on calculations inside the loop is in mathematical iterations such as Taylor expansions and Newton-Raphson iterations. Here the iteration is repeated until the residual error is lower than a certain tolerance. The time it takes to calculate the absolute value of the residual error and compare it to the tolerance may be so high that it is more efficient to determine the worst-case maximum repeat count and always use this number of iterations. The advantage of this method is that the microprocessor can execute the loop control branch ahead of time and resolve any branch misprediction long before the floating point calculations inside the loop are finished. This method is advantageous if the typical repeat count is near the maximum repeat count and the calculation of the residual error for each iteration is a significant contribution to the total calculation time.

A loop counter should preferably be an integer. If a loop needs a floating point counter then make an additional integer counter. Example:

```
// Example 6.28a
double x, n, factorial = 1.0;
for (x = 2.0; x <= n; x++) factorial *= x;
```

This can be improved by adding an integer counter and using the integer in the loop control condition:

```
// Example 6.28b
double x, n, factorial = 1.0;  int i;
for (i = (int)n - 2, x = 2.0; i >= 0; i--, x++) factorial *= x;
```

Note the difference between commas and semicolons in a loop with multiple counters, as in example 6.28b. A `for`-loop has three clauses: initialization, condition, and increment. The three clauses are separated by semicolons, while multiple statements within each clause are separated by commas. There should be only one statement in the condition clause.

Comparing an integer to zero is sometimes more efficient than comparing it to any other number. Therefore, it is slightly more efficient to make a loop count down to zero than making it count up to some positive value, `n`. But not if the loop counter is used as an array index. The data cache is optimized for accessing arrays forwards, not backwards.

Copying or clearing arrays

It may not be optimal to use a loop for trivial tasks such as copying an array or setting an array to all zeroes. Example:

```
// Example 6.29a
const int size = 1000;  int i;
float a[size], b[size];
// set a to zero
for (i = 0; i < size; i++) a[i] = 0.0;
// copy a to b
for (i = 0; i < size; i++) b[i] = a[i];
```

It is often faster to use the functions `memset` and `memcpy`:

```
// Example 6.29b
const int size = 1000;
float a[size], b[size];
// set a to zero
memset(a, 0, sizeof(a));
// copy a to b
memcpy(b, a, sizeof(b));
```

Most compilers will automatically replace such loops by calls to `memset` and `memcpy`, at least in simple cases. The explicit use of `memset` and `memcpy` is unsafe because serious errors can happen if the size parameter is bigger than the destination array. But the same errors can happen with the loops if the loop count is too big.

6.13 Functions

Function calls may slow down a program for the following reasons:

- The function call makes the microprocessor jump to a different code address and back again. This may take up to 4 clock cycles. In most cases the microprocessor is able to overlap the call and return operations with other calculations to save time.

- The code cache works less efficiently if the code is fragmented and scattered around in memory.

- Function parameters are stored on the stack in 32-bit mode. Storing the parameters on the stack and reading them again takes extra time. The delay is considerable if a parameter is part of a critical dependence chain, especially on the Pentium 4 processor.

- Extra time is needed for setting up a stack frame, saving and restoring registers, and possibly save exception handling information.

- Each function call statement occupies a space in the branch target buffer (BTB). Contentions in the BTB can cause branch mispredictions if the critical part of a program has many calls and branches.

The following methods may be used for reducing the time spent on function calls in the critical part of a program.

Avoid unnecessary functions

Some programming textbooks recommend that every function that is longer than a few lines should be split up into multiple functions. I disagree with this rule. Splitting up a function into multiple smaller functions only makes the program less efficient. Splitting up a function just because it is long does not make the program more clear unless the function is doing multiple logically distinct tasks. The most critical innermost loop in a program should preferably be kept entirely inside one function, if possible.

Transfer large objects by reference

If an object (class or structure) is transferred to a function as a parameter, then the entire object is copied. The copy constructor is called if there is one, and the destructor is called when the function returns. If copying the object is not necessary for the logic of the algorithm, then it is more efficient to transfer a pointer or reference to the object rather than a copy of the object. It is preferred to use a `const` reference as parameter because this gives the compiler the best opportunities for optimization. An alternative solution is to make the function a member or the object's class so that the parameter is not needed.

Arrays are always transferred as pointers unless they are wrapped into a class or structure.

Functions that return a large object are inefficient. The return type should preferably be a simple type such as integer, pointer, reference or floating point. An alternative to returning a class object from a function is to make the function a member of the object's class or a constructor.

Use inline functions

An inline function is expanded like a macro so that each statement that calls the function is replaced by the function body. A function is inlined if the `inline` keyword is used or if its body is defined inside a class definition. Inlining a function is advantageous if the function is small or if it is called only from one place in the program. Small functions are often inlined automatically by the compiler. On the other hand, the compiler may in some cases ignore a request for inlining a function if the inlining causes technical problems.

Avoid nested function calls in the innermost loop

A function that calls other functions is called a *frame function*, while a function that doesn't call any other function is called a *leaf function*. Leaf functions are more efficient than frame functions for reasons explained on page 49. If the critical innermost loop of a program contains calls to frame functions then the code can probably be improved by inlining the frame function or by turning the frame function into a leaf function by inlining all the functions that it calls.

Use macros instead of functions

A macro declared with `#define` is certain to be inlined. But beware that macro parameters are evaluated every time they are used. Example:

```
// Example 6.30
#define max(a,b) (a > b ? a : b)
y = max(sin(x), cos(x));
```

In this example, `sin(x)` and `cos(x)` are both calculated twice because the macro is referencing them twice. This is of course not optimal.

Use fastcall functions

The keyword `__fastcall` changes the function calling method in 32-bit mode so that the first two or three (depending on compiler) integer parameters are transferred in registers rather than on the stack. Floating point parameters are not affected by `__fastcall`. The implicit '`this`' pointer in member functions is also treated like a parameter, so there may be only one free register left for transferring additional parameters. Therefore, make sure that the most critical integer parameter comes first when you are using `__fastcall`. The fastcall method is the default calling convention in 64-bit mode. Therefore, the `__fastcall` keyword is not recognized in 64-bit mode.

Make functions static

It is recommended to add the keyword `static` to all non-member functions that are not needed outside the module in which they are defined. The `static` declaration restricts the scope of the function to the current module (i.e. the current `.cpp` file). This enables the compiler to optimize across function calls. You cannot use this optimization for class member functions, because the keyword `static` has a different meaning for member functions.

Use whole program optimization

Some compilers have an option for whole program optimization or for combining multiple `.cpp` files into a single object file. This enables the compiler to optimize register allocation and parameter transfer across all `.cpp` modules that make up a program. Whole program optimization cannot be used for function libraries distributed as object or library files.

Use 64-bit mode

Parameter transfer is more efficient in 64-bit mode than in 32-bit mode, and more efficient in 64-bit Linux than in 64-bit Windows. In 64-bit Linux, the first six integer parameters and the first eight floating point parameters are transferred in registers, totaling up to fourteen

register parameters. In 64-bit Windows, the first four parameters are transferred in registers, regardless of whether they are integers or floating point numbers. Therefore, 64-bit Linux is more efficient than 64-bit Windows if functions have more than four parameters. There is no difference between 32-bit Linux and 32-bit Windows in this respect.

6.14 Structures and classes

Nowadays, programming textbooks recommend object oriented programming as a means of making software more clear and modular. The so-called objects are instances of structures and classes. The object oriented programming style has both positive and negative impacts on program performance. The positive effects are:

- Variables that are used together are also stored together if they are members of the same structure or class. This makes data caching more efficient.

- Variables that are members of a class need not be passed as parameters to a class member function. The overhead of parameter transfer is avoided for these variables.

The negative effects of object oriented programming are:

- Non-static member functions have a '`this`' pointer which is transferred as an implicit parameter to the function. The overhead of parameter transfer for '`this`' is incurred on all non-static member functions.

- The '`this`' pointer takes up one register. Registers is a scarce resource in 32-bit systems.

- Virtual member functions are less efficient (see page 41).

No general statement can be made about whether the positive or the negative effects of object oriented programming are dominating. At least, it can be said that the use of classes and member functions is not expensive. You may use an object oriented programming style if it is good for the logical structure and clarity of the program as long as you avoid an excessive number of function calls in the most critical part of the program. The use of structures (without member functions) has no negative effect on performance.

6.15 Class data members (properties)

The data members of a class or structure are stored consecutively in the order in which they are declared whenever an instance of the class or structure is created. There is no performance penalty for organizing data into classes or structures. Accessing a data member of a class or structure object takes no more time than accessing a simple variable.

Most compilers will align data members to round addresses in order to optimize access, as given in the following table.

Type	size, bytes	alignment, bytes
bool	1	1
char, signed or unsigned	1	1
short int, signed or unsigned	2	2
int, signed or unsigned	4	4
__int64, signed or unsigned	8	8
pointer or reference, 32-bit mode	4	4
pointer or reference, 64-bit mode	8	8
float	4	4
double	8	8

long double	8, 10, 12 or 16	8 or 16

Table 6.2. Alignment of data members.

This alignment can cause holes of unused bytes in a structure or class with members of mixed sizes. For example:

```
// Example 6.31a
struct S1 {
   short int a;  // 2 bytes. first byte at  0, last byte at  1
                 // 6 unused bytes
   double b;     // 8 bytes. first byte at  8, last byte at 15
   int d;        // 4 bytes. first byte at 16, last byte at 19
                 // 4 unused bytes
};
S1 ArrayOfStructures[100];
```

Here, there are 6 unused bytes between a and b because b has to start at an address divisible by 8. There are also 4 unused bytes in the end. The reason for this is that the next instance of S1 in the array must begin at an address divisible by 8 in order to align its b member by 8. The number of unused bytes can be reduced to 2 by putting the smallest members last:

```
// Example 6.31b
struct S1 {
   double b;     // 8 bytes. first byte at  0, last byte at  7
   int d;        // 4 bytes. first byte at  8, last byte at 11
   short int a;  // 2 bytes. first byte at 12, last byte at 13
                 // 2 unused bytes
};
S1 ArrayOfStructures[100];
```

This reordering has made the structure 8 bytes smaller and the array 800 bytes smaller.

Structure and class objects can often be made smaller by reordering the data members. If the class has at least one virtual member functions then there is a pointer to a virtual table before the first data member. This pointer is 4 bytes in 32-bit systems and 8 bytes in 64-bit systems. If you are in doubt how big a structure or each of its members are then you may make some tests with the `sizeof` operator.

The code for accessing a data member is more compact if the offset of the member relative to the beginning of the structure or class is less than 128. Example:

```
// Example 6.32
class S2 {
   public:
   int a[100];  // 400 bytes. first byte at   0, last byte at 399
   int b;       // 4 bytes.   first byte at 400, last byte at 403
   int ReadB() {return b;}
};
```

The offset of b is 400 here. Any code that accesses b through a pointer or a member function such as ReadB needs to add a 4-byte offset to the pointer. If a and b are swapped then both can be accessed with a 1-byte signed integer as offset. This makes the code more compact so that the code cache is used more efficiently. It is therefore recommended that big arrays and other big objects come last in a structure or class declaration. If it is not possible to contain all data members within the first 128 bytes then put the most used members in the first 128 bytes.

6.16 Class member functions (methods)

Each time a new object of a class is declared or created it will generate a new instance of the data members. But each member function has only one instance. The function code is not copied because the same code can be applied to all instances of the class.

Calling a member function is as fast as calling a simple function with a pointer or reference to a structure. For example:

```
// Example 6.33
class S3 {
   public:
   int a;
   int b;
   int Sum1() {return a + b;}
};
int Sum2(S3 * p) {return p->a + p->b;}
int Sum3(S3 & r) {return  r.a +  r.b;}
```

The three functions `Sum1`, `Sum2` and `Sum3` are doing exactly the same thing and they are equally efficient. If you look at the code generated by the compiler, you will notice that some compilers will make exactly identical code for the three functions. `Sum1` has an implicit '`this`' pointer which does the same thing as `p` and `r` in `Sum2` and `Sum3`. Whether you want to make the function a member of the class or give it a pointer or reference to the class or structure is simply a matter of programming style. Some compilers make `Sum1` slightly more efficient than `Sum2` and `Sum3` in 32-bit Windows by transferring '`this`' in a register rather than on the stack.

A `static` member function cannot access any non-static data members or non-static member functions. A static member function is faster than a non-static member function because it doesn't need the '`this`' pointer. You may make member functions faster by making them static if they don't need any non-static access.

6.17 Virtual member functions

Virtual functions are used for implementing polymorphic classes. Each instance of a polymorphic class has a pointer to a table of pointers to the different versions of the virtual functions. This so-called virtual table is used for finding the right version of the virtual function at runtime. Polymorphism is one of the main reasons why object oriented programs can be less efficient than non-object oriented programs. If you can avoid virtual functions then you can obtain most of the advantages of object oriented programming without paying the performance costs.

The time it takes to call a virtual member function is a few clock cycles more than it takes to call a non-virtual member function, provided that the function call statement always calls the same version of the virtual function. If the version changes then you will get a misprediction penalty of 10 - 30 clock cycles. The rules for prediction and misprediction of virtual function calls is the same as for switch statements, as explained on page 33.

Runtime polymorphism is needed only if it cannot be known at compile time which version of a polymorphic member function is called. If virtual functions are used in a critical part of a program then you may consider whether it is possible to obtain the desired functionality without polymorphism or with compile-time polymorphism. Example 6.38 page 45 shows how it is possible to replace runtime polymorphism with the more efficient compile-time polymorphism by the use of templates.

6.18 Runtime type identification (RTTI)

Runtime type identification adds extra information to all class objects and is not efficient. If the compiler has an option for RTTI then turn it off and use alternative implementations.

6.19 Inheritance

An object of a derived class is implemented in the same way as an object of a simple class containing the members of both parent and child class. Members of parent and child class are accessed equally fast. In general, you can assume that there is hardly any performance penalty to using inheritance.

There may be a slight degradation in code caching for the following reasons:

1. The size of the parent class data members is added to the offset of the child class members. The code that accesses data members with a total offset bigger than 127 bytes is slightly less compact. See page 40.

2. The member functions of parent and child are typically stored in different modules. This may cause a lot of jumping around and less efficient code caching. This problem can be solved by making sure that functions which are called near each other are also stored near each other. See page 73 for details.

Inheritance from multiple parent classes in the same generation can cause complications with member pointers and virtual functions.

6.20 Constructors and destructors

A constructor is implemented as a member function which returns a reference to the object. The allocation of memory for a new object is rarely done by the constructor itself. Constructors are therefore as efficient as any other member functions. This applies to default constructors, copy constructors, and any other constructors.

A class doesn't need a constructor. A default constructor is not needed if the object doesn't need initialization. A copy constructor is not needed if the object can be copied simply by copying all data members. A simple constructor may be inlined for improved performance.

A destructor is as efficient as a member function. Do not make a destructor if it is not necessary. A virtual destructor is as efficient as a virtual member function. See page 41.

6.21 Unions

A union is a structure where data members share the same memory space. A union can be used for saving memory space by allowing two data members that are never used at the same time to share the same piece of memory. See page 74 for an example.

A union can also be used for accessing the same data in different ways. Example:

```
// Example 6.34
union {
   float f;
   int i;
} x;
x.f = 2.0f;
x.i |= 0x80000000;  // set sign bit of f
cout << x.f;        // will give -2.0
```

In this example, the sign bit of `f` is set by using the bitwise OR operator, which can only be applied to integers.

6.22 Bitfields

Bitfields may be useful for making data more compact. Accessing a member of a bitfield is less efficient than accessing a member of a structure. The extra time may be justified in case of large arrays if it can save cache space or make files smaller.

It is faster to compose a bitfield by the use of `<<` and `|` operations than to write the members individually. Example:

```
// Example 6.35a
struct Bitfield {
   int a:4;
   int b:2;
   int c:2;
};
Bitfield x;
int A, B, C;
x.a = A;
x.b = B;
x.c = C;
```

Assuming that the values of `A`, `B` and `C` are too small to cause overflow, this code can be improved in the following way:

```
// Example 6.35b
union Bitfield {
   struct {
      int a:4;
      int b:2;
      int c:2;
   };
   char abc;
};
Bitfield x;
int A, B, C;
x.abc = A | (B << 4) | (C << 6);
```

Or, if protection against overflow is needed:

```
// Example 6.35c
x.abc = (A & 0x0F) | ((B & 3) << 4) | ((C & 3) <<6 );
```

6.23 Overloaded functions

The different versions of an overloaded function are simply treated as different functions. There is no performance penalty for using overloaded functions.

6.24 Overloaded operators

An overloaded operator is equivalent to a function. Using an overloaded operator is exactly as efficient as using a function that does the same thing.

An expression with multiple overloaded operators will cause the creation of temporary objects for intermediate results, which may be undesired. Example:

```
// Example 6.36a
```

```
class vector {                                  // 2-dimensional vector
public:
   float x, y;                                  // x,y coordinates
   vector() {}                                  // default constructor
   vector(float a, float b) {x = a; y = b;}     // constructor
   vector operator + (vector const & a) {       // sum operator
      return vector(x + a.x, y + a.y);}         // add elements
};

vector a, b, c, d;
a = b + c + d;          // makes intermediate object for (b + c)
```

The creation of a temporary object for the intermediate result `(b+c)` can be avoided by joining the operations:

```
// Example 6.36b
a.x = b.x + c.x + d.x;
a.y = b.y + c.y + d.y;
```

Fortunately, most compilers will do this optimization automatically in simple cases.

6.25 Templates

A template is similar to a macro in the sense that the template parameters are replaced by their values before compilation. The following example illustrates the difference between a function parameter and a template parameter:

```
// Example 6.37
int Multiply (int x, int m) {
   return x * m;}

template <int m>
int MultiplyBy (int x) {
   return x * m;}

int a, b;
a = Multiply(10,8);
b = MultiplyBy<8>(10);
```

`a` and `b` will both get the value 10 * 8 = 80. The difference lies in the way `m` is transferred to the function. In the simple function, `m` is transferred at runtime from the caller to the called function. But in the template function, `m` is replaced by its value at compile time so that the compiler sees the constant 8 rather than the variable `m`. The advantage of using a template parameter rather than a function parameter is that the overhead of parameter transfer is avoided. The disadvantage is that the compiler needs to make a new instance of the template function for each different value of the template parameter. If `MultiplyBy` in this example is called with many different factors as template parameters then the code can become very big.

In the above example, the template function is faster than the simple function because the compiler knows that it can multiply by a power of 2 by using a shift operation. `x*8` is replaced by `x<<3`, which is faster. In the case of the simple function, the compiler doesn't know the value of `m` and therefore cannot do the optimization. (In the above example, the compiler is actually able to inline and optimize both functions and simply put 80 into `a` and `b`. But in more complex cases it might not be able to do so).

A template parameter can also be a type. The example on page 121 shows how you can make arrays of different types with the same template.

A template class can be used for implementing a compile-time polymorphism, which is more efficient than the runtime polymorphism that is obtained with virtual member functions. The following example shows first the runtime polymorphism:

```
// Example 6.38a
void DoX();
void DoY();

class CHello {
public:
   virtual void Dispatch() {};
   void Hello() {
      Dispatch();}
   void GoodBye();
};

class C1 : public CHello {
   public:
   virtual void Dispatch() {
      DoX();}
};

class C2 : public CHello {
   public:
   virtual void Dispatch() {
      DoY();}
};

void test () {
   C1 Object1;  C2 Object2;
   Object1.Hello();         // Will call DoX()
   Object2.Hello();         // Will call DoY()
}
```

Here, the call of `Object1.Hello()` will be dispatched at runtime to `C1::Dispatch` which then calls `DoX`, while the call of `Object2.Hello()` will be dispatched at runtime to `C2::Dispatch` which then calls `DoY`. It is explained on page 41 why this method is inefficient. If it is known at compile-time whether the object belongs to class `C1` or `C2`, then we can avoid the costly virtual function dispatch process. This can be obtained by using a template class:

```
// Example 6.38b
class C1 {
   public:
   void Dispatch() {
      DoX();}
};

class C2 {
   public:
   void Dispatch() {
      DoY();}
};

template <class MyParent>
class CHello : public MyParent {
public:
   void Hello() {
      Dispatch();}
   void GoodBye();
};

void test () {
```

```
   CHello<C1> Object1;  CHello<C2> Object2;
   Object1.Hello();         // Will call DoX()
   Object2.Hello();         // Will call DoY()
}
```

Here, it is known at compile-time that `Object1` belongs to class `CHello<C1>`, while `Object2` belongs to class `CHello<C2>`. The compiler can therefore replace the calls to `Object1.Hello` and `Object2.Hello` by direct calls to `DoX` and `DoY`, respectively. This is much more efficient than using virtual member functions because the dispatching is done at compile time rather than at runtime.

A disadvantage here is that the member function `GoodBye` has two instances, `CHello<C1>::GoodBye` and `CHello<C2>::GoodBye`. This is a waste of code cache space because the two instances of `GoodBye` are identical. This can be avoided by placing any non-polymorphic member functions in a grandparent class:

```
// Example 6.38c
class CGrandParent {
public:
   void GoodBye();
};

class C1 : public CGrandParent {
   public:
   void Dispatch() {
      DoX();}
};

class C2 : public CGrandParent {
   public:
   void Dispatch() {
      DoY();}
};

template <class MyParent>
class CHello : public MyParent {
public:
   void Hello() {
      Dispatch();}
};

void test () {
   CHello<C1> Object1;  CHello<C2> Object2;
   Object1.Hello();
   Object2.Hello();
}
```

The order of inheritance may be a little confusing here. The non-polymorphic member functions (`GoodBye`) are in a first generation class. The different versions of the polymorphic member functions (`Dispatch`) are in each their second generation class. Any function that calls the polymorphic member functions (`Hello`) is in a third generation template class.

We may want the order of inheritance to be changed in order to bring it in accordance with common object oriented programming practice. This requires a special trick which is used in the Active Template Library (ATL) and Windows Template Library (WTL):

```
// Example 6.38d
class CGrandParent {
public:
   void GoodBye();
```

```
};

template <class MyChild>
class CParent : public CGrandParent {
public:
   void Hello() {
      // call polymorphic child function:
      (static_cast<MyChild*>(this))->Dispatch();}
};

class CChild1 : public CParent<CChild1> {
   public:
   void Dispatch() {
      DoX();}
};

class CChild2 : public CParent<CChild2> {
   public:
   void Dispatch() {
      DoY();}
};

void test () {
   CChild1 Object1;  CChild2 Object2;
   Object1.Hello();         // Will call DoX()
   Object2.Hello();         // Will call DoY()
}
```

Here `CParent` is a template class which gets information about its child class through a template parameter. It can call the polymorphic member of its child class by type-casting its '`this`' pointer to a pointer to its child class. This is only safe if it has the correct child class name as template parameter. In other words, you must make sure that the declaration

```
class CChild1 : public CParent<CChild1> {
```

has the same name for the child class name and the template parameter.

The order of inheritance is now as follows. The first generation class (`CGrandParent`) contains any non-polymorphic member functions. The second generation class (`CParent<>`) contains any member functions that need to call a polymorphic function. The third generations classes contain the different versions of the polymorphic functions. The second generation class gets information about the third generation class through a template parameter.

Changing the order of inheritance here does not affect the efficiency. The calls to `Object1.Hello()` and `Object2.Hello()` are still replaced at compile time by `DoX` and `DoY`. No time is wasted on runtime dispatch to virtual member functions. This method can be recommended as a general way to replace runtime polymorphism by compile-time polymorphism.

6.26 Threads

Threads are used for doing two or more jobs simultaneously or seemingly simultaneously. If the computer has only one CPU kernel then it is not possible to do two jobs simultaneously. Each thread will get time slices of typically 30 ms for foreground jobs and 10 ms for background jobs. The context switches after each time slice are quite costly because all caches have to adapt to the new context. It is possible to reduce the number of context switches by making longer time slices. This will make applications run faster at the cost of longer response times for user input. (In Windows you can increase the time slices to 120

ms by selecting optimize performance for background services under advanced system performance options. I don't know if this is possible in Linux).

Threads are useful for assigning different priorities to different tasks. For example, in a word processor the user expects an immediate response to pressing a key or moving the mouse. This task must have a high priority. Other tasks such as spell-checking and repagination are running in other threads with lower priority. If the different tasks were not divided into threads with different priorities then the user might experience unacceptably long response times to keyboard and mouse inputs when the program is busy doing the spell checking.

Any task that takes a long time, such as heavy mathematical calculations, should be scheduled in a separate thread if the application has a graphical user interface. Otherwise the program will be unable to respond quickly to keyboard or mouse input.

It is possible to make a thread-like scheduling in an application program without invoking the overhead of the operating system thread scheduler. This can be accomplished by doing the heavy background calculations piece by piece in a function that is called from the message loop of a graphical user interface (`OnIdle` in Windows MFC). This method may be faster than making a separate thread in systems with only one CPU kernel, but it requires that the background job can be divided into small pieces of a suitable duration.

The best way to fully utilize systems with multiple CPU kernels is to divide the job into multiple threads. Each thread can then run on its own CPU kernel.

There are four kinds of costs to multithreading that we have to take into account when optimizing multithreaded applications:

1. The cost of starting and stopping threads. Don't put a task into a separate thread if it is short in duration compared with the time it takes to start and stop the thread.

2. The cost of task switching. This cost is minimized if the number of threads with the same priority is no more than the number of CPU kernels.

3. The cost of synchronizing and communicating between threads. The overhead of semaphores, mutexes, etc. is considerable. If two threads are often waiting for each other in order to get access to the same resource then it may be better to join them into one thread. A variable that is shared between multiple threads must be declared `volatile`. This prevents the compiler from doing optimizations on that variable.

4. The different threads need separate storage. No function or class that is used by multiple threads should rely on static or global variables. The threads have each their stack. This can cause cache contentions if the threads share the same cache.

6.27 Exception handling

Exception handling is intended for detecting errors that seldom occur and recovering from an error condition in a graceful way. You may think that exception handling takes no extra time as long as the error doesn't occur, but unfortunately this is not true. The program has to do a lot of bookkeeping in order to know how to recover in the rare event of an exception. The following example explains this:

```
// Example 6.39
class C1 {
   public:
   ...
   ~C1();
};
```

```
void F1() {
   C1 x;
   ...
}

void F0() {
   try {
      F1();
   }
   catch (...) {
   ...
   }
}
```

The function `F1` is supposed to call the destructor for the object `x` when it returns. But what if an exception occurs somewhere in `F1`? Then we are breaking out of `F1` without returning. `F1` is prevented from cleaning up because it has been brutally interrupted. Now it is the responsibility of the exception handler to call the destructor of `x`. This is only possible if `F1` has saved all information about the destructor to call or any other cleanup that may be necessary. If `F1` calls another function which in turn calls another function, etc., and if an exception occurs in the innermost function, then the exception handler needs all information about the chain of function calls and it needs to follow the track backwards though the function calls to check for all the necessary cleanup jobs to do. This is called stack unwinding.

All functions have to save some information for the exception handler, even if no exception ever happens. This is the reason why exception handling is expensive. If exception handling is not necessary for your application then you should disable it in order to make the code smaller and more efficient. You can disable exception handling for the whole program by turning off the exception handling option in the compiler. You can disable exception handling for a single function by adding `throw()` to the function prototype:

```
void F1() throw();
```

This tells the compiler not to save recovery information for function `F1`. It is recommended to add `throw()` to functions that are critical to program performance.

The compiler makes a distinction between *leaf functions* and *frame functions*. A frame function is a function that calls at least one other function. A leaf function is a function that doesn't call any other function. A leaf function is simpler than a frame function because the stack unwinding information can be left out if exceptions can be ruled out or if there is nothing to clean up in case of an exception. A frame function can be turned into a leaf function by inlining all the functions that it calls. The best performance is obtained if the critical innermost loop of a program contains no calls to frame functions.

In some cases, it is optimal to use exception handling even in the most critical part of a program. This is the case if alternative implementations are less efficient and you want to be able to recover from errors. The following example illustrates such a case:

```
// Example 6.40
// Portability note: This example is specific to Microsoft compilers.
#include <excpt.h>
#include <float.h>
#include <math.h>
#define EXCEPTION_FLT_OVERFLOW  0xC0000091L

void MathLoop() {
   const int arraysize = 1000;  unsigned int dummy;
   double  a[arraysize], b[arraysize], c[arraysize];
```

```
        // Enable exception for floating point overflow:
        _controlfp_s(&dummy, 0, _EM_OVERFLOW);
        // _controlfp(0, _EM_OVERFLOW); // if above line doesn't work

        int i = 0;   // Initialize loop counter outside both loops
        // The purpose of the while loop is to resume after exceptions:
        while (i < arraysize) {
           // Catch exceptions in this block:
           __try {
              // Main loop for calculations:
              for ( ; i < arraysize; i++) {

                 // Overflow may occur in multiplication here:
                 a[i] = log (b[i] * c[i]);
              }
           }
           // Catch floating point overflow but no other exceptions:
           __except (GetExceptionCode() == EXCEPTION_FLT_OVERFLOW
           ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
              // Floating point overflow has occurred.
              // Reset floating point status:
              _fpreset();
              _controlfp_s(&dummy, 0, _EM_OVERFLOW);
              // _controlfp(0, _EM_OVERFLOW); // if above doesn't work

              // Re-do the calculation in a way that avoids overflow:
              a[i] = log(b[i]) + log(c[i]);

              // Increment loop counter and go back into the for-loop:
              i++;
           }
        }
     }
```

Assume that the numbers in `b[i]` and `c[i]` are so big that overflow can occur in the multiplication `b[i]*c[i]`, though this only happens rarely. The above code will catch an exception in case of overflow and redo the calculation in a way that takes more time but avoids the overflow. Taking the logarithm of each factor rather than the product makes sure than no overflow can occur, but the calculation time is doubled.

The time it takes to make support for the exception handling is negligible because there is no `try` block or function call (other than `log`) inside the critical innermost loop. `log` is a library function which we assume is optimized. We cannot change its possible exception handling support anyway. The exception is costly when it occurs, but this is not a problem since we are assuming that the occurrence is rare.

Testing for the overflow condition inside the loop does not cost anything here because we are relying on the microprocessor hardware for raising an exception in case of overflow. The exception is caught by the operating system which redirects it to the exception handler in the program if there is a `try` block.

Let's look at the possible alternatives to exception handling in this example. We might check for overflow by checking if `b[i]` and `c[i]` are too big before multiplying them. This would require two floating point comparisons, which are relatively costly because they must be inside the innermost loop. Another possibility is to always use the safe formula `a[i] = log(b[i]) + log(c[i]);`. This would double the number of calls to `log`, and logarithms take a long time to calculate. If there is a way to check for overflow outside the loop without checking all the array elements then this might be a better solution. It might be possible to do such a check before the loop if all the factors are generated from the same few parameters. Or it might be possible to do the check after the loop if the results are combined by some formula into a single result. An uncaught overflow condition will generate

the value infinity, and this value will propagate through the calculations so that the final result will be infinity or NAN (Not A Number) if an overflow or other error has occurred anywhere in the calculations. The program can check the final result to see if it is a valid number (e.g. with `_finite()`) and redo the calculations in a safe way in case of error. The calculations may take more time than normal on some microprocessors when an operand is infinity or NAN.

Exception handling is not necessary when no attempt is made to recover from errors. If you just want the program to issue an error message and stop the program in case of an error then there is no reason to use `try`, `catch`, and `throw`. It is more efficient to define your own error-handling function that simply prints an appropriate error message and then calls `exit`.

There are other possible ways of handling errors without using exceptions. The function that detects an error can return with an error code which the caller can use for recovering or for issuing an error message.

There is a portability issue to catching hardware exceptions. The mechanism relies on non-standardized details in both compiler, operating system and CPU hardware. Porting such an application to a different platform is likely to require modifications in the code.

6.28 Other cases of stack unwinding

The preceding paragraph described a mechanism called stack unwinding that is used by exception handlers for cleaning up after jumping out of a function without properly returning in case of an exception. This mechanism is also used in two other situations:

The stack unwinding mechanism is used when a thread is terminated. The purpose is to detect if any objects declared in the thread have a destructor that needs to be called. It is recommended to return from functions that require cleanup before terminating a thread. A function needs cleanup if it has declared or created an object that has a destructor.

The stack unwinding mechanism is also used when the function `longjmp` is used for jumping out of a function. Avoid the use of `longjmp` if possible. Don't ever rely on `longjmp` in time-critical code.

6.29 Preprocessing directives

Preprocessing directives (everything that begins with `#`) are costless in terms of program performance because they are resolved before the program is compiled.

`#if` directives are useful for supporting multiple platforms or multiple configurations with the same source code. `#if` is more efficient than `if` because `#if` is resolved at compile time while `if` is resolved at runtime.

`#define` directives are equivalent to `const` definitions when used for defining constants. For example, `#define ABC 123` and `const int ABC = 123;` are equally efficient.

`#define` directives when used as macros are often more efficient than functions. See page 38 for a discussion.

7 Optimizations in the compiler

7.1 How compilers optimize

Modern compilers can do a lot of modifications to the code in order to improve performance. It is useful for the programmer to know what the compiler can do and what it can not do. The following sections describe some of the compiler optimizations that it is relevant for the programmer to know about.

Function inlining

The compiler can replace a function call by the body of the called function. Example:

```
// Example 7.1a
float square (float a) {
   return a * a;}

float parabola (float x) {
   return square(x) + 1.0f;}
```

The compiler may replace the call to square by the code inside square:

```
// Example 7.1b
float parabola (float x) {
   return x * x + 1.0f;}
```

The advantages of function inlining are:

- The overhead of call and return and parameter transfer are eliminated.
- Code caching will be better because the code becomes contiguous.
- The code becomes smaller if there is only one call to the inlined function.
- Function inlining can open the possibility for other optimizations, as explained below.

The disadvantage of function inlining is that the code becomes bigger if there is more than one call to the inlined function and the function is big. The compiler is more likely to inline a function if it is small or if it is called from only one or a few places.

Constant folding and constant propagation

An expression or subexpression containing only constants will be replaced by the calculated result. Example:

```
// Example 7.2a
double a, b;
a = b + 2.0 / 3.0;
```

The compiler will replace this by

```
// Example 7.2b
a = b + 0.66666666666666666667;
```

This is actually quite convenient. It is easier to write `2.0/3.0` than to calculate the value and write it with many decimals. It is recommended to put a parenthesis around such a subexpression to make sure the compiler recognizes it as a subexpression. For example, `b*2.0/3.0` will most likely be calculated as `(b*2.0)/3.0` rather than as `b*(2.0/3.0)` unless you put a parenthesis around the constant subexpression.

A constant can be propagated through a series of calculations:

```
// Example 7.3a
float parabola (float x) {
   return x * x + 1.0f;}
```

```
float a, b;
a = parabola (2.0f);
b = a + 1.0f;
```

The compiler may replace this by

```
// Example 7.3b
a = 5.0f;
b = 6.0f;
```

Constant folding and constant propagation is not possible if the expression contains a function which cannot be inlined or cannot be calculated at compile time. For example:

```
// Example 7.4
double a = sin(0.8);
```

The `sin` function is defined in a separate function library and you cannot expect the compiler to be able to inline this function and calculate it at compile time. Some compilers are able to calculate the most common math functions such as `sqrt` and `pow` at compile-time, but not the more complicated functions like `sin`.

Pointer elimination

A pointer or reference can be eliminated if the target pointed to is known. Example:

```
// Example 7.5a
void Plus2 (int * p) {
   *p = *p + 2;}

int a;
Plus2 (&a);
```

The compiler may replace this by

```
// Example 7.5b
a += 2;
```

Common subexpression elimination

If the same subexpression occurs more than once then the compiler may calculate it only once. Example:

```
// Example 7.6a
int a, b, c;
b = (a+1) * (a+1);
c = (a+1) / 4;
```

The compiler may replace this by

```
// Example 7.6b
int a, b, c, temp;
temp = a+1;
b = temp * temp;
c = temp / 4;
```

Register variables

The most commonly used variables are stored in registers (see page 20).

The maximum number of integer register variables is approximately six in 32-bit systems and fourteen in 64-bit systems.

The maximum number of floating point register variables is eight in 32-bit systems and sixteen in 64-bit systems. Some compilers have difficulties making floating point register variables in 32-bit systems unless the SSE2 (or later) instruction set is enabled.

The compiler will choose the variables that are used most for register variables. This includes pointers and references, which can be stored in integer registers. Typical candidates for register variables are temporary intermediates, loop counters, function parameters, pointers, references, '`this`' pointer, common subexpressions, and induction variables (see below).

A variable cannot be stored in a register if its address is taken, i.e. if there is a pointer or reference to it. Therefore, you should avoid making any pointer or reference to a variable that could benefit from register storage.

Live range analysis

The live range of a variable is the range of code in which the variable is used. An optimizing compiler can use the same register for more than one variable if their live-ranges do not overlap or if they are sure to have the same value. This is useful when the number of available registers is limited. Example:

```
// Example 7.7
int SomeFunction (int a, int x[]) {
   int b, c;
   x[0] = a;
   b = a + 1;
   x[1] = b;
   c = b + 1;
   return c;
}
```

In this example, `a`, `b` and `c` can share the same register because their live ranges do not overlap. If `c = b + 1` is changed to `c = a + 2` then `a` and `b` cannot use the same register because their live ranges now overlap.

Compilers do not normally use this principle for objects stored in memory. It will not use the same memory area for different objects even when their live ranges do not overlap. See page 74 for an example of how to make different objects share the same memory area.

Join identical branches

The code can be made more compact by joining identical pieces of code. Example:

```
// Example 7.8a
double x, y, z;  bool b;
if (b) {
   y = sin(x);
   z = y + 1.;
}
else {
   y = cos(x);
   z = y + 1.;
}
```

The compiler may replace this by

```
// Example 7.8b
double x, y;  bool b;
if (b) {
   y = sin(x);
}
```

```
else {
   y = cos(x);
}
z = y + 1.;
```

Eliminate jumps

Jumps can be avoided by copying the code that it jumps to. Example:

```
// Example 7.9a
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
   }
   else {
      a = a * 3;
   }
   return a + 1;
}
```

This code has a jump from `a=a*2;` to `return a+1;`. The compiler can eliminate this jump by copying the return statement:

```
// Example 7.9b
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
      return a + 1;
   }
   else {
      a = a * 3;
      return a + 1;
   }
}
```

A branch can be eliminated if the condition can be reduced to always true or always false:

```
// Example 7.10a
if (true) {
   a = b;
}
else {
   a = c;
}
```

Can be reduced to:

```
// Example 7.10b
a = b;
```

A branch can also be eliminated if the condition is known from a previous branch. Example:

```
// Example 7.11a
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
   }
   else {
      a = a * 3;
   }
   if (b) {
      return a + 1;
```

```
   }
   else {
      return a - 1;
   }
}
```

The compiler may reduce this to:

```
// Example 7.11b
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
      return a + 1;
   }
   else {
      a = a * 3;
      return a - 1;
   }
}
```

Loop unrolling

Some compilers will unroll loops if a high degree of optimization is requested. See page 34. This may be advantageous if the loop body is very small or if it opens the possibility for further optimizations. Loops with a very low repeat count may be completely unrolled to avoid the loop overhead. Example:

```
// Example 7.12a
int i, a[2];
for (i = 0; i < 2; i++) a[i] = i+1;
```

The compiler may reduce this to:

```
// Example 7.12b
int a[2];
a[0] = 1; a[1] = 2;
```

Excessive loop unrolling is not always optimal. In some cases it can be useful to turn off the loop unroll option in the compiler.

Loop invariant code motion

A calculation may be moved out of a loop if it is independent of the loop counter. Example:

```
// Example 7.13a
int i, a[100], b;
for (i = 0; i < 100; i++) {
   a[i] = b * b + 1;
}
```

The compiler may change this to:

```
// Example 7.13b
int i, a[100], b, temp;
temp = b * b + 1;
for (i = 0; i < 100; i++) {
   a[i] = temp;
}
```

Induction variables

An expression that is a linear function of a loop counter can be calculated by adding a constant to the previous value. Example:

```
// Example 7.14a
int i, a[100];
for (i = 0; i < 100; i++) {
   a[i] = i * 9 + 3;
}
```

The compiler may avoid the multiplication by changing this to:

```
// Example 7.14b
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++) {
   a[i] = temp;
   temp += 9;
}
```

Induction variables are often used for calculating the addresses of array elements. Example:

```
// Example 7.15a
struct S1 {double a; double b;};
S1 list[100];  int i;
for (i = 0; i < 100; i++) {
   list[i].a = 1.0;
   list[i].b = 2.0;
}
```

In order to access an element in `list`, the compiler must calculate its address. The address of `list[i]` is equal to the address of the beginning of `list` plus `i*sizeof(S1)`. This is a linear function of `i` which can be calculated by an induction variable. The compiler can use the same induction variable for accessing `list[i].a` and `list[i].b`. It can also eliminate `i` and use the induction variable as loop counter when the final value of the induction variable can be calculated in advance. This reduces the code to:

```
// Example 7.15b
struct S1 {double a; double b;};
S1 list[100], *temp;
for (temp = &list[0]; temp < &list[100]; temp++) {
   temp->a = 1.0;
   temp->b = 2.0;
}
```

The factor `sizeof(S1)` = 16 is actually hidden behind the C++ syntax in example 7.15b. The integer representation of `&list[100]` is `(int)(&list[100]) = (int)(&list[0]) + 100*16`, and `temp++` actually adds 16 to the integer value of `temp`.

The compiler doesn't need induction variables to calculate the addresses of array elements of simple types because the CPU has hardware support for calculating the address of an array element if the address can be expressed as a base address plus a constant plus an index multiplied by a factor of 1, 2, 4 or 8, but not any other factor. If `a` and `b` in example 7.15a were `float` instead of `double`, then `sizeof(S1)` would be 8 and no induction variable would be needed because the CPU has hardware support for multiplying the index by 8.

The compilers I have studied do not make induction variables for floating point expressions or more complex integer expressions. See page 65 for an example of how to use induction variables for calculating a polynomial.

Scheduling

A compiler may reorder instructions for the sake of parallel execution. Example:

```
// Example 7.16
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```

The compiler may interleave the two formulas in this example so that `a+b` is calculated first, then `d+e`, then `c` is added to the first sum, then `f` is added to the second sum, then the first result is stored in `x`, and last the second result is stored in `y`. The purpose of this is to help the CPU doing multiple calculations in parallel. Modern CPUs are actually able to reorder instructions without help of the compiler (see page 84), but the compiler can make this reordering easier for the CPU.

Algebraic reductions

Most compilers can reduce simple algebraic expressions using the fundamental laws of algebra. For example, a compiler may change the expression `-(-a)` to `a`.

I don't think that programmers write expressions like `-(-a)` very often, but such expressions may occur as a result of other optimizations such as function inlining. Reducible expressions also occur quite often as a result of macro expansions.

Programmers do, however, often write expressions that can be reduced. This may be because the non-reduced expression better explains the logic behind the program or because the programmer hasn't thought about the possibility of algebraic reduction. For example, a programmer may prefer to write `if(!a && !b)` rather than the equivalent `if(!(a || b))` even though the latter has one operator less. Fortunately, all compilers are able to do the reduction in this case.

You cannot expect a compiler to reduce complicated algebraic equations. For example, only one of the compilers I have tested were able to reduce `(a*b*c)+(c*b*a)` to `a*b*c*2`. It is quite difficult to implement the many rules of algebra in a compiler. Some compilers can reduce some types of equations and other compilers can reduce other types of equations, but no compiler I have ever seen can reduce them all. In the case of Boolean algebra, it is possible to implement a universal algorithm (e.g. Cline-McCluskey or Espresso) that can reduce any expression, but none of the compilers I have tested seem to do so.

The compilers are better at reducing integer expressions than floating point expressions, even though the rules of algebra are the same in both cases. This is because algebraic manipulations of floating point expressions may have undesired effects. This effect can be illustrated by the following example:

```
// Example 7.17
char a = -100, b = 100, c = 100, y;
y = a + b + c;
```

Here, `y` will get the value -100+100+100 = 100. Now, according to the rules of algebra, we may write:

```
y = c + b + a;
```

This may be useful if the subexpression `c+b` can be reused elsewhere. In this example, we are using 8-bit integers which range from -128 to +127. An integer overflow will make the value wrap around. Adding 1 to 127 will generate -128, and subtracting 1 from -128 generates 127. The calculation of `c+b` will generate an overflow and give the result -56 rather than 200. Next, we are adding -100 to -56 which will generate an underflow and give

the result 100 rather than -156. Surprisingly, we end up with the correct result because the overflow and underflow neutralize each other. This is the reason why it is safe to use algebraic manipulations on integer expressions (except for the <, <=, > and >= operators).

The same argument does not apply to floating point expressions. Floating point variables do not wrap around on overflow and underflow. The range of floating point variables is so large that we do not have to worry much about overflow and underflow except in special mathematical applications. But we do have to worry about loss of precision. Let's repeat the above example with floating point numbers:

```
// Example 7.18
float a = -1.0E8, b = 1.0E8, c = 1.23456, y;
y = a + b + c;
```

The calculation here gives `a+b=0`, and then `0+1.23456 = 1.23456`. But we will not get the same result if we change the order of the operands and add `b` and `c` first. `b+c = 100000001.23456`. The `float` type holds a precision of approximately seven significant digits, so the value of `b+c` will be rounded to `100000000`. When we add `a` to this number we get `0` rather than `1.23456`.

The conclusion to this argument is that the order of floating point operands cannot be changed without the risk of loosing precision. The compilers will not do so unless you specify an option that allows less precise floating point calculations. Even with all relevant optimization options turned on, the compilers will not do such obvious reductions as `0/a = 0` because this would be invalid if `a` was zero or infinity or NAN (not a number). Different compilers behave differently because there are different opinions on which imprecisions should be allowed an which not.

You cannot rely on the compiler to do any algebraic reductions on floating point code and you can rely on only the most simple reductions on integer code. It is more safe to do the reductions manually. I have tested the capability to reduce various algebraic expressions on seven different compilers. The results are listed in table 7.1 below.

7.2 Comparison of different compilers

I have made a series of experiments on seven different brands of C++ compilers to see whether they were able to do different kinds of optimizations. The results are summarized in table 7.1. The table shows whether the different compilers succeeded in applying the various optimization methods and algebraic reductions in my test examples.

The table can give some indication of which optimizations you can expect a particular compiler to do and which optimizations you have to do manually.

It must be emphasized that the compilers may behave differently on different test examples. You cannot expect a compiler to always behave according to the table.

Optimization method	Microsoft	Borland	Intel	Gnu	Digital Mars	Watcom	Codeplay
Function inlining	x	-	x	x	-	-	x
Constant folding	x	x	x	x	x	x	x
Constant propagation	x	-	x	x	-	-	x
Pointer elimination	x	-	x	x	x	x	x
Common subexpression, integer	x	x	x	x	x	x	x

Common subexpression, float	x	-	x	x	-	x	x
Register variables, integer	x	x	x	x	x	x	x
Register variables, float	x	-	x	x	-	x	x
Live range analysis	x	x	x	x	x	x	x
Join identical branches	x	-	-	x	-	x	-
Eliminate jumps	x	-	x	x	-	x	x
Eliminate branches	x	-	x	x	-	-	-
Eliminate branch that is always true/false	x	-	x	x	x	x	x
Loop unrolling	x	-	x	x	-	-	x
Loop invariant code motion	x	-	x	x	x	x	x
Induction variables for array elements	x	x	x	x	x	x	x
Induction variables for other integer expressions	x	-	x	x	x	x	x
Induction variables for float expressions	-	-	-	-	-	-	-
Automatic vectorization	-	-	x	-	-	-	x
Integer algebra reductions:							
a+b = b+a	x	x	x	x	-	x	x
a*b = b*a	x	x	x	x	-	x	x
(a+b)+c = a+(b+c)	x	x	x	x	x	x	-
a+b+c = c+b+a	x	-	-	x	-	-	-
a+b+c+d = (a+b)+(c+d)	-	-	x	x	-	-	-
a*b+a*c = a*(b+c)	x	-	x	x	-	-	x
a*x*x*x + b*x*x + c*x + d = ((a*x+b)*x+c)*x+d	x	-	-	x	-	-	x
$x*x*x*x*x*x*x*x = ((x^2)^2)^2$	-	-	x	-	-	-	-
a+a+a+a = a*4	x	-	x	x	-	-	x
-(-a) = a	x	-	x	x	x	x	-
a-(-b) = a+b	x	-	x	x	-	x	-
a-a = 0	x	-	x	x	x	x	x
a+0 = a	x	x	x	x	x	x	x
a*0 = 0	x	x	x	x	x	-	x
a*1 = a	x	x	x	x	x	x	x
(-a)*(-b) = a*b	x	-	x	x	-	-	-
a/a = 1	-	-	-	-	-	-	x
a/1 = a	x	x	x	x	x	x	x
0/a = 0	-	-	-	x	-	x	x
(-a == -b) = (a == b)	-	-	-	x	-	-	-
(a+c == b+c) = (a == b)	-	-	-	-	-	-	-
!(a < b) = (a >= b)	x	x	x	x	x	x	x
(a<b && b<c && a<c) = (a<b && b<c)	-	-	-	-	-	-	-
Multiply by constant = shift and add	x	x	x	x	x	x	-
Divide by constant = multiply and shift	x	-	x	x	x	-	-
Floating point algebra reductions:							
a+b = b+a	x	-	-	x	-	-	x
a*b = b*a	x	-	-	x	-	-	x
a+b+c = a+(b+c)	x	-	x	x	-	-	-
(a+b)+c = a+(b+c)	-	-	x	x	-	-	-
a*b*c = a*(b*c)	x	-	-	x	-	-	-
a+b+c+d = (a+b)+(c+d)	-	-	-	x	-	-	-
a*b+a*c = a*(b+c)	x	-	-	-	-	-	x
a*x*x*x + b*x*x + c*x + d = ((a*x+b)*x+c)*x+d	x	-	-	x	-	-	-
$x*x*x*x*x*x*x*x = ((x^2)^2)^2$	-	-	-	x	-	-	-
a+a+a+a = a*4	x	-	-	x	-	-	-
-(-a) = a	-	-	x	x	x	x	-
a-(-b) = a+b	-	-	-	x	-	x	-

a+0 = a	x	-	x	x	x	x	-
a*0 = 0	-	-	x	x	-	x	x
a*1 = a	x	-	x	x	x	-	x
(-a)*(-b) = a*b	-	-	-	x	-	-	-
a/a = 1	-	-	-	-	-	-	x
a/1 = a	x	-	x	x	x	-	-
0/a = 0	-	-	-	x	-	x	x
(-a == -b) = (a == b)	-	-	-	x	-	-	-
(-a > -b) = (a < b)	-	-	-	x	-	-	x
Divide by constant = multiply by reciprocal	x	x	x	x	-	x	-
Boolean algebra reductions:							
!(!a) = a	x	-	x	x	x	x	x
(a&&b) || (a&&c) = a&&(b||c)	x	-	x	x	-	-	-
!a && !b = !(a || b)	x	x	x	x	x	x	x
a && !a = false, a || !a = true	x	-	x	x	-	-	-
a && true = a, a || false = a	x	x	x	x	x	x	-
a && false = false, a || true = true	x	-	x	x	x	x	-
a && a = a	x	-	x	x	-	-	-
(a&&b) || (a&&!b) = a	x	-	-	x	-	-	-
(a&&b) || (!a&&c) = a ? b : c	x	-	x	x	-	-	-
(a&&b) || (!a&&c) || (b&&c) = a ? b : c	x	-	-	x	-	-	-
(a&&b) || (a&&b&&c) = a&&b	x	-	-	x	-	-	-
(a&&b) || (a&&c) || (a&&b&&c) = a&&(b||c)	x	-	-	x	-	-	-
(a&&!b) || (!a&&b) = a XOR b	-	-	-	-	-	-	-
Bit vector algebra reductions:							
~(~a) = a	x	-	x	x	x	-	-
(a&b)|(a&c) = a&(b|c)	x	-	x	x	-	-	x
(a|b)&(a|c) = a|(b&c)	x	-	x	x	-	-	x
~a & ~b = ~(a | b)	-	-	x	x	-	-	-
a & a = a	x	-	-	x	-	-	x
a & ~a = 0	-	-	x	x	-	-	-
a & -1 = a, a | 0 = a	x	-	x	x	x	x	x
a & 0 = 0, a | -1 = -1	x	-	x	x	x	x	x
(a&b) | (~a&c) | (b&c) = (a&b) | (~a&c)	-	-	-	-	-	-	-
a&b&c&d = (a&b)&(c&d)	-	-	-	x	-	-	-
a ^ 0 = a	x	x	x	x	x	x	x
a ^ -1 = ~a	x	-	x	x	x	x	-
a ^ a = 0	x	-	x	x	-	x	x
a ^ ~a = -1	-	-	-	x	-	-	-
(a&~b) | (~a&b) = a ^ b	-	-	-	-	-	-	-
~a ^ ~b = a ^ b	-	-	-	x	-	-	-
a<<b<<c = a<<(b+c)	x	-	x	x	-	x	x
Integer XMM (vector) reductions:							
a+b = b+a, a*b = b*a	-	n.a.	-	x	n.a.	n.a.	x
(a+b)+c = a+(b+c)	-	n.a.	-	-	n.a.	n.a.	-
a*b+a*c = a*(b+c)	-	n.a.	-	-	n.a.	n.a.	-
$x*x*x*x*x*x*x*x = ((x^2)^2)^2$	-	n.a.	-	-	n.a.	n.a.	-
a+a+a+a = a*4	-	n.a.	-	-	n.a.	n.a.	-
-(-a) = a	-	n.a.	-	-	n.a.	n.a.	-
a-a = 0	-	n.a.	x	-	n.a.	n.a.	-
a+0 = a	-	n.a.	-	-	n.a.	n.a.	-
a*0 = 0	-	n.a.	-	x	n.a.	n.a.	-

a*1 = a	-	n.a.	-	x	n.a.	n.a.	-
(-a)*(-b) = a*b	-	n.a.	-	-	n.a.	n.a.	-
!(a < b) = (a >= b)	-	n.a.	-	-	n.a.	n.a.	-
Floating point XMM (vector) reductions:							
a+b = b+a, a*b = b*a	x	n.a.	-	x	n.a.	n.a.	x
a+b+c = a+(b+c)	-	n.a.	-	-	n.a.	n.a.	-
a*b+a*c = a*(b+c)	-	n.a.	-	-	n.a.	n.a.	-
-(-a) = a	-	n.a.	-	-	n.a.	n.a.	-
a-a = 0	-	n.a.	-	x	n.a.	n.a.	-
a+0 = a	-	n.a.	x	-	n.a.	n.a.	-
a*0 = 0	-	n.a.	x	-	n.a.	n.a.	-
a*1 = a	-	n.a.	-	x	n.a.	n.a.	-
a/1 = a	-	n.a.	-	x	n.a.	n.a.	-
0/a = 0	-	n.a.	x	x	n.a.	n.a.	-
Divide by constant = multiply by reciprocal	-	n.a.	-	-	n.a.	n.a.	-
Boolean XMM (vector) reductions:							
~(~a) = a	-	n.a.	-	-	n.a.	n.a.	-
(a&b)|(a&c) = a&(b|c)	-	n.a.	-	-	n.a.	n.a.	-
a & a = a, a | a = a	-	n.a.	x	x	n.a.	n.a.	-
a & ~a = 0	-	n.a.	-	x	n.a.	n.a.	-
a & -1 = a, a | 0 = a	-	n.a.	-	-	n.a.	n.a.	-
a & 0 = 0	-	n.a.	-	x	n.a.	n.a.	-
a | -1 = -1	-	n.a.	-	-	n.a.	n.a.	-
a ^ a = 0	-	n.a.	x	x	n.a.	n.a.	-
andnot(a,a) = 0	-	n.a.	-	x	n.a.	n.a.	-
a<<b<<c = a<<(b+c)	-	n.a.	-	-	n.a.	n.a.	-

Table 7.1. Comparison of optimizations in different C++ compilers

The tests were carried out with all relevant optimization options turned on, including relaxed floating point precision. The following compiler versions were tested:
Microsoft C++ Compiler v. 14.00 for 80x86 / x64 (Visual Studio 2005).
Borland C++ Builder v. 10.0, 2005.
Intel C++ Compiler v. 9.1 Beta for 32-bit / EM64T, 2006.
Gnu C++ v. 4.1.0, 2006 (Red Hat).
Digital Mars Compiler v. 8.42n, 2004.
Open Watcom C/C++ v. 1.4, 2005.
Codeplay VectorC v. 2.1.7, 2004.
No differences were observed between the optimization capabilities for 32-bit and 64-bit code for the Microsoft, Intel and Gnu compilers.

7.3 Obstacles to optimization by compiler

There are several factors that can prevent the compiler from doing the optimizations that we want it to do. It is important for the programmer to be aware of these obstacles and to know how to avoid them. Some important obstacles to optimization are discussed below.

Pointer aliasing

When accessing a variable through a pointer or reference, the compiler may not be able to completely rule out the possibility that the variable pointed to is identical to some other variable in the code. Example:

```
// Example 7.19
void Func1 (int a[], int * p) {
   int i;
   for (i = 0; i < 100; i++) {
```

```
      a[i] = *p + 2;
   }
}

void Func2() {
   int list[100];
   Func1(list, &list[8]);
}
```

Here, it is necessary to reload `*p` and calculate `*p+2` a hundred times because the value pointed to by `p` is identical to one of the elements in `a[]` which will change during the loop. It is not permissible to assume that `*p+2` is a loop-invariant code that can be moved out of the loop. Example 7.19 is indeed a very contrived example, but the point is that the compiler cannot rule out the theoretical possibility that such contrived examples exist. Therefore the compiler is prevented from assuming that `*p+2` is a loop-invariant expression that it can move outside the loop.

Most compilers have an option for assuming no pointer aliasing (`/Oa`). The easiest way to overcome the obstacle of possible pointer aliasing is to turn on this option. This requires that you analyze all pointers and references in the code carefully to make sure that no variable or object is accessed in more than one way in the same part of the code. It is also possible to tell the compiler that a specific pointer does not alias anything by using the keyword `__restrict` or `__restrict__`, if supported by the compiler.

We can never be sure that the compiler takes the hint about no pointer aliasing. The only way to make sure that the code is optimized is to do it explicitly. In example 7.19, you could calculate `*p+2` and store it in a temporary variable outside the loop if you are sure that the pointer does not alias any elements in the array. This method requires that you can predict where the obstacles to optimization are.

Cannot optimize across modules

The compiler doesn't have information about functions in other modules than the one it is compiling. This prevents it from making optimizations across function calls. Example:

```
// Example 7.20
module1.cpp
int Func1(int x) {
   return x*x + 1;
}

module2.cpp
int Func2() {
   int a = Func1(2);
   ...
}
```

If `Func1` and `Func2` were in the same module then the compiler would be able do function inlining and constant propagation and reduce `a` to the constant 5. But the compiler doesn't have the necessary information about `Func1` when compiling `module2.cpp`.

The simplest way to solve this problem is to combine the multiple `.cpp` modules into one by means of `#include` directives. This is sure to work on all compilers. Some compilers have a feature called whole program optimization, which will enable optimizations across modules (See page 66).

Pure functions

A pure function is a function that has no side-effects and its return value depends only on the values of its arguments. This closely follows the mathematical notion of a "function".

Multiple calls to a pure function with the same arguments are sure to produce the same result. A compiler can eliminate common subexpressions that contain pure function calls and it can move out loop-invariant code containing pure function calls. Unfortunately, the compiler cannot know that a function is pure if the function is defined in a different module or a function library.

Therefore, it is necessary to do optimizations such as common subexpression elimination, constant propagation, and loop-invariant code motion manually when it involves pure function calls.

The Gnu compiler and the Intel compiler for Linux have an attribute which can be applied to a function prototype to tell the compiler that this is a pure function. Example:

```
// Example 7.21
#ifdef __GNUC__
#define pure_function  __attribute__((const))
#else
#define pure_function
#endif

double Func1(double) pure_function ;

double Func2(double x) {
   return Func1(x) * Func1(x) + 1.;
}
```

Here, the Gnu compiler will make only one call to `Func1`, while other compilers will make two.

Some other compilers (Microsoft, Intel) know that standard library functions like `sqrt`, `pow` and `log` are pure functions, but unfortunately there is no way to tell these compilers that a user-defined function is pure.

Algebraic reduction

Most compilers can do simple algebraic reductions such as -(-a) = a, but they are not able to do more complicated reductions. Algebraic reduction is a complicated process which is difficult to implement in a compiler.

Many algebraic reductions are not permissible for reasons of mathematical purity. In many cases it is possible to construct obscure examples where the reduction would cause overflow or loss of precision, especially in floating point expressions (see page 58). The compiler cannot rule out the possibility that a particular reduction would be invalid in a particular situation, but the programmer can. It is therefore necessary to do the algebraic reductions explicitly in many cases.

Integer expressions are less susceptible to problems of overflow and loss of precision for reasons explained on page 58. It is therefore possible for the compiler to do more reductions on integer expressions than on floating point expressions. Most reductions involving integer addition, subtraction and multiplication are permissible in all cases, while many reductions involving division and relational operators (e.g. '>') are not permissible for reasons of mathematical purity. For example, compilers cannot reduce the integer expression `-a > -b` to `a < b` because of a very obscure possibility of overflow.

Table 7.1 (page 62) shows which reductions the compilers are able to do, at least in some situations, and which reductions they cannot do. All the reductions that the compilers cannot do must be done manually by the programmer.

Floating point induction variables

Compilers cannot make floating point induction variables for the same reason that they cannot make algebraic reductions on floating point expressions. It is therefore necessary to do this manually. This principle is useful whenever a function of a loop counter can be calculated more efficiently from the previous value than from the loop counter. Any expression that is an n'th degree polynomial of the loop counter can be calculated by n additions and no multiplications. The following example shows the principle for a 2'nd order polynomial:

```
// Example 7.22a. Loop to make table of polynomial
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100];                      // Table
int x;                                  // Loop counter
for (x = 0; x < 100; x++) {
   Table[x] = A*x*x + B*x + C;          // Calculate polynomial
}
```

The calculation of this polynomial can be done with just two additions by the use of two induction variables:

```
// Example 7.22b. Calculate polynomial with induction variables
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100];                      // Table
int x;                                  // Loop counter
const double A2 = A + A;                // = 2*A
double Y = C;                           // = A*x*x + B*x + C
double Z = A + B;                       // = Delta Y
for (x = 0; x < 100; x++) {
   Table[x] = Y;                        // Store result
   Y += Z;                              // Update induction variable Y
   Z += A2;                             // Update induction variable Z
}
```

The loop in example 7.22b has two loop-carried dependence chains, namely the two induction variables `Y` and `Z`. Each dependence chain has a latency which is the same as the latency of a floating point addition. This is small enough to justify the method. A longer loop-carried dependence chain would make the induction variable method unfavorable, unless the value is calculated from a value that is two or more iterations back.

The method of induction variables can also be vectorized if you take into account that each value is calculated from the value that lies r places back in the sequence, where r is the number of elements in a vector or the loop unroll factor. A little math is required for finding the right formula in each case.

Inlined functions have a non-inlined copy

Function inlining has the complication that the same function may be called from another module. The compiler has to make a non-inlined copy of the inlined function for the sake of the possibility that the function is also called from another module. This non-inlined copy is dead code if no other modules call the function. This fragmentation of the code makes caching less efficient.

There are various ways around this problem. If a function is not referenced from any other module then add the keyword `static` to the function definition. This tells the compiler that the function cannot be called from any other module. The `static` declaration makes it easier for the compiler to evaluate whether it is optimal to inline the function, and it prevents the compiler from making an unused copy of an inlined function. The `static` keyword also makes various other optimizations possible because the compiler doesn't have to obey any specific calling conventions for functions that are not accessible from other modules. You may add the `static` keyword to all local non-member functions.

Unfortunately, this method doesn't work for class member functions because the `static` keyword has a different meaning for member functions. You can force a member function to be inlined by declaring the function body inside the class definition. This will prevent the compiler from making a non-inlined copy of the function, but it has the disadvantage that the function is always inlined even when it is not optimal to do so (i.e. if the member function is big and is called from many different places).

Some compilers have an option (Windows: `/Gy`, Linux: `-ffunction-sections`) which allows the linker to remove unreferenced functions. It is recommended to turn on this option.

7.4 Obstacles to optimization by CPU

Modern CPU's can do a lot of optimization by executing instructions out of order. Long dependence chains in the code prevent the CPU from doing out-of-order execution, as explained on page 15.

Avoid long dependence chains, especially loop-carried dependence chains with long latencies.

7.5 Compiler optimization options

All C++ compilers have various optimization options that you can turn on and off. It is important to study the available options for the compiler you are using and turn on all relevant options.

Many optimization options are incompatible with debugging. A debugger can execute a code one line at a time and show the values of all variables. Obviously, this is not possible when parts of the code have been reordered, inlined, or optimized away. It is common to make two versions of a program executable: a debug version with full debugging support which is used during program development, and a release version with all relevant optimization options turned on. Most IDE's (Integrated Development Environments) have facilities for making a debug version and a release version of object files and executables. Make sure to distinguish these two versions and turn off debugging and profiling support in the optimized version of the executable.

Most compilers offer the choice between optimizing for size and optimizing for speed. Optimizing for size is relevant when the code is fast anyway and you want the executable to be as small as possible or when code caching is critical. Optimizing for speed is relevant when CPU access and memory access are critical time consumers. Choose the strongest optimization option available.

Some compilers offer profile-guided optimization. This works in the following way. First you compile the program with profiling support. Then you make a test run with a profiler which determines the program flow and the number of times each function and branch is executed. The compiler can then use this information to optimize the code and put the different functions in the optimal order.

Some compilers have support for whole program optimization. This works by compiling in two steps. All source files are first compiled to an intermediate file format instead of the usual object file format. The intermediate files are then linked together in the second step where the compilation is finished. Register allocation and function inlining is done at the second step. The intermediate file format is not standardized. It is not even compatible with different versions of the same compiler. It is therefore not possible to distribute function libraries in this format.

Other compilers offer the possibility of compiling multiple `.cpp` files into a single object file. This enables the compiler to do cross-module optimizations when interprocedural optimization is enabled. A more primitive, but efficient, way of doing whole program optimization is to join all source files into one by means of `#include` directives and declare all functions static or inline. This will enable the compiler to do interprocedural optimizations of the whole program.

During the history of CPU development, each new generation of CPU's increased the available instruction set. The newer instruction sets enable the compiler to make more efficient code, but this makes the code incompatible with old CPU's. The Pentium Pro instruction set makes floating point comparisons more efficient. This instruction set is supported by all modern CPU's. The SSE2 instruction set is particularly interesting because it makes floating point code more efficient in some cases and it makes it possible to use vector instructions (see page 86). Using the SSE2 instruction set is not always optimal, though. In some cases the SSE2 instruction set makes floating point code slower, especially when the code mixes float and double (see page 114). The SSE2 instruction set is not supported by all 32-bit CPU's and operating systems available today (2006).

You may choose a newer instruction set when compatibility with old CPU's is not needed. Even better, you may make multiple versions of the most critical part of the code to support different CPU's. This method is explained on page 105.

The code becomes more efficient when there is no exception handling. It is recommended to turn off support for exception handling unless the code relies on structured exception handling and you want the code to be able to recover from exceptions. See page 48.

It is recommended to turn off support for runtime type identification (RTTI). See page 42.

It is recommended to enable fast floating point calculations or turn off requirements for strict floating point calculations unless the strictness is required. See page 59 and 58 for discussions.

Turn on the option for "function level linking" if available. See page 66 for an explanation of this option.

Use the option for "assume no pointer aliasing" if you are sure the code has no pointer aliasing. See page 62 for an explanation. (The Microsoft compiler supports this option only in the Professional and Enterprise editions).

Do not turn on correction for the "FDIV bug". The FDIV bug is a minor error in some old Pentium CPU's which may cause slight imprecision in some rare cases of floating point division. Correction for the FDIV bug causes floating point division to be slower.

Many compilers have an option for "standard stack frame" or "frame pointer". The standard stack frame is used for debugging and exception handling. Omitting the standard stack frame makes function calls faster and makes an extra register available for other purposes. This is advantageous because registers is a scarce resource. Do not use a stack frame unless your program relies on exception handling.

7.6 Optimization directives

Some compilers have many keywords and directives which are used for giving specific optimization instructions at specific places in the code. Many of these directives are compiler-specific. You cannot expect a directive for a Windows compiler to work on a Linux compiler, or vice versa. But most of the Microsoft directives work on the Intel compiler for Windows and the Gnu compiler for Windows, while most of the Gnu directives work on the Intel compiler for Linux.

Keywords that work on all C++ compilers

The `register` keyword can be added to a variable declaration to tell the compiler that you want this to be a register variable. The register keyword is only a hint and the compiler may not take the hint, but it can be useful in situations where the compiler is unable to predict which variables will be used most.

The opposite of `register` is `volatile`. The `volatile` keyword makes sure that a variable is never stored in a register, not even temporarily. This is intended for variables that are shared between multiple threads, but it can also be used for turning off all optimizations of a variable for test purposes.

The `const` keyword tells that a variable is never changed. This allows the compiler to replace all references to this variable by its value. A `const` pointer or `const` reference cannot change what it points to. A `const` member function cannot modify data members. It is recommended to use the `const` keyword wherever appropriate to give the compiler additional information about a variable, pointer or member function because this may improve the possibilities for optimization.

The `static` keyword has several meanings depending on the context. The keyword `static`, when applied to a non-member function, means that the function is not accessed by any other modules. This makes inlining more efficient and enables interprocedural optimizations. See page 65.

The keyword `static`, when applied to a global variable means that it is not accessed by any other modules. This enables interprocedural optimizations.

The keyword `static`, when applied to a local `const` variable means that it is initialized only the first time the function is called. Example:

```
// Example 7.23
void Func () {
   static const double log2 = log(2.);
   ...
}
```

Here, `log(2.)` is only calculated the first time `Func` is executed. Without `static`, the logarithm would be re-calculated every time `Func` is executed. This has the disadvantage that the function must check if it has been called before. This is faster than calculating the logarithm again, but it would be even faster to make `log2` a global `const` variable or replace it with the calculated value.

The keyword `static`, when applied to a class member function means that it cannot access any non-static data members or member functions. A static member function is called faster than a non-static member function because it doesn't need a '`this`' pointer. Make member functions static where appropriate.

Compiler-specific keywords

Fast function calling. `__fastcall` or `__attribute__((fastcall))`. The fastcall modifier can make function calls faster in 32-bit mode. The first two integer parameters are transferred in registers rather than on the stack (three parameters on Borland compiler). Fastcall functions are not compatible across compilers. Fastcall is not needed in 64-bit mode where the parameters are transferred in registers anyway.

Pure function. `__attribute__((const))` (Linux only). Specifies a function to be pure. This allows common subexpression elimination and loop-invariant code motion. See page 63.

Assume no pointer aliasing. `__declspec(noalias)` or `__restrict` or `#pragma optimize("a",on)`. Specifies that pointer aliasing does not occur. See page 62 for an explanation. Note that these directives do not always work.

Data alignment. `__declspec(align(16))` or `__attribute__((aligned(16)))`. Specifies alignment of arrays and structures. Useful for vector operations, see page 86.

7.7 Checking what the compiler does

It can be very useful to study the code that a compiler generates to see how well it optimizes the code. Sometimes the compiler does quite ingenious things to make the code more efficient, and sometimes it does incredibly stupid things. Looking at the compiler output can often reveal things that can be improved by modifications of the source code, as the example below shows.

The best way to check the code that the compiler generates is to use a compiler option for assembly language output. On most compilers you can do this by invoking the compiler from the command line with all the relevant optimization options and the options `-S` or `/Fa` for assembly output. The assembly output option is also available from the IDE on some systems. If the compiler doesn't have an assembly output option then use an object file disassembler.

Note that the Intel compiler has an option for source annotation in the assembly output (`/FAs` or `-fsource-asm`). This option makes the assembly output more readable but unfortunately it prevents certain optimizations. Do not use the source annotation option if you want to see the result of full optimization.

It is also possible to see the compiler-generated code in the disassembly window of a debugger. However, the code that you see in the debugger is not the optimized version because the debugging options prevent optimization. The debugger cannot set a breakpoint in the fully optimized code because it doesn't have the line number information. It is often possible to insert a fixed breakpoint in the code with an inline assembly instruction for interrupt 3. The code is `__asm int 3;` or `__asm ("int 3");` or `__debugbreak();`. If you run the optimized code (release version) in the debugger then it will break at the interrupt 3 breakpoint and show a disassembly, probably without information about function names and variable names. Remember to remove the interrupt 3 breakpoint again.

The following example shows what the assembly output of a compiler can look like and how you can use it for improving the code.

```
// Example 7.24a
void Func(int a[], int & r) {
   int i;
   for (i = 0; i < 100; i++) {
      a[i] = r + i/2;
   }
}
```

The Intel compiler generates the following assembly code from example 7.24a (32-bit mode):

```
; Example 7.24a compiled to assembly:
ALIGN     4                              ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z                ; mangled function name
?Func@@YAXQAHAAH@Z  PROC NEAR            ; start of Func
; parameter 1: 8 + esp                   ; a
; parameter 2: 12 + esp                  ; r
```

```
$B1$1:                                  ; unused label
        push    ebx                     ; save ebx on stack
        mov     ecx, DWORD PTR [esp+8]  ; ecx = a
        xor     eax, eax                ; eax = i = 0
        mov     edx, DWORD PTR [esp+12] ; edx = r
$B1$2:                                  ; top of loop
        mov     ebx, eax                ; compute i/2 in ebx
        shr     ebx, 31                 ; shift down sign bit of i
        add     ebx, eax                ; i + sign(i)
        sar     ebx, 1                  ; shift right = divide by 2
        add     ebx, DWORD PTR [edx]    ; add what r points to
        mov     DWORD PTR[ecx+eax*4],ebx ; store result in array
        add     eax, 1                  ; i++
        cmp     eax, 100                ; check if i < 100
        jl      $B1$2                   ; repeat loop if true
$B1$3:                                  ; unused label
        pop     ebx                     ; restore ebx from stack
        ret                             ; return
        ALIGN   4                       ; align
?Func@@YAXQAHAAH@Z ENDP                 ; mark end of procedure
```

Most of the comments generated by the compiler have been replaced by my comments, in green. It takes some experience to get used to read and understand compiler-generated assembly code. Let me explain the above code in details. The funny looking name `?Func@@YAXQAHAAH@Z` is the name of `Func` with a lot of added information about the function type and its parameters. This is called name mangling. The characters '`?`', '`@`' and '`$`' are allowed in assembly names. The details about name mangling are explained in manual 5: "Calling conventions for different C++ compilers and operating systems". The parameters `a` and `r` are transferred on the stack at address `esp+8` and `esp+12` and loaded into `ecx` and `edx`, respectively. (In 64-bit mode, the parameters would be transferred in registers rather than on the stack). `ecx` now contains the address of the first element of the array `a` and `edx` contains the address of the variable that `r` points to. A reference is the same as a pointer in assembly code. Register `ebx` is pushed on the stack before it is used and popped from the stack before the function returns. This is because the register usage convention says that a function is not allowed to change the value of `ebx`. Only the registers `eax`, `ecx` and `edx` can be changed freely. The loop counter `i` is stored as a register variable in `eax`. The loop initialisation `i=0;` has been translated to the instruction `xor eax,eax`. This is a common way of setting a register to zero that is more efficient than `mov eax,0`. The loop body begins at the label `$B1$2:`. This is just an arbitrary name that the compiler has chosen for the label. It uses `ebx` as a temporary register for computing `i/2+r`. The instructions `mov ebx,eax` / `shr ebx,31` copies the sign bit of `i` into the least significant bit of `ebx`. The next two instructions `add ebx, eax` / `sar ebx,1` adds this to `i` and shifts one place to the right in order to divide `i` by 2. The instruction `add ebx, DWORD PTR [edx]` adds, not `edx` but the variable whose address is in `edx`, to `ebx`. The square bracket means use the value in `edx` as a memory pointer. This is the variable that `r` points to. Now `ebx` contains `i/2+r`. The next instruction `mov DWORD PTR [ecx+eax*4],ebx` stores this result in `a[i]`. Note how efficient the calculation of the array address is. `ecx` contains the address of the beginning of the array. `eax` holds the index, `i`. This index must be multiplied by the size (in bytes) of each array element in order to calculate the address of element number `i`. The size of an `int` is 4. So the address of array element `a[i]` is `ecx+eax*4`. The result `ebx` is then stored at address `[ecx+eax*4]`. This is all done in a single instruction. The CPU supports this kind of instructions for fast access to array elements. The instruction `add eax,1` is the loop increment `i++`. `cmp eax, 100` / `jl $B1$2` is the loop condition `i < 100`. It compares `eax` with `100` and jumps back to the `$B1$2` label if `i < 100`. `pop ebx` restores the value of `ebx` that was saved in the beginning. `ret` returns from the function.

The assembly listing reveals three things that can be optimized further. The first thing we notice is that it does some funny things with the sign bit of `i` in order to divide `i` by `2`. The compiler has not noticed that `i` can never be negative so that we don't have to care about the sign bit. We can tell it this by making `i` an `unsigned int` or by type-casting `i` to `unsigned int` before dividing by `2` (See page 112).

The second thing we notice is that the value pointed to by `r` is re-loaded from memory a hundred times. This is because we forgot to tell the compiler to assume no pointer aliasing (see page 62). Adding the compiler option "assume no pointer aliasing" (if valid) can possibly improve the code.

The third thing that can be improved is that `r+i/2` could be calculated by an induction variable because it is a staircase function of the loop index. The integer division prevents the compiler from making an induction variable unless the loop is rolled out by 2. (See page 56).

The conclusion is that we can help the compiler optimize example 7.24a by rolling out the loop by two and making an explicit induction variable. (This eliminates the need for the first two suggested improvements).

```
// Example 7.24b
void Func(int a[], int & r) {
   int i;
   int Induction = r;
   for (i = 0; i < 100; i += 2) {
      a[i] = Induction;
      a[i+1] = Induction;
      Induction++;
   }
}
```

The compiler generates the following assembly code from example 7.24b:

```
; Example 7.24b compiled to assembly:
ALIGN     4                                ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z                  ; mangled function name
?Func@@YAXQAHAAH@Z  PROC NEAR              ; start of Func
; parameter 1: 4 + esp                     ; a
; parameter 2: 8 + esp                     ; r
$B1$1:                                     ; unused label
        mov       eax, DWORD PTR [esp+4]   ; eax = address of a
        mov       edx, DWORD PTR [esp+8]   ; edx = address in r
        mov       ecx, DWORD PTR [edx]     ; ecx = Induction
        lea       edx, DWORD PTR [eax+400] ; edx = point to end of a
$B2$2:                                     ; top of loop
        mov       DWORD PTR [eax], ecx     ; a[i] = Induction;
        mov       DWORD PTR [eax+4], ecx   ; a[i+1] = Induction;
        add       ecx, 1                   ; Induction++;
        add       eax, 8                   ; point to a[i+2]
        cmp       edx, eax                 ; compare with end of array
        ja        $B2$2                    ; jump to top of loop
$B2$3:                                     ; unused label
        ret                                ; return from Func
        ALIGN     4
; mark_end;
?Func2@@YAXQAHAAH@Z ENDP
```

This solution is clearly better. The loop body now contains only six instructions rather than nine, even though it is doing two iterations in one. The compiler has replaced `i` by a second induction variable (`eax`) which contains the address of the current array element. Rather

than comparing `i` with `100` in the loop control it compares the array pointer `eax` to the address of the end of the array, which it has calculated in advance and stored in `edx`. Furthermore, this solution is using one register less so that it doesn't have to push and pop `ebx`.

8 Optimizing memory access

8.1 Caching of code and data

A cache is a proxy for the main memory in a computer. The proxy is smaller and closer to the CPU than the main memory and therefore it is accessed much faster. There may be two or three levels of cache for the sake of fastest possible access to the most used data.

The speed of CPU's is increasing faster than the speed of RAM memory. Efficient caching is therefore becoming more and more important.

8.2 Cache organization

It is useful to know how a cache is organized if you are making programs that have big data structures with non-sequential access and you want to prevent cache contention. You may skip this section if you are satisfied with more heuristic guidelines.

Most caches are organized into lines and sets. Let me explain this with an example. My example is a cache of 8 kb size with a line size of 64 bytes. Each line covers 64 consecutive bytes of memory. One kilobyte is 1024 bytes, so we can calculate that the number of lines is 8*1024/64 = 128. These lines are organized as 32 sets $\times$ 4 ways. This means that a particular memory address cannot be loaded into an arbitrary cache line. Only one of the 32 sets can be used, but any of the 4 lines in the set can be used. We can calculate which set of cache lines to use for a particular memory address by the formula: (*set*) = (*memory address*) / (*line size*) % (*number of sets*). Here, / means integer division with truncation, and % means modulo. For example, if we want to read from memory address a = 10000, then we have (*set*) = (10000 / 64) % 32 = 28. This means that a must be read into one of the four cache lines in set number 28. The calculation becomes easier if we use hexadecimal numbers because all the numbers are powers of 2. Using hexadecimal numbers, we have a = 0x2710 and (*set*) = (0x2710 / 0x40) % 0x20 = 0x1C. Reading or writing a variable from address 0x2710 will cause the cache to load the entire 64 or 0x20 bytes from address 0x2700 to 0x271F into one of the four cache lines from set 0x1C. If the program afterwards reads or writes to any other address in this range then the value is already in the cache so we don't have to wait for another memory access.

Assume that a program reads from address 0x2710 and later reads from addresses 0x2F00, 0x3700, 0x3F00 and 0x4700. These addresses all belong to set number 0x1C. There are only four cache lines in each set. If the cache always chooses the least recently used cache line then the line that covered the address range from 0x2700 to 0x271F will be evicted when we read from 0x4700. Reading again from address 0x2710 will cause a cache miss. But if the program had read from different addresses with different set values then the line containing the address range from 0x2700 to 0x271F would still be in the cache. The problem only occurs because the addresses are spaced a multiple of 0x800 apart. I will call this distance the critical stride. Variables whose distance in memory is a multiple of the critical stride will contend for the same cache lines. The critical stride can be calculated as (*critical stride*) = (*number of sets*) $\times$ (*line size*) = (*total cache size*) / (*number of ways*).

If a program contains many variables and objects that are scattered around in memory then there is a risk that several variables happen to be spaced by a multiple of the critical stride and cause contentions in the data cache. The same can happen in the code cache if there

are many functions scattered around in program memory. If several functions that are used in the same part of the program happen to be spaced by a multiple of the critical stride then this can cause contentions in the code cache. The subsequent sections describe various ways to avoid these problems.

More details about how caches work can be found in Wikipedia under CPU cache (en.wikipedia.org/wiki/L2_cache).

Table 8.1 lists the cache organization of some common CPU's.

Pro-cessor	Cache line size	Level 1 code cache		Level 1 data cache			Level 2 combined cache			Main memo-ry
		size	ways	size kb	ways	access time	size kb	ways	access time	access time
P3	32	16 kb	4	16	4	3	256	8	8	140
P4	64	12k uops	8	8	4	2	512	8	19	350
PM	64	32 kb	8	32	8	3	2048	8	10	80
Core2	64	32 kb	8	32	8	3	4096	16	14	185
Opte-ron	64	64 kb	2	64	2	3	1024	16	13	100

Table 8.1. Examples of cache organization of various microprocessors

Note that the numbers can vary. There are different versions of the microprocessors with different sizes of the level 2 cache. The access times are measured in clock cycles. The access time for main memory can vary a lot depending on the external hardware.

8.3 Functions that are used together should be stored together

The code cache works most efficiently if functions that are used near each other are also stored near each other in the code memory. The functions are usually stored in the order in which they appear in the source code. It is therefore a good idea to collect the functions that are used in the most critical part of the code together near each other in the same source file. Keep often used functions separate from seldom used functions, and put seldom used branches such as error handling in the end of a function or in a separate function.

Sometimes, functions are kept in different source files for the sake of modularity. For example, it may be convenient to have the member functions of a parent class in one source file and the derived class in another source file. If the member functions of parent class and derived class are called from the same critical part of the program then it can be advantageous to keep the two modules contiguous in program memory. This can be done by controlling the order in which the modules are linked together. The link order is usually the order in which the modules appear in the project window or makefile. You can check the order of functions in memory by requesting a map file from the linker. The map file tells the address of each function relative to the beginning of the program. The map file includes the addresses of library functions linked from static libraries (`.lib` or `.a`), but not dynamic libraries (`.dll` or `.so`). There is no easy way to control the addresses of dynamically linked library functions.

8.4 Variables that are used together should be stored together

Cache misses are very expensive. A variable can be fetched from the cache in just a few clock cycles, but it can take more than a hundred clock cycles to fetch the variable from RAM memory if it is not in the cache. Some examples of fetch times are given in table 8.1 above.

The cache works most efficiently if data that are used together are stored near each other in memory. Variables and objects should preferably be declared in the function in which they are used. Such variables and objects will be stored on the stack, which is very likely to be in the level-1 cache. The different kinds of variable storage are explained on page 18. Avoid global and static variables if possible, and avoid dynamic memory allocation (`new` and `delete`).

Object oriented programming can be an efficient way of keeping data together. Variables that are members of the same class will always be stored together (see page 39).

The order in which data are stored can be important if you have big data structures. For example, if a program has two arrays, `a` and `b`, and the elements are accessed in the order `a[0]`, `b[0]`, `a[1]`, `b[1]`, ... then you may improve the performance by organizing the data as an array of structures:

```
// Example 8.1a
int Func(int);
const int size = 1024;
int a[size], b[size], i;
...
for (i = 0; i < size; i++) {
   b[i] = Func(a[i]);
}
```

The data in this example can be accessed sequentially in memory if organized as follows:

```
// Example 8.1b
int Func(int);
const int size = 1024;
struct Sab {int a; int b;};
Sab ab[size];
int i;
...
for (i = 0; i < size; i++) {
   ab[i].b = Func(ab[i].a);
}
```

There will be no extra overhead in the program code for making the structure in example 8.1b. On the contrary, the code becomes simpler because it needs only calculate element addresses for one array rather than two.

Some compilers will use different memory spaces for different arrays even if they are never used at the same time. Example:

```
// Example 8.2a
void F1(int   x[]);
void F2(float x[]);

void F3(bool y) {
   if (y) {
      int a[1000];
      F1(a);
   }
   else {
      float b[1000];
      F2(b);
   }
}
```

Here it is possible to use the same memory area for `a` and `b` because their live ranges do not overlap. You can save a lot of cache space by joining `a` and `b` in a union:

```
// Example 8.2b
void F3(bool y) {
   union {
      int   a[1000];
      float b[1000];
   };
   if (y) {
      F1(a);
   }
   else {
      F2(b);
   }
}
```

Using a union is not a safe programming practice, of course, because you will get no warning from the compiler if the uses of `a` and `b` overlap. You should use this method only for big objects that take a lot of cache space. Putting simple variables into a union is not optimal because it prevents these use of register variables.

8.5 Alignment of data

A variable is accessed most efficiently if it is stored at a memory address which is divisible by the size of the variable. For example, a `double` takes 8 bytes of storage space. It should therefore preferably be stored at an address divisible by 8. The size should always be a power of 2. Objects bigger than 16 bytes should be stored at an address divisible by 16. You can generally assume that the compiler takes care of this alignment automatically.

The alignment of structure and class members may cause a waste of cache space, as explained in example 6.31 page 40.

You may choose to align large objects and arrays by the cache line size. This makes sure that the beginning of the object or array coincides with the beginning of a cache line. Some compilers will align large static arrays automatically but you may as well specify the alignment explicitly by writing:

```
__declspec(align(64)) int BigArray[1024]; // Windows syntax
```

or

```
int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
```

8.6 Dynamic memory allocation

Objects and arrays can be allocated dynamically with `new` and `delete`, or `malloc` and `free`. This can be useful when the amount of memory required is not known at compile time. Three typical uses of dynamic memory allocation can be mentioned here:

1. A large array can be allocated dynamically when the size of the array is not known at compile time.

2. A large number of objects can be allocated dynamically when the total number of objects is not known at compile time. These objects are typically organized as a linked list.

3. Text strings and similar objects of variable size can be allocated dynamically.

The advantages of dynamic memory allocation are:

- Gives a more clear program structure in some cases.

- Does not allocate more space than needed. This makes data caching more efficient than when a fixed-size array is made very big in order to cover the worst case situation of the maximum possible memory requirement.

- Useful when no reasonable upper limit to the required amount of memory space can be given in advance.

The disadvantages of dynamic memory allocation are:

- The process of dynamic allocation and deallocation of memory takes much more time than other kinds of storage. See page 18.

- The heap space becomes fragmented when objects of different sizes are allocated and deallocated in random order. This makes data caching inefficient.

- The heap manager will call a garbage collector when the heap space has become too fragmented. The garbage collector may start at unpredictable times and cause delays in the program flow at inconvenient times.

- It is the responsibility of the programmer to make sure that everything that has been allocated is also deallocated. Failure to do so will cause the heap to be filled up. This is a common programming error known as a memory leak.

- It is the responsibility of the programmer to make sure that no object is accessed after it has been deallocated. Failure to do so is also a common programming error.

It is important to weigh the advantages over the disadvantages when deciding whether to use dynamic memory allocation. There is no reason to use dynamic memory allocation when the size of an array or the number of objects is known at compile time.

The cost of dynamic memory allocation is negligible when the number of allocations is limited. Dynamic memory allocation can therefore be advantageous when a program has one or a few arrays of variable size. The alternative solution of making the arrays very big to cover the worst case situation is a waste of cache space. A situation where a program has several large arrays and where the size of each array is a multiple of the critical stride (see above, page 72) is likely to cause contentions in the data cache.

If the number of elements in an array grows during program execution then it is preferable to allocate the final array size right from the beginning rather than allocating more space step by step. You cannot increase the size of a memory block that has already been allocated. The only way to increase the size of an array after it has been allocated is to allocate a new bigger array and copy the contents of the old array into the beginning of the new bigger array. This is quite inefficient, of course, and causes the heap space to be fragmented.

A collection of a variable number of objects is often implemented as a linked list. A linked list is less efficient than a linear array for the following reasons:

- Each object is allocated separately. The allocation and deallocation takes a considerable amount of time.

- The objects may not be stored contiguously in the memory. This makes data caching less efficient.

- Extra memory space is used for the link pointers and for information stored by the heap manager for each allocated block.

- Walking through a linked list takes more time than looping through a linear array. No link pointer can be loaded until the previous link pointer has been loaded. This makes a critical dependence chain which prevents out-of-order execution.

An alternative to making a linked list is to allocate one big block of memory for all the objects rather than allocating a small block for each object. For example, a First-In-First-Out queue is implemented more efficiently in an circular buffer than a linked list. See page 121 for a number of useful templates for queues etc. A binary tree can be replaced by a sorted list or a hash list (see page 124).

The container class templates in the standard template library (STL) all use dynamic memory allocation. Some of these containers make a new allocation every time an element is added. The efficiency of the `vector<>` container class can be improved a lot by using the method `reserve()` to allocate the necessary amount of memory before adding elements. If you are using container class templates, then it can be advantageous to make your own templates that do not use dynamic memory allocation. Some useful container class templates without dynamic memory allocation are listed on page 121.

The implementation of `string`, `wstring` or `CString` uses `new` and `delete`. Therefore, the old C-style character arrays are much more efficient, but unfortunately also unsafe. To improve speed without jeopardizing safety, you may define your own class or template to handle character strings with overflow checking. See page 107 for an efficient way of checking for buffer overflow.

A little-known alternative to using `new` and `delete` is to allocate variable-size arrays with `alloca`. This is a function that allocates memory on the stack rather than the heap. The space is automatically deallocated when returning from the function in which `alloca` was called. There is no need to deallocate the space explicitly when `alloca` is used. The advantages of `alloca` over `new` and `delete` or `malloc` and `free` are:

- There is very little overhead to the allocation process because the microprocessor has hardware support for the stack.

- The memory space never becomes fragmented thanks to the first-in-last-out nature of the stack.

- Deallocation has no cost because it goes automatically when the function returns. There is no need for garbage collection.

- The allocated memory is contiguous with other objects on the stack, which makes data caching very efficient.

The following example shows how to make a variable-size array with `alloca`:

```
// Example 8.3
#include <malloc.h>

void SomeFunction (int n) {
   if (n > 0) {
      // Make dynamic array of n floats:
      float * DynamicArray = (float *)alloca(n * sizeof(float));
```

```
      // (Some compilers use the name _alloca)
      for (int i = 0; i < n; i++) {
         DynamicArray[i] = WhateverFunction(i);
         // ...
      }
   }
}
```

Obviously, a function should never return any pointer or reference to anything it has allocated with `alloca`, because it is deallocated when the function returns. `alloca` may not be compatible with structured exception handling. See the manual for your compiler for restrictions on using `alloca`.

8.7 Access data sequentially

A cache works most efficiently when the data are accessed sequentially. It works somewhat less efficiently when data are accessed backwards and much less efficiently when data are accessed in a random manner. This applies to reading as well as writing data.

Multidimensional arrays should be accessed with the last index changing in the innermost loop. This reflects the order in which the elements are stored in memory. Example:

```
// Example 8.4
const int NUMROWS = 100, NUMCOLUMNS = 100;
int matrix[NUMROWS][NUMCOLUMNS];
int row, column;
for (row = 0; row < NUMROWS; row++)
   for (column = 0; column < NUMCOLUMNS; column++)
      matrix[row][column] = row + column;
```

Do not swap the order of the two loops (except in Fortran where the storage order is opposite).

8.8 Cache contentions in large data structures

It is not always possible to access a multidimensional array sequentially. Some applications (e.g. in linear algebra) require other access patterns. This can cause severe delays if the distance between rows in a big matrix happen to be equal to the critical stride, as explained on page 72. This will happen if the size of a matrix line (in bytes) is a high power of 2.

The following example illustrates this. My example is a function which transposes a quadratic matrix, i.e. each element `matrix[r][c]` is swapped with element `matrix[c][r]`.

```
// Example 8.5a
const int SIZE = 64;          // number of rows/columns in matrix

void transpose(double a[SIZE][SIZE]) { // function to transpose matrix
   // define a macro to swap two array elements:
   #define swapd(x,y) {temp=x; x=y; y=temp;}

   int r, c;  double temp;
   for (r = 1; r < SIZE; r++) {        // loop through rows
      for (c = 0; c < r; c++) {        // loop columns below diagonal
         swapd(a[r][c], a[c][r]);      // swap elements
      }
   }
}

void test () {
```

```
    __declspec(__align(64))        // align by cache line size
    double matrix[SIZE][SIZE];     // define matrix
    transpose(matrix);             // call transpose function
}
```

Transposing a matrix is the same as reflecting it at the diagonal. Each element `matrix[r][c]` below the diagonal is swapped with element `matrix[c][r]` at its mirror position above the diagonal. The `c` loop in example 8.5a goes from the leftmost column to the diagonal. The elements at the diagonal remain unchanged.

The problem with this code is that if the elements `matrix[r][c]` below the diagonal are accessed row-wise, then the mirror elements `matrix[c][r]` above the diagonal are accessed column-wise.

Assume now that we are running this code with a 64×64 matrix on a Pentium 4 computer where the level-1 data cache is 8 kb = 8192 bytes, 4 ways, with a line size of 64. Each cache line can hold 8 `double`'s of 8 bytes each. The critical stride is 8192 / 4 = 2048 bytes = 4 rows.

Let's look at what happens inside the loop, for example when `r` = 28. We take the elements from row 28 below the diagonal and swap these elements with column 28 above the diagonal. The first eight elements in row 28 share the same cache line. But these eight elements will go into eight different cache lines in column 28 because the cache lines follow the rows, not the columns. Every fourth of these cache lines belong to the same set in the cache. When we reach element number 16 in column 28, the cache will evict the cache line that was used by element 0 in this column. Number 27 will evict number 1. Number 18 will evict number 2, etc. This means that all the cache lines we used above the diagonal have been lost at the time we are swapping column 29 with line 29. Each cache line has to be reloaded eight times because it is evicted before we need the next element. I have confirmed this by measuring the time it takes to transpose a matrix using example 8.5a on a Pentium 4 with different matrix sizes. The results of my experiment are given below. The time unit is clock cycles per array element.

Matrix size	Total kilobytes	Time per element
63×63	31	11.6
64×64	32	16.4
65×65	33	11.8
127×127	126	12.2
128×128	128	17.4
129×129	130	14.4
511×511	2040	38.7
512×512	2048	230.7
513×513	2056	38.1

Table 8.2. Time for transposition of different size matrices, clock cycles per element.

The table shows that it takes 40% more time to transpose the matrix when the size of the matrix is a multiple of the level-1 cache size. This is because the critical stride is a multiple of the size of a matrix line. The delay is less than the time it takes to reload the level-1 cache from the level-2 cache because the out-of-order execution mechanism can prefetch the data.

The effect is much more dramatic when contentions occur in the level-2 cache. The level-2 cache is 512 kb, 8 ways. The critical stride for the level-2 cache is 512 kb / 8 = 64 kb. This corresponds to 16 lines in a 512×512 matrix. My experimental results in table 8.2 show that

it takes six times as long time to transpose a matrix when contentions occur in the level-2 cache as when contentions do not occur. The reason why this effect is so much stronger for level-2 cache contentions than for level-1 cache contentions is that the level-2 cache cannot prefetch more than one line at a time.

A simple way of solving the problem is to make the rows in the matrix longer than needed in order to avoid that the critical stride is a multiple of the matrix line size. I tried to make the matrix 512×520 and leave the last 8 columns unused. This removed the contentions and the time consumption was down to 36.

There may be cases where it is not possible to add unused columns to a matrix. For example, a library of math functions should work efficiently on all sizes of matrices. An efficient solution in this case is to divide the matrix into smaller squares and handle one square at a time. This is called square blocking or tiling. This technique is illustrated in example 8.5b.

```
// Example 8.5b
void transpose(double a[SIZE][SIZE]) {
   // Define macro to swap two elements:
   #define swapd(x,y) {temp=x; x=y; y=temp;}
   // Check if level-2 cache contentions will occur:
   if (SIZE > 256 && SIZE % 128 == 0) {
      // Cache contentions expected. Use square blocking:
      int r1, r2, c1, c2; double temp;
      // Define size of squares:
      const int TILESIZE = 8;   // SIZE must be divisible by TILESIZE
      // Loop r1 and c1 for all squares:
      for (r1 = 0; r1 < SIZE; r1 += TILESIZE) {
         for (c1 = 0; c1 < r1; c1 += TILESIZE) {
            // Loop r2 and c2 for elements inside sqaure:
            for (r2 = r1; r2 < r1+TILESIZE; r2++) {
               for (c2 = c1; c2 < c1+TILESIZE; c2++) {
                  swapd(a[r2][c2],a[c2][r2]);
               }
            }
         }
         // At the diagonal there is only half a square.
         // This triangle is handled separately:
         for (r2 = r1+1; r2 < r1+TILESIZE; r2++) {
            for (c2 = r1; c2 < r2; c2++) {
               swapd(a[r2][c2],a[c2][r2]);
            }
         }
      }
   }
   else {
      // No cache contentions. Use simple method.
      // This is the code from example 8.5a:
      int r, c;  double temp;
      for (r = 1; r < SIZE; r++) {    // loop through rows
         for (c = 0; c < r; c++) {    // loop columns below diagonal
            swapd(a[r][c], a[c][r]);  // swap elements
         }
      }
   }
}
```

This code took 50 clock cycles per element for a 512×512 matrix in my experiments.

Contentions in the level-2 cache are so expensive that it is very important to do something about them. You should therefore be aware of situations where the number of columns in a matrix is a high power of 2. Contentions in the level-1 cache are less expensive. Using

complicated techniques like square blocking for the level-1 cache may not be worth the effort.

Square blocking and similar methods are further described in the book "Performance Optimization of Numerically Intensive Codes", by S. Goedecker and A. Hoisie, SIAM 2001.

8.9 Explicit cache control

Microprocessors with the SSE and SSE2 instruction sets have certain instructions that allow you to manipulate the data cache. These instructions are accessible from compilers that have support for intrinsic functions (i.e. Microsoft, Intel and Gnu). Other compilers need assembly code to access these instructions.

Function	Assembly name	Intrinsic function name	Instruction set
Prefetch	PREFETCH	_mm_prefetch	SSE
Store 4 bytes without cache	MOVNTI	_mm_stream_si32	SSE2
Store 8 bytes without cache	MOVNTQ	_mm_stream_pi	SSE
Store 16 bytes without cache	MOVNTPS	_mm_stream_ps	SSE
Store 16 bytes without cache	MOVNTPD	_mm_stream_pd	SSE2
Store 16 bytes without cache	MOVNTDQ	_mm_stream_si128	SSE2

Table 8.3. Cache control instructions.

There are other cache control instructions than the ones mentioned in table 8.3, such as flush and fence instructions, but these are hardly relevant to optimization.

Prefetching data

The prefetch instruction can be used for fetching a cache line that we expect to use later in the program flow. However, this did not improve the execution speed in any of the examples I have tested on a Pentium 4 processor. The reason is that modern processors prefetch data automatically thanks to out-of-order execution and advanced prediction mechanisms. It can be expected that future microprocessors will be able to automatically prefetch data for regular access patterns containing one or two streams with different strides. Therefore, you don't have to prefetch data explicitly if data access can be arranged in regular patterns with fixed strides.

Uncached memory store

An uncached write is more expensive than an uncached read because the write causes an entire cache line to be read and written back.

The so-called nontemporal write instructions (MOVNT) are designed to solve this problem. These instructions write directly to memory without loading a cache line. This is advantageous in cases where we are writing to uncached memory and we do not expect to read from the same or a nearby address again before the cache line would be evicted. Don't mix nontemporal writes with normal writes or reads to the same memory area.

The nontemporal write instructions are not suitable for example 8.5 because we are reading and writing from the same address so a cache line will be loaded anyway. If we modify example 8.5 so that it writes only, then the effect of nontemporal write instructions becomes noticeable. The following example transposes a matrix and stores the result in a different array.

```
// Example 8.6a
const int SIZE = 512; // number of rows and columns in matrix
```

```
// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE]) {
   int r, c;
   for (r = 0; r < SIZE; r++) {
      for (c = 0; c < SIZE; c++) {
         a[c][r] = b[r][c];
      }
   }
}
```

This function writes to matrix a in a column-wise manner where the critical stride causes all writes to load a new cache line in both the level-1 and the level-2 cache. Using the nontemporal write instruction prevents the level-2 cache from loading any cache lines for matrix a:

```
// Example 8.6b.
#include "xmmintrin.h"  // header for intrinsic functions

// This function stores a double without loading a cache line:
static inline void StoreNTD(double * dest, double const & source) {
   _mm_stream_pi((__m64*)dest, *(__m64*)&source);  // MOVNTQ
   _mm_empty();                                    // EMMS
}

const int SIZE = 512; // number of rows and columns in matrix
// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE]) {
   int r, c;
   for (r = 0; r < SIZE; r++) {
      for (c = 0; c < SIZE; c++) {
         StoreNTD(&a[c][r], b[r][c]);
      }
   }
}
```

The execution times per matrix cell for different matrix sizes were measured on a Pentium 4 computer. The measured results were as follows:

Matrix size	Time per element Example 8.6a	Time per element Example 8.6b
64×64	14.0	80.8
65×65	13.6	80.9
512×512	378.7	168.5
513×513	58.7	168.3

Table 8.4. Time for transposing and copying different size matrices, clock cycles per element.

As table 8.4 shows, the method of storing data without caching is advantageous if, and only if, a level-2 cache miss can be expected. The 64×64 matrix size causes misses in the level-1 cache. This has hardly any effect on the total execution time because the cache miss on a store operation doesn't delay the subsequent instructions. The 512×512 matrix size causes misses in the level-2 cache. This has a very dramatic effect on the execution time because the memory bus is saturated. This can be ameliorated by using nontemporal writes. If the cache contentions can be prevented in other ways, as explained in chapter 8.8, then the nontemporal write instructions are not optimal.

There are certain restrictions on using the instructions listed in table 8.3. All these instructions require that the microprocessor has the SSE or SSE2 instruction set, as listed in

the table. The 16-byte instructions `MOVNTPS`, `MOVNTPD` and `MOVNTDQ` require that the operating system has support for XMM registers; see page 105.

The Intel compiler can insert nontemporal writes automatically in vectorized code when the `#pragma vector nontemporal` is used. However, this does not work in example 8.6b.

The `MOVNTQ` instruction must be followed by an `EMMS` instruction before any floating point instructions. This is coded as `_mm_empty()` as shown in example 8.6b. The `MOVNTQ` instruction cannot be used in 64-bit device drivers for Windows.

9 Using multiple CPU kernels

The speed of CPU-intensive programs is limited by the clock frequency of the CPU. The way to increase throughput when the clock frequency is limited is to do multiple things at the same time. There are three ways to do things in parallel:

1. Using multiple CPU's or multi-kernel CPU's, as described in this chapter.

2. Using the out-of-order capabilities of modern CPU's, as described in chapter 10.

3. Using the vector operations of modern CPU's, as described in chapter 11.

It is important to distinguish between coarse-grained parallelism and fine-grained parallelism when deciding whether it is advantageous to do things in parallel. Coarse-grained parallelism refers to the situation where a long sequence of operations can be carried out independently of other tasks that are running in parallel. Fine-grained parallelism is the situation where a task can be divided into many small subtasks, but it is not possible to work for very long on a particular subtask before coordination with other subtasks is necessary.

Some CPU's today have multiple kernels, and it can be expected that most CPU's will have at least two kernels in the future. The use of multiple CPU kernels is useful for coarse-grained parallelism but not for fine-grained parallelism because communication and synchronization between the different kernels is slow. The methods described in chapter 10 and 11 are more useful for fine-grained parallelism.

The way to use multiple CPU kernels is to divide the work into multiple threads. The use of threads is discussed on page 47. We should preferably have no more threads with the same priority than the number of kernels or logical processors available in the system. The number of logical processors can be determined by a system call (e.g. `GetSystemInfo` in Windows).

There are three ways to divide the workload between multiple CPU kernels:

1. Define multiple threads and put an equal amount of work into each thread. This method works with all compilers.

2. Use automatic parallelization. The Intel compiler can automatically detect opportunities for parallelization in the code and divide it into multiple threads.

3. Use OpenMP directives. OpenMP is a standard for specifying parallel processing in C++ and Fortran. These directives are supported by Intel compilers and certain other compilers. See www.openmp.org and the Intel compiler manual for details.

Some versions of Intel Pentium 4 CPU's use a technology called hyper-threading for making one CPU kernel appear as two in order to run two threads simultaneously. The CPU has two logical processors but only one kernel. The advantage of hyper-threading is limited

because two threads running in parallel will compete for the same execution units in the CPU kernel and use the same caches. Hyper-threading may be advantageous if the two threads are of a very different nature and use different resources, e.g. if one thread does a memory-intensive job and the other thread does a CPU-intensive job. It is not advantageous to divide a job into two tasks that are similar in nature on a CPU with one hyper-threading kernel because the two threads will compete for the same resources.

Some CPU's have two or more kernels, and each kernel may have one or two logical processors (hyper-threading). The optimal number of threads for CPU intensive tasks is equal to the number of kernels, not the number of logical processors, if the different tasks are likely to compete for the same resources in the kernel.

Two or more logical processors may share the same cache. This is a disadvantage if they run different threads that use different areas of memory. But it is an advantage if two threads use the same memory area and have to communicate with each other. The Intel compiler is capable of making two threads where one thread is used for prefetching data for the other thread.

It may not be possible to get more information than the number of logical processors from operating system calls. Additional information about the number of kernels and caches can be obtained from the CPUID instruction. See the manuals from the microprocessor vendors for details.

A CPU with multiple kernels or a system with multiple CPU's is very useful for running multiple tasks in parallel. A CPU-intensive program with possibilities for coarse-grained parallelism can benefit very much from dividing the calculations into multiple threads. The workload should preferably be divided evenly between the threads. Parallel processing may not be advantageous for fine-grained parallelism because the communication and synchronization between threads consume extra resources.

Running multiple threads on a system with only one logical processor is not an advantage if the threads are competing for the same resources. But it can be a good idea to put time-consuming calculations in a separate thread with lower priority than the user interface. It is also useful to put file access and network access in separate threads so that one thread can do calculations while another thread is waiting for response from a hard disk or network.

10 Out of order execution

Modern x86 CPU's can execute instructions out of order or do more than one thing at the same time. The following example shows how to take advantage of this capability:

```
// Example 10.1a
float a, b, c, d, y;
y = a + b + c + d;
```

This expression is calculated as `((a+b)+c)+d`. This is a dependence chain where each addition has to wait for the result of the preceding one. You can improve this by writing:

```
// Example 10.1b
float a, b, c, d, y;
y = (a + b) + (c + d);
```

Now the two parentheses can be calculated independently. The CPU will start to calculate `(c+d)` before it has finished the calculation of `(a+b)`. This can save several clock cycles. You cannot assume that an optimizing compiler will change the code in example 10.1a to 10.1b automatically, although it appears to be an obvious thing to do. The reason why

compilers do not make this kind of optimizations is that it may cause a loss of precision, as explained on page 58. You have to set the parentheses manually.

The effect of dependence chains is stronger when they are long. This is often the case in loops. Consider the following example, which calculates the sum of 100 numbers:

```
// Example 10.2a
const int size = 100;
float list[size], sum = 0;  int i;
for (i = 0; i < size; i++) sum += list[i];
```

This has a long dependence chain. If a floating point addition takes 5 clock cycles, then this loop will take approximately 500 clock cycles. You can improve the performance dramatically by unrolling the loop and splitting the dependence chain in two:

```
// Example 10.2b
const int size = 100;
float list[size], sum1 = 0, sum2 = 0;  int i;
for (i = 0; i < size; i += 2) {
   sum1 += list[i];
   sum2 += list[i+1];}
sum1 += sum2;
```

If the microprocessor is doing an addition to `sum1` from time T to T+5, then it can do another addition to `sum2` from time T+1 to T+6, and the whole loop will take only 256 clock cycles.

Calculations in a loop where each iteration needs the result of the preceding one is called a loop-carried dependence chain. Such dependence chains can be very long and very time-consuming. There is a lot to gain if such dependence chains can be broken up. The two summation variables `sum1` and `sum2` are called accumulators. The optimal number of accumulators for floating point addition and multiplication is three or four. Current CPU's have only one floating point addition unit, but this unit is pipelined, as explained above, so that it can start a new addition before the preceding addition is finished.

Unrolling a loop becomes a little more complicated if the number of iterations is not divisible by the unroll factor. For example, if the number of elements in `list` in example 10.2b was an odd number then we would have to add the last element outside the loop or add an extra dummy element to `list` and make this extra element zero.

It is not necessary to unroll a loop and use multiple accumulators if there is no loop-carried dependence chain. A microprocessor with out-of-order capabilities can overlap the iterations and start the calculation of one iteration before the preceding iteration is finished. Example:

```
// Example 10.3
const int size = 100;  int i;
float a[size], b[size], c[size];
float register temp;
for (i = 0; i < size; i++) {
   temp = a[i] + b[i];
   c[i] = temp * temp;
}
```

Microprocessors with out-of-order capabilities are very smart. They can detect that the value of register `temp` in one iteration of the loop in example 10.3 is independent of the value in the previous iteration. This allows it to begin calculating a new value of `temp` before it is finished using the previous value. It does this by assigning a new physical register to `temp` even though the logical register that appears in the machine code is the same. This is called register renaming. The CPU can hold many renamed instances of the same logical register.

This advantage comes automatically. There is no reason to unroll the loop and have a `temp1` and `temp2`. All modern CPU's are capable of register renaming and doing multiple calculations in parallel if certain conditions are satisfied. The conditions that make it possible for the CPU to overlap the calculations of loop iterations are:

- No loop-carried dependence chain. Nothing in the calculation of one iteration should depend on the result of the previous iteration (except for the loop counter, which is calculated fast if it is an integer).

- All intermediate results should be saved in registers, not in memory. The renaming mechanism works only on registers, not on variables in memory or cache. Most compilers will make `temp` a register variable in example 10.3 even without the `register` keyword. The Borland compiler cannot make floating point register variables, but will save `temp` in memory. This prevents the CPU from overlapping calculations.

- The loop branch should be predicted. This is no problem if the repeat count is large or constant. If the loop count is small and changing then the CPU may occasionally predict that the loop exits, when in fact it does not, and therefore fail to start the next calculation. However, the out-of-order mechanism allows the CPU to increment the loop counter ahead of time so that it may detect the misprediction before it is too late. You should therefore not be too worried about this condition.

In general, the out-of-order execution mechanism works automatically. However, there are a couple of things that the programmer can do to take maximum advantage of out-of-order execution. The most important thing is to avoid long dependence chains. Another thing that you can do is to mix different kinds of operations in order to divide the work evenly between the different execution units in the CPU. It can be advantageous to mix integer and floating point calculations as long as you don't need conversions between integers and floating point numbers. It can also be advantageous to mix floating point addition with floating point multiplication, to mix simple integer with vector integer operations, and to mix mathematical calculations with memory access.

11 Using vector operations

Newer microprocessors have vector instructions that make it possible to do operations on all elements of a vector simultaneously. This is also called Single-Instruction-Multiple-Data (SIMD) operations or MMX and XMM instructions.

Vector operations are useful when doing calculations on large data sets where the same operation is performed on multiple data elements and the program logic allows parallel calculations. Examples are image processing, sound processing, and mathematical operations on vectors and matrixes. Algorithms that are inherently serial, such as most sorting algorithms, are not suited for vector operations. Algorithms that rely heavily on table lookup or require a lot of data shuffling, such as many encryption algorithms, cannot easily be implemented as vector operations.

The vector operations use a set of special vector registers called XMM registers. Each XMM register is 128 bits wide. This register can be organized as a vector of sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, two 64-bit integers, four `float`'s or two `double`'s. For example, you can add two vectors of each four `float`'s in one operation. This is four additions in one.

It is also possible to use the older MMX registers that are 64 bits wide, but only in code that has no floating point operations because the MMX registers are aliased upon the floating

point registers. It is preferred to use the 128-bit XMM registers which do not have this problem.

In most cases the vector operations require that the SSE2 or later instruction set is enabled in both the compiler, the CPU, and the operating system. See page 105 for how to check this.

Vector operations are particularly fast on processors with full 128-bit execution units. At the time of writing (August 2006), the only x86 processor with true 128-bit execution units is the Intel Core 2. This processor will calculate a vector just as fast as a scalar (Scalar means not a vector). Earlier processors are using a 64-bit execution unit twice for calculating a 128-bit vector. There is hardly any advantage in making a vector of two double precision floats on processors with 64-bit units when the time it takes to calculate a vector is double the time it takes to calculate two scalars.

The use of vector operations is more advantageous the smaller the data elements are. For example, it takes the same time to add two vectors of four `float`'s and two vectors of two `double`'s. With `float`'s you get four additions in the same time that it takes to do two additions with `double`'s. It is almost always advantageous to vectorize a loop containing float's, 8-bit integers or 16-bit integers. It may not be advantageous to vectorize a loop containing double's on processors with 64-bit execution units. Current AMD processors and Intel processors with NetBurst architecture are doing simple integer additions faster than vector additions. It may not be advantageous to use vector operations for adding 32-bit integers on these processors.

11.1 Automatic vectorization

The Intel compiler can use vector operations automatically in cases where the parallelism is obvious. See the compiler documentation for detailed instructions. (The Codeplay compiler can also use vector instructions automatically, but unfortunately it does so even when it is not optimal. It is therefore preferred to use the Intel compiler).

Example:

```
// Example 11.1a
// Use Intel compiler f. Windows with options /Ox /QxB
// or  Intel compiler f. Linux with options -O2 -xB

const int size = 100;

void AddTwo(float aa[size], float bb[size]) {
   #pragma vector aligned       // assume aa and bb are aligned by 16
   #pragma ivdep                // assume no pointer aliasing
   for (int i = 0; i < size; i++) {
      aa[i] = bb[i] + 2.0f;
   }
}

void CallAddTwo() {
   __declspec (align(16))       // arrays must be aligned by 16
   float a[size], b[size];      // define two arrays
   AddTwo(a, b);                // call AddTwo function
}
```

The vector operations require that arrays `a` and `b` be aligned by 16, i.e. stored at an address divisible by 16. Here, the `#pragma vector aligned` tells the Intel compiler that `aa` and `bb` are properly aligned. The compiler cannot assume this because the function `AddTwo` could be called with unaligned arrays as parameters. The `#pragma ivdep` tells the Intel compiler to assume that there is no pointer aliasing (see page 62). An alternative way of

giving the compiler this information is to use the `__restrict__` keyword to tell the compiler that there is no pointer aliasing, and use the `__assume_aligned` directive to tell the compiler that the arrays are aligned by 16:

```
// Example 11.1b
void AddTwo(float * __restrict__ aa, float * __restrict__ bb) {
   // Tell Intel compiler that pointers are aligned:
   __assume_aligned(aa,16); __assume_aligned(bb,16);
   for (int i = 0; i < size; i++) {
      aa[i] = bb[i] + 2.0f;
   }
}
```

Example 11.1a and 11.1b are equivalent. Here, the compiler will use the vector instructions to do four additions at a time. This will increase the execution speed by a factor two or more.

The automatic vectorization works optimally if the following conditions are satisfied:

1. Use Intel compiler.

2. Use appropriate compiler options to enable the desired instruction set (`/Ox /QxB` for Windows, `-O2 -xB` for Linux)

3. Align arrays and structures by 16.

4. If the alignment is not visible in the scope of the function where you want vectorization then it is necessary to tell the compiler that data are aligned.

5. If the arrays or structures are accessed through pointers or references then tell the compiler explicitly that pointers do not alias, if appropriate.

6. The loop count should preferably be a constant which is divisible by the number of elements in a vector.

The compiler can vectorize the code if only conditions 1 - 3 are satisfied, but the code will be more efficient if conditions 4 - 6 are also satisfied. If conditions 4 or 5 are not satisfied then the compiler will generate two versions of the code and check the data addresses at runtime. If data addresses are misaligned or overlapping then it will use the non-vectorized version of the code. If condition 6 is not satisfied then the compiler will generate extra code to take care of the possible remaining data (The number of remaining data is the remainder of the loop count divided by the number of elements in a vector). These extra runtime checks is a waste of time and the extra code is a waste of cache space. You should therefore make sure that condition 4 - 6 are satisfied if possible.

The compiler tells when it has vectorized a loop. The output says "remark: LOOP WAS VECTORIZED" with an indication of the line number. The compiler does not tell if it has inserted extra runtime checks for condition 4 - 6, not even with the maximum optimization report level. You have to look at the assembly output listing to see this (see page 69).

The alignment of the data you want to vectorize is important. If you tell the compiler to assume that data are aligned when in fact they are not then the program will crash with a general protection exception.

The Intel compiler can also use vector operations where there is no loop if the same operation is performed on a sequence of consecutive variables. Example:

```
// Example 11.2
```

```
// Use Intel compiler f. Windows with options /Ox /QxB
// or  Intel compiler f. Linux with options -O2 -xB

__declspec(align(16))     // Make all instances of S1 aligned
struct S1 {               // Structure of 4 floats
   float a, b, c, d;
};

void Func() {
   S1 x, y;
   ...
   #pragma vector always  // Tell compiler to vectorize
   x.a = y.a + 1.;
   x.b = y.b + 2.;
   x.c = y.c + 3.;
   x.d = y.d + 4.;
};
```

A structure of four `float`'s fits into a 128-bit XMM register. In example 11.2, the compiler will store the constant vector `(1.,2.,3.,4.)` in static memory. When `Func` is called, it will load the structure `y` into an XMM register, add the constant vector `(1.,2.,3.,4.)`, and store the result in `x`. This may or may not be faster than doing one addition at a time. The `#pragma vector always` tells the compiler to override the normal heuristics and vectorize the code regardless of whether it is judged to be advantageous or not. The compiler is not always able to predict correctly whether vectorization will be advantageous or not. You can use the `#pragma vector always` to tell the compiler to vectorize, or `#pragma novector` to tell the compiler not to vectorize. The pragmas must be placed immediately before the loop or the series of statements that you want them to apply to.

It is recommended to use the smallest data size that fits the application. Example:

```
// Example 11.3
// Use Intel compiler f. Windows with options /Ox /QxB
// or  Intel compiler f. Linux with options -O2 -xB

const int size = 1000;
__declspec(align(16))         // align a and b by 16
int a[size], b[size];
...
for (int i = 0; i < size, i++) {
   a[i] = b[i] * 2 + 1;
}
```

The Intel compiler will vectorize this loop, which takes approximately 1100 clock cycles on a Pentium 4. If the values in `a` and `b` can fit into 16-bit integers rather than the default 32-bit integers without overflow, then the execution speed can be almost doubled by changing the type of `a` and `b` to `short int`. If 8-bit integers are sufficient then the speed can be increased further by changing the type of `a` and `b` to `char`. Without vectorization, the loop takes 4000 clock cycles regardless of integer size.

The vector instructions cannot multiply integers of any other size than 16 bits. The multiplication by 2 in example 11.3 is done by adding the value to itself. If `b[i]` is multiplied by another factor such as 10 then the loop can only be vectorized if `a` and `b` are of type `short int`. It is not possible to make integer division in vectors.

11.2 Explicit vectorization

It is difficult to predict whether the compiler will vectorize a loop or not. Some loops are vectorized automatically when the elements are floating point, but not if they are integers.

And some loops are vectorized in 32-bit mode but not in 64-bit mode. The following example shows a code that the Intel compiler does not vectorize automatically. The code has a branch that chooses between two expressions for every element in the arrays:

```
// Example 11.4a
const int size = 128;        // Array size
short int aa[size], bb[size], cc[size], dd[size];

void SelectAndAdd(short int g) {
   for (int i = 0; i < size; i++) {
      aa[i] = (bb[i] > 0) ? (cc[i] + 2) : (dd[i] + g);
   }
}
```

Using intrinsic functions

It is possible to tell the compiler which vector operations to use by using the so-called intrinsic functions. This is useful in situations like example 11.4a where the Intel compiler doesn't vectorize the code automatically. It is also useful for compilers that cannot vectorize the code automatically.

Intrinsic functions are primitive operations in the sense that each intrinsic function call is translated to just one or a few machine instructions. Intrinsic functions are supported by the Intel, Gnu and Microsoft compilers. The Codeplay compiler also has some support for intrinsic functions, but the function names are not compatible with the other compilers.

The best performance is obtained with the Gnu or the Intel compiler. The Microsoft and Codeplay compilers do not optimize code containing intrinsic functions as good as the Gnu and Intel compilers do.

We want to vectorize the loop in example 11.4a so that we can handle eight elements at a time in vectors of eight 16-bit integers. The branch inside the loop is implemented by making a bit-mask which is all 1's when `bb[i] > 0` is true, and all 0's when false. The expression `cc[i]+2` is AND'ed with this mask, and `dd[i]+g` is AND'ed with the inverted mask. The expression that is AND'ed with all 1's is unchanged, while the expression that is AND'ed with all 0's gives zero. An OR combination of these two gives the chosen expression. Example 11.4b shows how this can be implemented with intrinsic functions:

```
// Example 11.4b
// SSE2 instruction set must be enabled
#include <emmintrin.h>       // Define intrinsic functions

const int size = 128;        // Array size
__declspec(align(16))        // Align arrays by 16
short int aa[size], bb[size], cc[size], dd[size];

// Macro to type-cast an array address to a vector address:
#define ToVectorAddress(x) ((__m128i*)&(x))

void SelectAndAdd(short int g) {
   // Define local vector variables:
   __m128i a, b, c, d, mask, zero, two, g_broadcast;
   // Set zero to a vector of (0,0,0,0,0,0,0,0)
   zero = _mm_set1_epi16(0);
   // Set two to a vector of (2,2,2,2,2,2,2,2)
   two  = _mm_set1_epi16(2);
   // Set g_broadcast to a vector of (g,g,g,g,g,g,g,g)
   g_broadcast = _mm_set1_epi16(g);

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < size; i += 8) {
      // Load eight consecutive elements from bb into vector b:
```

```
            b = _mm_load_si128(ToVectorAddress(bb[i]));
            // Load eight consecutive elements from cc into vector c:
            c = _mm_load_si128(ToVectorAddress(cc[i]));
            // Load eight consecutive elements from dd into vector d:
            d = _mm_load_si128(ToVectorAddress(dd[i]));
            // Add 2 to each element in vector c
            c = _mm_add_epi16(c, two);
            // Add g to each element in vector d
            d = _mm_add_epi16(d, g_broadcast);
            // Compare each element in b to 0 and generate a bit-mask:
            mask = _mm_cmpgt_epi16(b, zero);
            // AND each element in vector c with the corresponding bit-mask:
            a = _mm_and_si128(c, mask);
            // AND each element in vector d with the inverted bit-mask:
            mask = _mm_andnot_si128(mask, d);
            // OR the results of the two AND operations:
            a = _mm_or_si128(a, mask);
            // Store the result vector in eight consecutive elements in aa:
            _mm_store_si128(ToVectorAddress(aa[i]), a);
        }
    }
```

The resulting code will be very efficient because it handles eight elements at a time and it avoids the branch inside the loop. Example 11.4b executes three to seven times faster than example 11.4a, depending on how predictable the branch inside the loop is.

The intrinsic vector functions have names that begin with `_mm_`. These functions are listed in the header file `emmintrin.h` and explained in the documentation for the Intel C++ compiler and in "IA-32 Intel Architecture Software Developer's Manual" Volume 2A and 2B under "Instruction set reference".

Using vector classes

Programming in the way of example 11.4b is quite tedious indeed, and the program is not very readable. It is possible to write the same in a more human-intelligible way by wrapping the vectors into classes. A set of vector classes for this purpose is provided in various header files that come with the Intel compiler. See the documentation for the Intel C++ compiler for details.

The following table lists the available vector classes. Including `dvec.h` will give you access to all of these.

Vector class	Size of each element, bits	Number of elements	Type of elements	Total size of vector, bits	Header file
Is8vec8	8	8	char	64	ivec.h
Iu8vec8	8	8	unsigned char	64	ivec.h
Is16vec4	16	4	short int	64	ivec.h
Iu16vec4	16	4	unsigned short int	64	ivec.h
Is32vec2	32	2	int	64	ivec.h
Iu32vec2	32	2	unsigned int	64	ivec.h
I64vec1	64	1	__int64	64	ivec.h
Is8vec16	8	16	char	128	dvec.h
Iu8vec16	8	16	unsigned char	128	dvec.h
Is16vec8	16	8	short int	128	dvec.h
Iu16vec8	16	8	unsigned short int	128	dvec.h
Is32vec4	32	4	int	128	dvec.h
Iu32vec4	32	4	unsigned int	128	dvec.h
I64vec2	64	2	__int64	128	dvec.h
F32vec4	32	4	float	128	fvec.h
F64vec2	64	2	double	128	dvec.h

Table 11.1. Vector classes

It is not recommended to use the 64-bit vectors in `ivec.h` because these are incompatible with floating point code. If you do use the 64-bit vectors then you have to execute `_mm_empty()` after the 64-bit vector operations and before any floating point code. The 128-bit vectors do not have this problem.

Rewriting example 11.4b with the use of vector classes makes the code look simpler:

```
// Example 11.4c
// SSE2 instruction set must be enabled
#include <dvec.h>            // Define vector classes

const int size = 128;        // Array size
__declspec(align(16))        // Align arrays by 16
short int aa[size], bb[size], cc[size], dd[size];

// Macro to type-cast an array to a vector:
#define ToVector(x) (*(Is16vec8*)&(x))

void SelectAndAdd(short int g) {
   // Define local vector class variables:
   Is16vec8 a, b, c, d, zero;

   // Set zero to a vector of (0,0,0,0,0,0,0,0)
   zero = (Is16vec8)_mm_set1_epi16(0);

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < size; i += 8) {

      b = ToVector(bb[i]);       // Load eight elements from bb[i]
      c = ToVector(cc[i]);       // Load eight elements from cc[i]
      d = ToVector(dd[i]);       // Load eight elements from dd[i]
```

```
      d += _mm_set1_epi16(g);    // Make eight copies of g and add
      c += _mm_set1_epi16(2);    // Make eight copies of 2 and add
      a = select_gt(b,zero,c,d); // a = b > 0 ? c : d;
      ToVector(aa[i]) = a;       // Store eight elements in aa[i]
   }
}
```

Example 11.4c looks simpler than example 11.4b because we can use the operator + or += and the predefined function `select_gt` on the vector class objects.

We still haven't got rid of all the intrinsic functions and the ugly type-casting macro `ToVector`. We can make the code even more readable by defining our own vector class and operators, as explained below.

Redefining a vector class

It is possible to define your own vector classes with associated operators and functions. This is done most easily by inheriting from the predefined vector classes. You can define new member functions and operators for the derived class, but you cannot add any data members without destroying the vectorization. The vector classes have a size that fits into the vector registers. Adding any data members or virtual member functions will make the class objects bigger so they don't fit into the vector registers any more.

The following example shows how example 11.4c can be made even more readable by redefining the vector classes.

```
// Example 11.4d
// SSE2 instruction set must be enabled
#include <dvec.h>            // Define vector classes

const int size = 128;        // Array size
__declspec(align(16))        // Align arrays by 16
short int aa[size], bb[size], cc[size], dd[size];

// Redefine class for vector of eight 16-bit integers:
class Int16Vector8 : public Is16vec8 {
   public:
   // Default constructor:
   Int16Vector8() {};
   // Constructor to convert from parent type:
   Int16Vector8(Is16vec8 const & x) : Is16vec8(x) { }
   // Constructor to convert from type __m128i used in intrinsics:
   Int16Vector8(__m128i const & x) : Is16vec8(x) { }
   // Constructor to broadcast integer into all elements of vector:
   Int16Vector8(short int const n) {*this = _mm_set1_epi16(n);}
   // Constructor to load eight values from array:
   Int16Vector8(short int const * p) {
      *this = *(Int16Vector8 const*)p;}
   // Member function to store eight values into array:
   void store(short int * p) {*(Int16Vector8*)p = *this;}
};

// Define another class for a vector of eight 16-bit masks,
// Each mask can only be 0xFFFF for true or 0x0000 for false.
class Bool16Vector8 : public M128 {
   public:
   // Default constructor:
   Bool16Vector8() {};
   // Constructor to convert from type __m128i used in intrinsics:
   Bool16Vector8(__m128i const x) : M128(x) { }
};

// Operator + adds the same integer to all elements of vector
```

```
static inline Int16Vector8 operator + (Int16Vector8 const & a,
short int const b) {
   return a + Int16Vector8(b);}

// Operator > compares two vectors and returns a vector of masks.
// Each mask is 0xFFFF for true, 0x0000 for false.
static inline Bool16Vector8 operator > (Int16Vector8 const & a,
Int16Vector8 const & b) {
   return cmpgt(a, b);}

// Operator > compares all elements of vector with the same integer
// and returns a vector of masks.
static inline Bool16Vector8 operator > (Int16Vector8 const & a,
short int const b) {
   return a > Int16Vector8(b);}

// Function select does the same as the ?: operator,
// element by element:
// return_value[i] = s[i] ? a[i] : b[i]
// s is a vector of 8 masks, each mask is
// 0xFFFF for true, 0xFFFF for false.
static inline Int16Vector8 select (Bool16Vector8 const & s,
Int16Vector8 const & a, Int16Vector8 const & b) {
   return (s & a) | andnot(s, b);}

void SelectAndAdd(short int g) {
   int i;
   // Define local vector class variables:
   Int16Vector8 a, b, c, d;

   // Roll out loop by eight to fit the eight-element vectors:
   for (i = 0; i < size; i += 8) {

      b = bb + i;            // Load eight elements from bb[i]
      c = cc + i;            // Load eight elements from cc[i]
      d = dd + i;            // Load eight elements from dd[i]
      a = select(b > 0, c + 2, d + g); // Select c+2 or d+g
      a.store(aa + i);       // Store eight elements in aa[i]
   }
}
```

Example 11.4d does exactly the same as example 11.4b and c. The `SelectAndAdd` function looks simpler now because all the intrinsic functions and type conversions have been wrapped into the new class definitions. Defining the new class `Int16Vector8` which inherits from class `Is16vec8` allows us to redefine the constructors and operators to suit our needs. The constructor that converts from `Is16vec8` to `Int16Vector8` makes it possible to use all the operators that are defined for `Is16vec8` on vectors of class `Int16Vector8`. For example, it would be legal to write `a = b * c + d`.

The constructor that converts from `__m128i` to `Int16Vector8` makes it possible to use the intrinsic functions that return `__m128i`.

The constructor that converts from `short int` to `Int16Vector8` makes it easy to make a vector with the same value in all elements.

The constructor that converts from `short int const *` to `Int16Vector8` makes it easy to make a vector out of eight consecutive elements in an array. Note that the first element must be properly aligned. For example `a = aa` in the above example would be legal, `a = aa + 1` would cause an error, `a = aa + 8` would be legal.

The `store` member function makes it easy to store a vector into eight consecutive elements of an array. The first element must be properly aligned.

The `operator +` is already defined, but I have made an overloaded version for adding an integer to all elements in an array. Without this operator we would have to replace `c+2` by `c+Int16Vector8(2)`. We may define similar operators for `-` and `*`.

The `operator >` is not predefined, so we have to define it. I have made two overloaded versions of the `operator >`, one where the second operand is a vector and one where the second operand is an integer.

The `operator >` returns a vector of eight 16-bit masks that are used as Booleans. I have defined a separate class, `Bool16Vector8`, for this vector of Booleans. We could have used `Int16Vector8` for this purpose, but I think it is safer to define a separate class for a vector of elements that can have no other values than `0x0000` and `0xFFFF`. This allows us to make sure that the `select` function is called with only this type as the `s` parameter, and not just any vector.

Note that I have declared all the functions and operators `static` and `inline` in order to make it easier for the compiler to optimize the code. The vector parameters are declared as references (`&`) in order to make the parameter transfer efficient in case the function or operator is not inlined. The reference parameters have the `const` modifier in order to allow an expression rather than a variable as input.

It is, admittedly, not easier to make the code like example 11.4d than the previous examples because of the extra class definitions and operators when these are used only once. The idea is to put these definitions into a reusable header file. Once these definitions are made and tested, the only thing that has to be done to make vectorized code is to write the code like the `SelectAndAdd` function in example 11.4d.

The three versions 11.4b, c and d generate almost exactly the same code and they are equally efficient. It may come as a surprise that the complicated syntax with inheritance in many levels doesn't generate a complicated code. We can thank the compiler for that. The Intel compiler does an excellent job of optimizing this code, and so does the Gnu compiler. The functions, operators and constructors are all inlined and the references and parameter transfer overheads eliminated. Furthermore, it uses the methods of loop-invariant code motion. The ability of the compilers to do constant propagation on vector code is very limited, however.

Table 11.2 below shows measured execution times for the loop in example 11.4a, b and c on different microprocessors. The execution time for example 11.4a depends on the predictability of the `bb[i] > 0` branch. The values listed as best case apply when the branch always goes the same way. The values listed as worst case apply when the branch goes randomly one way or the other. The timing results for example 11.4b and c do not depend on branch predictability.

Code example	Branch predicta-bility	Time per element AMD64	Time per element Pentium 4	Time per element Pentium M	Time per element Core 2
11.4a	best case	8.0	4.6	9.0	8.0
11.4a	worst case	9.2	9.8	10.9	10.4
11.4b	worst case	1.0	1.6	1.3	0.5
11.4c	worst case	1.0	1.6	1.3	0.7

Table 11.2. Loop execution times for example 11.4 for different microprocessors. Clock cycles per element.

Defining missing operators

It can be quite complicated to do operations that are not supported by the vector class operators or intrinsic functions. For example, there is no function for multiplying vectors of 32 bit integers. There is a 32 x 32 → 64 bit multiply operation that we can use, but it requires a lot of data shuffling to get the data into the right places:

```
// Example 11.5
static inline Is32vec4 operator * (Is32vec4 const & a,
Is32vec4 const & b) {
   __m128i a13, b13, prod02, prod13, prod01, prod23, prod0123;
   a13 = _mm_shuffle_epi32(a, 0xF5);              // (-,a3,-,a1)
   b13 = _mm_shuffle_epi32(b, 0xF5);              // (-,b3,-,b1)
   prod02 = _mm_mul_epu32(a, b);                  // (-,a2*b2,-,a0*b0)
   prod13 = _mm_mul_epu32(a13, b13);              // (-,a3*b3,-,a1*b1)
   prod01 = _mm_unpacklo_epi32(prod02,prod13);    // (-,-,a1*b1,a0*b0)
   prod23 = _mm_unpackhi_epi32(prod02,prod13);    // (-,-,a3*b3,a2*b2)
   prod0123 = _mm_unpacklo_epi64(prod01,prod23); // (ab3,ab2,ab1,ab0)
   return prod0123;
}
```

This, of course, takes much longer time than 16 bit multiplications, which have intrinsic support. Multiplications by a power of 2 can be done by shifting.

There is no way to do integer division in vectors. Integer division by a constant can sometimes be done by multiplication and shifting. Otherwise, you have to convert to floating point, as example 11.6 below shows, or do the entire calculation with floating point vectors.

Mixed vector types

The following examples show how to vectorize a loop with mixed types:

```
// Example 11.6a
const int size = 128;  // size of arrays

void AddAndDivide(int aa[], int bb[], int cc[], float ff[]) {
      for (int i = 0; i < size; i++) {
         aa[i] = (bb[i] + cc[i]) / ff[i];
   }
}
```

Example 11.6a is not vectorized automatically by the compiler. We can use the same method as in example 11.4d to get the following code:

```
// Example 11.6b
// SSE2 instruction set must be enabled
#include <dvec.h>
const int size = 128;  // size of arrays

// Redefine class for vector of four 32-bit integers:
```

```
class Int32Vector4 : public Is32vec4 {
   public:
   // Default constructor:
   Int32Vector4() {};
   // Constructor to convert from parent type:
   Int32Vector4(Is32vec4 const & x) : Is32vec4(x) { }
   // Constructor to convert from type __m128i used in intrinsics:
   Int32Vector4(__m128i const & x) : Is32vec4(x) { }
   // Constructor to broadcast integer into all elements of vector:
   Int32Vector4(int const n) {*this = _mm_set1_epi32(n);}
   // Constructor to load four values from array:
   Int32Vector4(int const * p) {*this = *(Int32Vector4 const*)p;}
   // Member function to store four values into array:
   void store(int * p) {*(Int32Vector4*)p = *this;}
};

// Redefine class for vector of four 32-bit floats:
class FloatVector4 : public F32vec4 {
   public:
   // Default constructor:
   FloatVector4() {};
   // Constructor to convert from parent type:
   FloatVector4(F32vec4 const & x) : F32vec4(x) { }
   // Constructor to convert from type __m128 used in intrinsics:
   FloatVector4(__m128 const & x) : F32vec4(x) { }
   // Constructor to broadcast float into all elements of vector:
   FloatVector4(float const x) : F32vec4(x) { }
   // Constructor to load four values from array:
   FloatVector4(float const * p) {*this = *(FloatVector4 const*)p;}
   // Constructor to convert from vector of integers:
   FloatVector4(Int32Vector4 const & x) {*this = _mm_cvtepi32_ps(x);}
   // Member function to store four values into array:
   void store(float * p) {*(FloatVector4*)p = *this;}
};

// Function to convert vector of floats to vector of integers,
// using rounding:
static inline Int32Vector4 round(FloatVector4 const & x) {
   return _mm_cvtps_epi32(x);
}

// Function to convert vector of floats to vector of integers,
// using truncation:
static inline Int32Vector4 truncate(FloatVector4 const & x) {
   return _mm_cvttps_epi32(x);}

// Vectorized function:
void AddAndDivide(int aa[], int bb[], int cc[], float ff[]) {
   Int32Vector4 a, b, c;
   FloatVector4 f, t;

   // Roll out loop by four to fit the four-element vectors:
   for (int i = 0; i < size; i += 4) {
      b = bb + i;            // Load four values from bb
      c = cc + i;            // Load four values from cc
      f = ff + i;            // Load four values from ff
      t = FloatVector4(b + c); // Add integers, convert to floats
      t = t / f;             // Divide by floats
      a = truncate(t);       // Convert back to integers
      a.store(aa + i);       // Store four values in aa
   }
}
```

Mixing vectors with different number of elements

Vectorization of loops becomes more difficult when mixing different types of vectors with different numbers of elements. If the integers in example 11.6b are changed to 16-bit integers then we have integer vectors with eight elements and floating point vectors with four elements. The loop must be rolled out by eight and make two floating point vectors for every integer vector:

```
// Example 11.6c
// SSE2 instruction set must be enabled
#include <dvec.h>
const int size = 128;  // Size of arrays

class Int32Vector4; // Define as in example 11.6b

class Int16Vector8; // Define as in example 11.4d

class FloatVector4; // Define as in example 11.6b

// Define as in example 11.6b:
static inline Int32Vector4 truncate(FloatVector4 const & x);

// Unpack a vector of eight 16-bit integers into two vectors of
// four 32-bit integers each, using sign-extension:
static inline void unpack (Int32Vector4 & uLo, Int32Vector4 & uHi,
Int16Vector8 const & pk) {
   __m128i aLo, aHi;
   aLo = _mm_unpacklo_epi16(pk, pk);  // Unpack low  four integers
   aHi = _mm_unpackhi_epi16(pk, pk);  // Unpack high four integers
   uLo = _mm_srai_epi32(aLo, 16);     // Sign-extend to 32 bits
   uHi = _mm_srai_epi32(aHi, 16);     // Sign-extend to 32 bits
}

// Pack two vectors of four 32-bit integers each into one
// vector of eight 16-bit integers using signed saturation:
static inline void pack (Int16Vector8 & pk, Int32Vector4 const & uLo,
Int32Vector4 const & uHi) {
   pk = _mm_packs_epi32(uLo, uHi);
}

void AddAndDivide (short int *aa, short int *bb, short int *cc,
float *ff) {
   // Define temporary vectors:
   Int16Vector8 a, b, c;          // Vectors of eight 16-bit integers
   Int32Vector4 p1, p2;           // Vectors of four 32-bit integers
   FloatVector4 f1, f2, t1, t2;   // Vectors of four floats

   // Roll out loop by eight to fit the vector with most elements
   for (int i = 0; i < size; i += 8) {
      b = bb + i;                 // Load eight values from bb
      c = cc + i;                 // Load eight values from cc
      f1 = ff + i;                // Load first four values from ff
      f2 = ff + i + 4;            // Load next  four values from ff
      a = b + c;                  // Add eight integers
      unpack (p1, p2, a);         // Unpack into 2 times 4 integers
      t1 = FloatVector4(p1) / f1; // Divide with first 4 floats
      t2 = FloatVector4(p2) / f2; // Divide with next  4 floats
      p1 = truncate(t1);          // Convert first 4 back to int
      p2 = truncate(t2);          // Convert next  4 back to int
      pack (a, p1, p2);           // Pack 2 times 4 int into 8 int
      a.store(aa + i);            // Store eight values into aa
   }
}
```

Example 11.6c does not give the same result as example 11.6b in case of overflow because it uses saturation. Example 11.6c is less efficient than example 11.6b because of the extra time needed for packing and unpacking the vectors with different numbers of elements. Mixing vectors with different element sizes should be avoided if possible.

Making your own vector classes

In the above examples, I have defined my own vector classes by inheriting from the vector classes that are already defined in `dvec.h` in order to inherit the functions and operators that are already defined. In some cases it is preferable to make our own vector classes from scratch rather than basing our classes on inheritance from predefined classes. This can be useful if you don't have the Intel header files, if you don't want to use them, or if you want the operators to work differently.

It is a bad idea to inherit from the predefined vector classes if the algebra for your vectors is different from the simple element-by-element operations defined in `dvec.h`. Let's consider the example of complex numbers. The product of two complex numbers is not computed just by multiplying the real parts and the imaginary parts. The formula is more complicated:

$$(a+ib)\cdot(c+id) = ((ac-bd)+i(ad+bc))$$

The formula for division is even more complicated. It is possible to define a class `complex` for complex numbers by inheriting from `F64vec2` (a vector of two `double`'s) and then redefining the `*` and `/` operators for class `complex`. The `+` and `-` operators do not have to be redefined. But there is a pitfall here. The expression `y = (a + b) * (c + d)` would use the `*` operator for the base class `F64vec2`, not the `*` operator for the derived class `complex`, because both parentheses have type `F64vec2`. We can avoid this by also redefining the `+` operator, but the method is still somewhat risky because there may be some other operator, such as `*=`, that we have forgotten to redefine. It is safer to define a new base class rather than redefining an existing class. This prevents the inadvertent use of operators that we have forgotten to redefine.

The following example shows how to define a vector class from scratch and define operators for complex number algebra.

```
// Example 11.7
#include <emmintrin.h>

// Define class complexd for vector of two doubles.
// The low element represents the real part and the high element
// represents the imaginary part of a complex number:

__declspec(align(16))       // must be aligned
class complexd {
   protected:
   __m128d vec;             // Vector of two double's
   public:

   // Default constructor:
   complexd() {};

   // Constructor to make from real and imaginary part:
   complexd(double const re, double const im) {
      vec = _mm_set_pd(im, re);}

   // Constructor to convert from real number. Set imag. part to 0:
   complexd(double const x) {
      vec = _mm_set_pd(0., x);}

   // Constructor to convert from type __m128d used in intrinsics:
```

```
   complexd(__m128d const & x) {vec = x;}

   // Operator to convert to __m128d used in intrinsics
   operator __m128d() const {return vec;}

   // Member function to extract real part:
   double Re() const {
      return *(double const*)&vec;}

   // Member function to extract Imaginary part:
   double Im() const {
      return *(double const*)&_mm_shuffle_pd(vec, vec, 1);}
};

// Define operators for class complexd:

// complex + real: add only to real part:
static inline complexd operator + (complexd const &a, double const b){
   return _mm_add_sd(a, _mm_load_sd(&b));}

// complex - real: subtract only from real part:
static inline complexd operator - (complexd const &a, double const b){
   return _mm_sub_sd(a, _mm_load_sd(&b));}

// complex * real: multiply both real and imaginary part:
static inline complexd operator * (complexd const &a, double const b){
   return _mm_mul_pd(a, _mm_set1_pd(b));}

// complex / real: multiply both real and imag. part by reciprocal b:
static inline complexd operator / (complexd const &a, double const b){
   return _mm_mul_pd(a, _mm_set1_pd(1. / b));}

// complex + complex: add both parts:
static inline complexd operator + (complexd const & a,
complexd const & b) {
   return _mm_add_pd(a, b);}

// complex - complex: subtract both parts:
static inline complexd operator - (complexd const & a,
complexd const & b) {
   return _mm_sub_pd(a, b);}

// complex * complex: (a.re*b.re-a.im*b.im, a.re*b.im+b.re*a.im).
// This version is for SSE2 instruction set. It is more efficient
// to use _mm_addsub_pd and _mm_hadd_pd if SSE3 is available.
static inline complexd operator * (complexd const & a,
complexd const & b) {
   __m128d a_flipped;                // (a.im,a.re)
   __m128d b_re;                     // (b.re,b.re)
   __m128d b_im;                     // (-b.im,b.im)
   static const union {              // (signbit,0)
      int i[4];  __m128d v;
   } signbitlow = {0,0x80000000,0,0};

   b_im = _mm_shuffle_pd(b,b,3);     // Imag. part of b in both
   b_re = _mm_shuffle_pd(b,b,0);     // Real  part of b in both
   a_flipped = _mm_shuffle_pd(a,a,1); // Swap real and imag parts of a
   b_im = _mm_xor_pd(b_im, signbitlow.v); // Change sign of low

   // Multiply and add:
   return (complexd)_mm_mul_pd(a, b_re) +
          (complexd)_mm_mul_pd(a_flipped, b_im);
}

// complex / complex:
```

```
// (a.re*b.re+a.im*b.im, b.re*a.im-a.re*b.im)/(b.re*b.re+b.im*b.im).
// This version is for SSE2 instruction set. It is more efficient
// to use _mm_addsub_pd and _mm_hadd_pd if SSE3 is available.
static inline complexd operator / (complexd const & a,
complexd const & b) {
   __m128d a_flipped;                 // (a.im,a.re)
   __m128d b_re;                      // (b.re,b.re)
   __m128d b_im;                      // (-b.im,b.im)
   __m128d ab_conj;                   // a * conjugate(b)
   __m128d abs_b_square;              // (b.re*b.re,b.im*b.im)
   double b_abs_square_recip;         // 1/(abs(b*b))

   abs_b_square = _mm_mul_pd(b, b);   // (b.re*b.re,b.im*b.im)
   // Reciprocal of horizontal add:
   b_abs_square_recip = 1. /
      *(double const*)& _mm_add_sd(abs_b_square,
      _mm_shuffle_pd(abs_b_square, abs_b_square, 1));

   // The following code is made as similar to the operator * as
   // possible, to enable common subexpression elimination in code
   // that contains both operator * and operator / with the same
   // second operand:
   static const union {               // (signbit,0)
      int i[4];  __m128d v;
   } signbitlow = {0,0x80000000,0,0};
   b_im = _mm_shuffle_pd(b,b,3);      // Imag. part of b in both
   b_re = _mm_shuffle_pd(b,b,0);      // Real  part of b in both
   a_flipped = _mm_shuffle_pd(a,a,1); // Swap real and imag parts of a
   b_im = _mm_xor_pd(b_im, signbitlow.v); // Change sign of re

   // Multiply and subtract:
   ab_conj = (complexd)_mm_mul_pd(a, b_re) -
             (complexd)_mm_mul_pd(a_flipped, b_im);
   // Multiply by b_abs_square_recip:
   return (complexd)ab_conj * b_abs_square_recip;
}

// - complex: (-a.re, -a.im):
static inline complexd operator - (complexd const & a) {
   static const union {               // (signbit,signbit)
      int i[4];  __m128d v;
   } signbits = {0,0x80000000,0,0x80000000};
   return _mm_xor_pd(a, signbits.v);  // Change sign of both elements
}

// complex conjugate: (a.re, -a.im)
static inline complexd operator ~ (complexd const & a) {
   static const union {               // (signbit,signbit)
      int i[4];  __m128d v;
   } signbithigh = {0,0,0,0x80000000};
   return _mm_xor_pd(a, signbithigh.v); // Change sign of imag. part
}


// Example of use:
// Second order complex polynomial:
complexd polynomial(complexd x) {
   const complexd a(1.,2.), b(3.,4.), c(5.,6.);
   return (a * x + b) * x + c;
}
```

Example 11.7 shows how to define your own vector class from scratch and make appropriate operators for it. The Intel and Gnu compilers optimize this code quite well. The code may be further improved by using SSE3 instructions, if available. (The SSE3

instructions `_mm_addsub_pd` and `_mm_movedup_pd` are designed specifically for complex math. `_mm_hadd_pd` is useful for horizontal addition).

Another question is whether it is optimal to use vector operations at all in this case. The `*` and `/` operators involve a lot of shuffling in order to get the data elements into the right places. An implementation without vectorization does not require these shuffle operations. My tests on a code that contains an equal number of additions and multiplications shows that the vectorized implementation of complex math is 22% faster than the non-vectorized code on a Pentium 4, but 30 - 40% slower than the non-vectorized code on Pentium M and AMD processors. This observation is typical indeed. The extra code for shuffling and putting the data into the right positions is so expensive that the advantage of vectorization may be lost. The advantage of vectorization is likely to be higher on Intel Core 2 and other future microprocessors.

11.3 Mathematical functions

Intel has supplied various function libraries for computing mathematical functions such as logarithms, exponential functions, trigonometric functions, etc. in vector operands. These function libraries are useful for vectorizing mathematical code. The following example is a piece of code that can benefit from vectorization.

```
// Example 11.8a
#include <math.h>

const int size = 128;
float a[size], b[size];

void Func() {
   for (int i = 0; i < size; i++) {
      a[i] = 0.5f * exp(b[i]);
   }
}
```

The exponential function is available in a vector version calculating four exponential function values in one function call. This function is found in the Intel library called "Short Vector Math Library" (SVML) which comes with the Intel C++ compiler. Unfortunately, a header file for this library is not supplied. The function prototypes have to be copied from the document cited below. The vectorized code looks like this:

```
// Example 11.8b
// Compile with SSE2 enabled.
// Link with Intel math library svml_dispmt.lib

// Define vector classes:
#include <dvec.h>

// Function prototype for exponential function on vector of 4 floats
// (Copied from Mike Stoner: "Integrating Fast Math Libraries for the
// Intel Pentium® 4 Processor"):
extern "C" __m128 vmlsExp4(__m128);

const int size = 128;
__declspec(align(16))
float a[size], b[size];

void Func() {
   // Define vector of 4 floats for temporary storage:
   F32vec4 temp;

   // Roll out loop by four to fit vectors of four floats:
   for (int i = 0; i < size; i += 4) {
```

```
        // Load four values from b into vector:
        temp = *(F32vec4 const *)&b[i];

        // Call exponential founction on vector and multiply by 0.5:
        temp = vmlsExp4(temp) * F32vec4(0.5f);

        // Store result in four consecutive elements of a:
        *(F32vec4 *)&a[i] = temp;
    }
}
```

The necessary function prototypes and description of the different math libraries can be found in: Mike Stoner: "Integrating Fast Math Libraries for the Intel Pentium® 4 Processor". www.intel.com, n. d.

The code in example 11.8a is automatically converted to 11.8b when compiled with the Intel compiler, but not when compiled with any other compiler. The explicit call to the SVML library is useful if another compiler is used or if the Intel compiler cannot vectorize the code automatically.

An alternative is to use the Intel Math kernel Library, which is available from www.intel.com. This library contains mathematical functions for large vectors. The following code shows how to use the Intel Math kernel Library:

```
// Example 11.8c
// Link with mkl_c.lib from Intel Math Kernel Library

// Obtain this header file from Intel Math Kernel Library:
#include <mkl_vml_functions.h>

// Obtain header file for vector classes from Intel C++ compiler:
#include <dvec.h>

const int size = 128;
__declspec(align(16))
float a[size], b[size];

void Func() {
   // Define vector of 4 floats for temporary storage:
   F32vec4 temp;

   // Intel Math Kernel library function vsExp calculates 'size'
   // exponentials. The loop is inside this library function:
   vsExp(size, b, a);

   // Multiply all elements in a by 0.5:
   for (int i = 0; i < size; i += 4) {

      // Multiply four elements by 0.5:
      *(F32vec4 *)&a[i] *= F32vec4(0.5f);
   }
}
```

The following table compares the measured computation time in clock cycles per element on a various CPU's for example 11.8a, b and c.

Code example	Compiler used	Time per element AMD64	Time per element Pentium 4	Time per element Pentium M	Time per element Core 2
11.8a	Microsoft	59	311	121	130
11.8a	Intel 9.1	20	45	33	53
11.8b	Intel 9.1	25	17	18	14
11.8c	Intel 9.1	16	12	14	15

Table 11.3. Loop execution times for example 11.8 for different microprocessors. Clock cycles per element.

These tests show that it is very advantageous to use the math libraries. The Math Kernel Library (example 11.8c) is a little faster than the Short Vector Math Library (example 11.8b) on some processors, but the Math Kernel Library has a much larger footprint in the code cache and a very large initialization routine, which is called the first time the library function is called. The initialization time is not included in the above time measurements. The performance of the Core 2 processor is lower than expected due to the fact that the Intel function libraries are not yet optimized for this processor and that the performance is limited by the predecoder rather than the execution units in this processor.

11.4 Conclusion

Vectorized code often contains a lot of extra instructions for converting the data to the right format and getting them into the right positions. The amount of extra data conversion and shuffling that is needed determines whether it is profitable to use vectorized code or not.

The code in example 11.7 is slower than non-vectorized code on Pentium M and AMD processors, but faster on P4 and Core 2 processors. The code in example 11.6b and 11.6c is faster than the non-vectorized code on all processors despite the extra data conversion, packing and unpacking. This is because the bottleneck here is not data conversion and packing, but division. Division is very time-consuming and there is a lot to save by doing division in single precision vectors. The code in example 11.8b and c benefit a lot from vectorization.

I will conclude this section by summing up the factors that decide whether vectorization is optimal or not.

Factors that make vectorization favorable:

- Small data types: `char`, `short int`, `float`.
- Similar operations on all data in large arrays.
- Array size divisible by vector size.
- Unpredictable branches that select between two simple expressions.
- Operations that are only available with vector operands: minimum, maximum, saturated addition, fast approximate reciprocal, fast approximate reciprocal square root, RGB color difference.
- Mathematical vector function libraries.
- Use Gnu or Intel compiler.
- Use the newest CPU's.

Factors that make vectorization less favorable:

- Large data types: `int`, `__int64`, `double`.
- Misaligned data.
- Extra data conversion, shuffling, packing, unpacking needed.
- Predictable branches that can skip large expressions when not selected.
- Compiler has insufficient information about pointer alignment and aliasing.

- Operations that are missing in the instruction set for the appropriate type of vector, such as integer division and 32-bit integer multiplication.

Vectorized code is more difficult for the programmer to make and therefore more error prone. The vectorized code should therefore preferably be put away in reusable and well-tested library modules and header files.

12 Make critical code in multiple versions for different CPU's

Microprocessor producers keep adding new instructions to the instruction set. These new instructions can make certain kinds of code execute faster. The most important addition to the instruction set is the vector operations mentioned in chapter 11.

A disadvantage of using the newest instruction set is that the compatibility with older microprocessors is lost. This dilemma can be solved very elegantly by making the most critical subroutines in multiple versions for different microprocessors. For example, you may want to make one version that takes advantage of the SSE2 instruction set and another version that is compatible with old microprocessors. Using the SSE or later instruction set requires that the operating system supports the use of XMM registers. There are still old versions of Linux and Windows in use that do not support XMM registers.

The program should automatically detect which instruction set is supported by the CPU and the operating system and choose the appropriate version of the subroutine that contains the critical innermost loops. The different versions of a critical subroutine can be compiled separately with different compiler options to enable or disable the new instruction sets.

The Intel compiler has a feature for making multiple versions of a function for different CPU's. Every time the function is called, a dispatch is made to the right version of the function. The automatic dispatching can be made for all suitable functions in a module by compiling the module with the option `/QaxB` or `-axB`. This will make multiple versions even of functions that are not critical. It is possible to do the dispatching only for speed-critical functions by using the directive `__declspec(cpu_dispatch(...))`. See the Intel C++ Compiler Documentation for details.

A disadvantage with the CPU dispatch mechanism in the Intel compiler is that a dispatch branch is executed every time the function is called. The extra time used for the dispatching may offset the advantage obtained by CPU-specific optimization. If a dispatched function calls another dispatched function then the dispatch branch of the latter is executed even though the CPU-type is already known at this place. This can be avoided by inlining the latter function. The mechanism in the Intel compiler has certain compatibility problems, which are mentioned below.

It may be better to do the CPU dispatching explicitly by calling a function that detects which instruction set is available and then branching to the appropriate version of the critical code. This makes the code compatible with multiple compilers and makes it possible to do the dispatching at a higher level than the critical innermost function in order to make the function call faster.

The availability of various instruction sets can be determined with system calls (e.g. `IsProcessorFeaturePresent` in Windows). Alternatively, you may use the CPU detection function that I have supplied in www.agner.org/optimize/asmlib.zip. The name of the function is `InstructionSet()`. This function checks both the CPU and the operating system for support of the different instructions sets. The following example shows how to use this function:

```
// Example 12.1
// Link with appropriate version of alib.

// Header file for InstructionSet() etc.:
#include "alib.h"

// Backwards compatible version of critical inner function:
void InnerFunctionLegacy();

// Optimized version of same function, using SSE2 instr. set:
void InnerFunctionSSE2();

// This function calls one of the versions of the
// critical inner function:
void OuterFunction() {
   int i;
   int instrset = InstructionSet();
   if (instrset >= 4) {
      // SSE2 instruction set supported
      for (i = 0; i < 1000; i++) {
         ...
         InnerFunctionSSE2();
      }
   }
   else {
      // Use legacy version to support old systems
      for (i = 0; i < 1000; i++) {
         ...
         InnerFunctionLegacy();
      }
   }
}
```

Here, the check of instruction set is moved out of the inner function and out of the loop in order to save time. The code must be linked with the function library "`alib`", which is available in different versions for different compilers and operating systems. The function `InnerFunctionSSE2` in example 12.1 should be placed in a module that is compiled with a compiler that supports the SSE2 instruction set. My recommendation is to use the Intel compiler with option `/QxB` or `-xB` to optimize for the SSE2 instruction set.

Using explicit CPU dispatching with the `InstructionSet()` function has the following advantages over the method that is built into the Intel compiler:

- The CPU dispatching can be made outside the innermost loop and the innermost function. The Intel method makes a dispatch branch in every function.

- Functions for the same CPU can be placed together in memory and separate from function versions for different CPU's in order to improve code caching. This is possible with the individual function dispatch method of the Intel compiler but not with the automatic dispatch method.

- The `InstructionSet()` function checks both CPU and operating system for support of XMM registers, while the Intel dispatch method checks only the CPU. The Intel dispatch method will therefore make the program crash on old operating systems that do not support XMM registers.

- A method for detecting operating system support of XMM registers is described in Intel application note "AP-900 Identifying Support for Streaming SIMD Extensions in the Processor and Operating System", 1999. The method described in the Intel note requires that exception handling is enabled and that the compiler and operating system supports catching invalid opcode exceptions. The `InstructionSet()` function does not depend on exception catching.

- The `InstructionSet()` function works with all compilers, all programming languages and all operating systems on 32-bit and 64-bit x86 platforms. This makes the code portable and independent of the compiler.

CPU dispatching is less important in 64-bit systems than in 32-bit systems because the SSE2 instruction set is available in all 64-bit systems. The SSE3 and SSE4 instruction sets are not always available in 64-bit processors, but these instruction sets have only few advantages over SSE2. (SSE3 has horizontal addition of floating point vectors and some instructions useful for complex numbers. SSE4 has horizontal addition of integer vectors).

You may want to compile an application for a 64-bit operating system in order to get the advantages mentioned on page 5. Unfortunately, it is not possible to mix 32-bit and 64-bit code in the same program. A 32-bit program can run in a 64-bit operating system, but not vice versa. The best solution may be to have both a 32-bit version and a 64-bit version of the program. The appropriate version of the program may be selected during the installation process or by an `.exe` file stub.

13 Specific optimization advices

13.1 Bounds checking

In C++, it is often necessary to check if an array index is out of range. This may typically look like this:

```
// Example 13.1a
const int size = 16; int i;
float list[size];
...
if (i < 0 || i >= size) {
   cout << "Error: Index out of range";
}
else {
   list[i] += 1.0f;
}
```

The two comparisons `i < 0` and `i >= size` can be replaced by a single comparison:

```
// Example 13.1b
if ((unsigned int)i >= (unsigned int)size) {
   cout << "Error: Index out of range";
}
else {
   list[i] += 1.0f;
}
```

A possible negative value of `i` will appear as a large positive number when `i` is interpreted as an unsigned integer and this will trigger the error condition. Replacing two comparisons by one makes the code faster because testing a condition is relatively expensive, while the type conversion generates no extra code at all.

This method can be extended to the general case where you want to check whether an integer is within a certain interval:

```
// Example 13.2a
const int min = 100, max = 110;  int i;
...
if (i >= min && i <= max) { ...
```

can be changed to:

```
// Example 13.2b
if ((unsigned int)(i - min) <= (unsigned int)(max - min)) { ...
```

There is an even faster way to limit the range of an integer if the length of the desired interval is a power of 2. Example:

```
// Example 13.3
float list[16]; int i;
...
list[i & 15] += 1.0f;
```

This needs a little explanation. The value of `i&15` is guaranteed to be in the interval from 0 to 15. If `i` is outside this interval, for example `i` = 18, then the `&` operator (bitwise and) will cut off the binary value of `i` to four bits, and the result will be 2. The result is the same as `i` modulo 16. This method is useful for preventing program errors in case the array index is out of range and we don't need an error message if it is. It is important to note that this method works only for powers of 2 (i.e. 2, 4, 8, 16, 32, 64, ...). We can make sure that a value is less than 2^n and not negative by AND'ing it with 2^n-1. The bitwise AND operation isolates the least significant n bits of the number and sets all other bits to zero.

13.2 Use lookup tables

Reading a value from a table of constants is very fast if the table is cached. Usually it takes only one or two clock cycles to read from a table in the level-1 cache. We can take advantage of this fact by replacing a function call with a table lookup if the function has only a limited number of possible inputs.

Let's take the integer factorial function (n!) as an example. The only allowed inputs are the integers from 0 to 12. Higher inputs give overflow and negative inputs give infinity. A typical implementation of the factorial function looks like this:

```
// Example 13.4a
int factorial (int n) {          // n!
   int i, f = 1;
   for (i = 2; i <= n; i++) f *= i;
   return f;
}
```

This calculation requires `n-1` multiplications, which can take quite a long time. It is more efficient to use a lookup table:

```
// Example 13.4b
int factorial (int n) {          // n!
   // Table of factorials:
   static const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
      5040, 40320, 362880, 3628800, 39916800, 479001600};
   if ((unsigned int)n < 13) {   // Bounds checking (see page 107)
      return FactorialTable[n];  // Table lookup
   }
   else {
```

```
        return 0;                  // return 0 if out of range
    }
}
```

This implementation uses a lookup table instead of calculating the value each time the function is called. I have added a bounds check on `n` here because the consequence of `n` being out of range is possibly more serious when `n` is an array index than when `n` is a loop count. The method of bounds checking is explained above on page 107.

The table should be declared `const` in order to enable constant propagation and other optimizations. In most cases it is also recommended to declare the table `static`. This makes sure that the table is initialized when the program is loaded rather than each time the function is called. You may declare the function `inline`.

Replacing a function with a lookup table is advantageous in most cases where the number of possible inputs is limited and there are no cache problems. It is not advantageous to use a lookup table if you expect the table to be evicted from the cache between each call, and the time it takes to calculate the function is less than the time it takes to reload the value from memory plus the costs to other parts of the program of occupying a cache line.

Storing something in static memory can cause caching problems because static data are likely to be scattered around at different memory addresses. If caching is a problem then it may be useful to copy the table from static memory to stack memory outside the innermost loop. This is done by declaring the table inside a function but outside the innermost loop and without the `static` keyword:

```
// Example 13.4c
void CriticalInnerFunction () {
   // Table of factorials:
   const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
      5040, 40320, 362880, 3628800, 39916800, 479001600};
   ...
   int i, a, b;
   // Critical innermost loop:
   for (i = 0; i < 1000; i++) {
      ...
      a = FactorialTable[b];
      ...
   }
}
```

The `FactorialTable` in example 13.4c is copied from static memory to the stack when `CriticalInnerFunction` is called. The compiler will store the table in static memory and insert a code that copies the table to stack memory at the start of the function. Copying the table takes extra time, of course, but this is permissible when it is outside the critical innermost loop. The loop will use the copy of the table that is stored in stack memory which is contiguous with other local variables and therefore likely to be cached more efficiently than static memory.

If you don't care to calculate the table values by hand and insert the values in the code then you may of course make the program do the calculations. The time it takes to calculate the table is not significant as long as it is done only once. One may argue that it is safer to calculate the table in the program than to type in the values because a typo in a hand-written table may go undetected.

The principle of table lookup can be used in any situation where a program chooses between two or more constants. For example, a branch that chooses between two constants can be replaced by a table with two entries. This may improve the performance if the branch is poorly predictable. For example:

```
// Example 13.5a
float a;  int b;
a = (b == 0) ? 1.0f : 2.5f;
```

If we assume that `b` is always 0 or 1 and that the value is poorly predictable, then it is advantageous to replace the branch by a table lookup:

```
// Example 13.5b
float a;  int b;
static const float OneOrTwo5[2] = {1.0f, 2.5f};
a = OneOrTwo5[b & 1];
```

Here, I have AND'ed `b` with `1` for the sake of security. `b & 1` is certain to have no other value than `0` or `1` (see page 108). This extra check on `b` can be omitted, of course, if the value of `b` is guaranteed to be `0` or `1`. Writing `a = OneOrTwo5[b!=0];` will also work, although slightly less efficiently. This method is inefficient, however, when `b` is a `float` or `double` because all the compilers I have tested implement `OneOrTwo5[b!=0]` as `OneOrTwo5[(b!=0) ? 1 : 0]` in this case so we don't get rid of the branch. It may seem illogical that the compiler uses a different implementation when `b` is floating point. The reason is, I guess, that compiler makers assume that floating point comparisons are more predictable than integer comparisons. The solution `a = 1.0f + b * 1.5f;` is efficient when `b` is a `float`, but not if `b` is an integer because the integer-to-float conversion takes more time than the table lookup.

Lookup tables are particular advantageous as replacements for `switch` statements because `switch` statements often suffer from poor branch prediction. Example:

```
// Example 13.6a
int n;
switch (n) {
case 0:
   printf("Alpha");  break;
case 1:
   printf("Beta");   break;
case 2:
   printf("Gamma");  break;
case 3:
   printf("Delta");  break;
}
```

This can be improved by using a lookup table:

```
// Example 13.6b
int n;
static char const * const Greek[4] = {
   "Alpha", "Beta", "Gamma", "Delta"
};
if ((unsigned int)n < 4) { // Check that index is not out of range
   printf(Greek[n]);
}
```

The declaration of the table has `const` twice because both the pointers and the texts they point to are constant.

13.3 Integer multiplication

Integer multiplication takes longer time than addition and subtraction (3 - 10 clock cycles). Optimizing compilers will often replace integer multiplication by a constant with a

combination of additions and shift operations. Multiplying by a power of 2 is faster than multiplying by other constants because it can be done as a shift operation. For example, `a * 16` is calculated as `a << 4`, and `a * 17` is calculated as `(a << 4) + a`.

You can take advantage of this by preferably using powers of 2 when multiplying with a constant. The compilers also have fast ways of multiplying by 3, 5 and 9.

Multiplications are done implicitly when calculating the address of an array element. In some cases this multiplication will be faster when the factor is a power of 2. Example:

```
// Example 13.7
const int rows = 10, columns = 8;
float matrix[rows][columns];
int i, j;
int order(int x);
...
for (i = 0; i < rows; i++) {
   j = order(i);
   matrix[j][0] = i;
}
```

Here, the address of `matrix[j][0]` is calculated internally as `(int)&matrix[0][0] + j * (columns * sizeof(float))`. Now, the factor to multiply `j` by is `(columns * sizeof(float)) = 8 * 4 = 32`. This is a power of 2, so the compiler can replace `j * 32` with `j << 5`. If `columns` had not been a power of 2 then the multiplication would take longer time. It can therefore be advantageous to make the number of columns in a matrix a power of 2 if the rows are accessed in a non-sequential order.

The same applies to an array of structure or class elements. The size of each object should preferably be a power of 2 if the objects are accessed in a non-sequential order. Example:

```
// Example 13.8
struct S1 {
   int a;
   int b;
   int c;
   int UnusedFiller;
};
int order(int x);
const int size = 100;
S1 list[size];  int i, j;
...
for (i = 0; i < size; i++) {
   j = order(i);
   list[j].a = list[j].b + list[j].c;
}
```

Here, we have inserted `UnusedFiller` in the structure to make sure its size is a power of 2 in order to make the address calculation faster.

The advantage of using powers of 2 applies only when elements are accessed in non-sequential order. If the code in example 13.7 and 13.8 is changed so that it has `i` instead of `j` as index then the compiler can see that the addresses are accessed in sequential order and it can calculate each address by adding a constant to the preceding one (see page 56). In this case it doesn't matter if the size is a power of 2 or not.

The advise of using powers of 2 does not apply to very big data structures. On the contrary, you should by all means avoid powers of 2 if a matrix is so big that caching becomes a

problem. If the number of columns in a matrix is a power of 2 and the matrix is bigger than the cache then you can get very expensive cache contentions, as explained on page 78.

13.4 Integer division

Integer division takes much longer time than addition, subtraction and multiplication (40 - 80 clock cycles for 32-bit integers).

Integer division by a power of 2 can be done with a shift operation, which is must faster.

Division by a constant is faster than division by a variable because optimizing compilers can compute a / b as a * (2^n / b) >> n with a suitable choice of n. The constant (2^n / b) is calculated in advance and the multiplication is done with an extended number of bits. The method is somewhat more complicated because various corrections for sign and rounding errors must be added. This method is described in more detail in manual 2: "Optimizing subroutines in assembly language". The method is faster if the dividend is unsigned.

The following guidelines can be used for improving code that contains integer division:

- Integer division by a constant is faster than division by a variable
- Integer division by a constant is faster if the constant is a power of 2
- Integer division by a constant is faster if the dividend is unsigned

Examples:

```
// Example 13.9
int a, b, c;
a = b / c;                // This is slow
a = b / 10;               // Division by a constant is faster
a = (unsigned int)b / 10; // Still faster if unsigned
a = b / 16;               // Faster if divisor is a power of 2
a = (unsigned int)b / 16; // Still faster if unsigned
```

The same rules apply to modulo calculations:

```
// Example 13.10
int a, b, c;
a = b % c;                // This is slow
a = b % 10;               // Modulo by a constant is faster
a = (unsigned int)b % 10; // Still faster if unsigned
a = b % 16;               // Faster if divisor is a power of 2
a = (unsigned int)b % 16; // Still faster if unsigned
```

You can take advantage of these guidelines by using a constant divisor that is a power of 2 if possible and by changing the dividend to unsigned if you are sure that it will not be negative.

Division of a loop counter by a constant can be avoided by rolling out the loop by the same constant. Example:

```
// Example 13.11a
int list[300];
int i;
for (i = 0;  i < 300;  i++) {
   list[i] += i / 3;
}
```

This can be replaced with:

```
// Example 13.11b
int list[300];
int i, i_div_3;
for (i = i_div_3 = 0;  i < 300;  i += 3, i_div_3++) {
   list[i]   += i_div_3;
   list[i+1] += i_div_3;
   list[i+2] += i_div_3;
}
```

A similar method can be used to avoid modulo operations:

```
// Example 13.12a
int list[300];
int i;
for (i = 0;  i < 300;  i++) {
   list[i] = i % 3;
}
```

This can be replaced with:

```
// Example 13.12b
int list[300];
int i;
for (i = 0;  i < 300;  i += 3) {
   list[i]   = 0;
   list[i+1] = 1;
   list[i+2] = 2;
}
```

The loop unrolling in example 13.11b and 13.12b works only if the loop count is divisible by the unroll factor. If not, then you must do the extra operations outside the loop:

```
// Example 13.12c
int list[301];
int i;
for (i = 0;  i < 301;  i += 3) {
   list[i]   = 0;
   list[i+1] = 1;
   list[i+2] = 2;
}
list[300] = 0;
```

13.5 Floating point division

Floating point division takes much longer time than addition, subtraction and multiplication (20 - 45 clock cycles).

Floating point division by a constant should be done by multiplying with the reciprocal:

```
// Example 13.13a
double a, b;
a = b / 1.2345;
```

Change this to:

```
// Example 13.13b
double a, b;
a = b * (1. / 1.2345);
```

The compiler will calculate `(1./1.2345)` at compile time and insert the reciprocal in the code, so you will never spend time doing the division. Some compilers will replace the code in example 13.13a with 13.13b automatically but only if certain options are set to relax floating point precision (see page 59). It is therefore more safe to do this optimization explicitly.

Divisions can sometimes be eliminated completely. For example:

```
// Example 13.14a
if (a > b / c)
```

can sometimes be replaced by

```
// Example 13.14b
if (a * c > b)
```

But beware of the pitfalls here: The inequality sign must be reversed if `c` < 0. The division is inexact if `b` and `c` are integers, while the multiplication is exact.

Multiple divisions can be combined. For example:

```
// Example 13.15a
double y, a1, a2, b1, b2;
y = a1/b1 + a2/b2;
```

Here we can eliminate one division by making a common denominator:

```
// Example 13.15b
double y, a1, a2, b1, b2;
y = (a1*b2 + a2*b1) / (b1*b2);
```

The trick of using a common denominator can even be used on completely independent divisions. Example:

```
// Example 13.16a
double a1, a2, b1, b2, y1, y2;
y1 = a1 / b1;
y2 = a2 / b2;
```

This can be changed to:

```
// Example 13.16b
double a1, a2, b1, b2, y1, y2, reciprocal_divisor;
reciprocal_divisor = 1. / (b1 * b2);
y1 = a1 * b2 * reciprocal_divisor;
y2 = a2 * b1 * reciprocal_divisor;
```

13.6 Don't mix float and double

Floating point calculations usually take the same time regardless of whether you are using single precision or double precision, but there is a penalty for mixing single and double precision in programs compiled for 64-bit operating systems and programs compiled for the instruction set SSE2 or later. Example:

```
// Example 13.17a
float a, b;
a = b * 1.2;       // Mixing float and double is bad
```

The C++ standard specifies that all floating point constants are double precision by default, so `1.2` in this example is a double precision constant. It is therefore necessary to convert `b` from single precision to double precision before multiplying with the double precision constant and then convert the result back to single precision. These conversions take a lot of time. You can avoid the conversions and make the code up to 5 times faster either by making the constant single precision or by making `a` and `b` double precision:

```
// Example 13.17b
float a, b;
a = b * 1.2f;      // everything is float

// Example 13.17c
double a, b;
a = b * 1.2;       // everyting is double
```

There is no penalty for mixing different floating point precisions when the code is compiled for old processors without the SSE2 instruction set, but it may be preferable to keep the same precision in all operands in case the code is later ported to another platform.

13.7 Conversions between floating point numbers and integers

Conversion from floating point to integer

According to the standards for the C++ language, all conversions from floating point numbers to integers use truncation towards zero, rather than rounding. This is unfortunate because truncation takes much longer time than rounding unless the SSE2 instruction set is used. A typical conversion time is 40 clock cycles. If you cannot avoid conversions from `float` or `double` to `int` in the critical part of the code, then you may improve efficiency by using rounding instead of truncation. This is approximately three times faster. The logic of the program may need modification to compensate for the difference between rounding and truncation.

It is beyond my comprehension why there is no rounding function in standard C++ libraries. The function below fills this need. The function rounds a floating point number to the nearest integer. If two integers are equally near then the even integer is returned. There is no check for overflow. This function is intended for 32-bit Windows and 32-bit Linux with Microsoft, Intel and Gnu compilers.

```
// Example 13.18
static inline int round (double const x) { // Round to nearest integer
   int n;
#if defined(__unix__) || defined(__GNUC__)
   // 32-bit Linux, Gnu/AT&T syntax:
   __asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );
#else
   // 32-bit Windows, Intel/MASM syntax:
   __asm fld qword ptr x;
   __asm fistp dword ptr n;
#endif
   return n;}
```

This code will work only on Intel/x86-compatible microprocessors. The round function is also available in the function library at www.agner.org/optimize/asmlib.zip.

The following example shows how to use the `round` function:

```
// Example 13.19
double d = 1.6;
int a, b;
```

```
a = (int)d;      // Truncation is slow. Value of a will be 1
b = round(d);    // Rounding is fast. Value of b will be 2
```

In 64-bit mode or when the SSE2 instruction set is enabled there is no difference in speed between rounding and truncation. The missing `round` function can be implemented as follows in 64-bit mode or when the SSE2 instruction set is enabled:

```
// Example 13.20.  // Only for SSE2 or x64
#include <emmintrin.h>

static inline int round (float const x) {
   return _mm_cvtss_si32(_mm_load_ss(&x));}

static inline int round (double const x) {
   return _mm_cvtsd_si32(_mm_load_sd(&x));}
```

The code in example 13.20 is faster than other methods of rounding, but neither faster nor slower than truncation when the SSE2 instruction set is enabled.

Conversion from integer to floating point

Conversion of integers to floating point is faster than from floating point to integer. The conversion time is typically between 5 and 20 clock cycles. It may in some cases be advantageous to do simple integer calculations in floating point variables in order to avoid conversions from integer to floating point.

Conversion of unsigned integers to floating point numbers is less efficient than signed integers. It is more efficient to convert unsigned integers to signed integers before conversion to floating point if the conversion to signed integer doesn't cause overflow. Example:

```
// Example 13.21a
unsigned int u;  double d;
d = u;
```

If you are certain that `u` < 2^{31} then convert it to signed before converting to floating point:

```
// Example 13.21b
unsigned int u;  double d;
d = (double)(signed int)u;
```

13.8 Using integer operations for manipulating floating point variables

Floating point numbers are stored in a binary representation according to the IEEE standard 754 (1985). This standard is used in almost all modern microprocessors and operating systems (but not in some very old DOS compilers).

The representation of `float`, `double` and `long double` reflects the floating point value written as $\pm 2^{eee} \cdot 1.fffff$, where $\pm$ is the sign, `eee` is the exponent, and `fffff` is the binary decimals of the fraction. The sign is stored as a single bit which is 0 for positive and 1 for negative numbers. The exponent is stored as a biased binary integer, and the fraction is stored as the binary digits. The exponent is always normalized, if possible, so that the value before the decimal point is 1. This '1' is not included in the representation, except in the `long double` format. The formats can be expressed as follows:

```
struct Sfloat {
   unsigned int fraction : 23; // fractional part
   unsigned int exponent :  8; // exponent + 0x7F
   unsigned int sign     :  1; // sign bit
};
```

```
struct Sdouble {
   unsigned int fraction : 52; // fractional part
   unsigned int exponent : 11; // exponent + 0x3FF
   unsigned int sign     :  1; // sign bit
};

struct Slongdouble {
   unsigned int fraction : 63; // fractional part
   unsigned int one      :  1; // always 1 if nonzero and normal
   unsigned int exponent : 15; // exponent + 0x3FFF
   unsigned int sign     :  1; // sign bit
};
```

The values of nonzero floating point numbers can be calculated as follows:

$$floatvalue = (-1)^{sign} \cdot 2^{exponent-127} \cdot \left(1 + fraction \cdot 2^{-23}\right),$$
$$doublevalue = (-1)^{sign} \cdot 2^{exponent-1023} \cdot \left(1 + fraction \cdot 2^{-52}\right),$$
$$longdoublevalue = (-1)^{sign} \cdot 2^{exponent-16383} \cdot \left(one + fraction \cdot 2^{-63}\right).$$

The value is zero if all bits except the sign bit are zero. Zero can be represented with or without the sign bit.

The fact that the floating point format is standardized allows us to manipulate the different parts of the floating point representation directly with the use of integer operations. This can be an advantage because integer operations are faster than floating point operations. You should use such methods only if you are sure you know what you are doing. See the end of this section for some caveats.

We can change the sign of a floating point number simply by inverting the sign bit:

```
// Example 13.22
float f;
*(int*)&f ^= 0x80000000; // flip sign bit
```

We can take the absolute value by setting the sign bit to zero:

```
// Example 13.23
float f;
*(int*)&f &= 0x7FFFFFFF; // set sign bit to zero
```

We can check if a floating point number is zero by testing all bits except the sign bit:

```
// Example 13.24
float f;
if (*(int*)&f & 0x7FFFFFFF) { // test bits 0 - 30
   // f is nonzero
}
else {
   // f is zero
}
```

We can multiply a nonzero floating point number by 2^n by adding n to the exponent:

```
// Example 13.25
float f;  int n;
if (*(int*)&f & 0x7FFFFFFF) { // check if nonzero
   *(int*)&f += n << 23;      // add n to exponent
}
```

Example 13.25 does not check for overflow and works only for positive n. You can divide by 2^n by subtracting n from the exponent if there is no risk of underflow.

The fact that the representation of the exponent is biased allows us to compare two positive floating point numbers simply by comparing them as integers:

```
// Example 13.26
float a, b;
if (*(int*)&a > *(int*)&b) {
   // a > b if both positive
}
```

Example 13.26 assumes that we know that a and b are both positive. It will fail if both are negative or if a is 0 and b is -0 (zero with sign bit set).

We can shift out the sign bit to compare absolute values:

```
// Example 13.27
float a, b;
if (*(unsigned int*)&a * 2 > *(unsigned int*)&b * 2) {
   // abs(a) > abs(b)
}
```

The multiplication by 2 in example 13.27 will shift out the sign bit so that the remaining bits represent a monotonically increasing function of the absolute value of the floating point number.

We can convert an integer in the interval $0 <= i < 2^{23}$ to a floating point number in the interval $1.0 <= x < 2.0$ by setting the fraction bits:

```
// Example 13.28
int n;  float x;
*(int*)&x = (n & 0x7FFFFF) | 0x3F800000; // Now 1.0 <= x < 2.0
```

This method is useful for random number generators. See www.agner.org/random/randomc.zip for examples of source code that uses this method.

In general, it is faster to access a floating point variable as an integer if it is stored in memory, but not if it is a register variable. The address operator (&) forces the variable to be stored in memory, at least temporarily. Using the methods in the above examples will therefore be a disadvantage if other nearby parts of the code could benefit from using registers for the same variables.

The above examples all use single precision. Using double precision in 32-bit systems gives rise to some extra complications. A double is represented with 64 bits, but 32-bit systems do not have inherent support for 64-bit integers. Many 32-bit systems allow you to define 64-bit integers, but they are in fact represented as two 32-bit integers, which is less efficient. You may use the upper 32 bits of a `double` which gives access to the sign bit, the exponent, and the most significant part of the fraction. For example, to change the sign of a double:

```
// Example 13.22b
double d;
*((int*)&d + 1) ^= 0x80000000;  // flip sign bit
```

We can make an approximate comparison of doubles by comparing bit 32-62. This can be useful for finding the numerically largest element in a matrix for use as pivot in a Gauss elimination. The method in example 13.27 can be implemented like this in a pivot search:

```
// Example 13.29
```

```
const int size = 100;
double a[size];
unsigned int absvalue, largest_abs = 0;
int i, largest_index = 0;
for (i = 0; i < size; i++) {
   // Get upper 32 bits of a[i] and shift out sign bit:
   absvalue = *((unsigned int*)&a[i] + 1) * 2;
   // Find numerically largest element (approximately):
   if (absvalue > largest_abs) {
      largest_abs = absvalue;
      largest_index = i;
   }
}
```

Example 13.29 finds the numerically largest element in an array, or approximately so. It may fail to distinguish elements with a relative difference less than 2^{-20}, but this is sufficiently accurate for the purpose of finding a suitable pivot element. The integer comparison is much faster than a floating point comparison.

There is a penalty for accessing part of a variable when the full variable is accessed shortly afterwards because the CPU becomes confused about the size of the variable. You shouldn't manipulate half of a `double` with 32-bit integer instructions and then access it as a `double` immediately afterwards. In 64-bit systems you may use 64-bit integer operations rather than 32-bit operations to manipulate doubles.

Another problem with accessing 32 bits of a 64-bit double is that it is not portable to systems with big-endian storage. Example 13.22b and 13.29 will therefore need modification if implemented on other platforms with big-endian storage. All x86 platforms (Windows, Linux, BSD, Intel-based Mac OS, etc.) have little-endian storage, but other systems may have big endian storage (e.g. PowerPC-based Mac OS).

13.9 Mathematical functions

The most common mathematical functions such as logarithms, exponential functions, trigonometric functions, etc. are implemented in hardware in the x86 CPU's. However, a software implementation is faster than the hardware implementation in most cases when the SSE2 instruction set is available. The Intel compiler uses the software implementation if the SSE2 instruction set is enabled. Most other compilers use the hardware implementation.

The advantage of using a software implementation rather than a hardware implementation of these functions is higher for single precision than for double precision and higher for Intel processors than for AMD processors. But the software implementation is faster than the hardware implementation in most cases, even for double precision on AMD processors.

You may use the Intel math function library with a different compiler by including the library `libmmt.lib` and the header file `mathimf.h` that come with the Intel C++ compiler. This library contains many useful mathematical functions. A lot of advanced mathematical functions are supplied in Intel's Math Kernel Library, available from www.intel.com. (See also page 102).

14 Testing speed

Testing the speed of a program is an important part of the optimization job. You have to check if your modifications actually increase the speed or not.

There are various profilers available which are useful for finding the hot spots and measuring the overall performance of a program. The profilers are not always accurate,

however, and it may be difficult to measure exactly what you want when the program spends most of its time waiting for user input or reading disk files.

If it is known where the hot spot is then it may be useful to isolate the hot spot and make measurements on this part of the code only. This can be done very accurately by using the so-called time stamp counter. This is a counter which measures the number of clock pulses since the CPU was started. The length of a clock cycle is the reciprocal of the clock frequency, as explained on page 11. If you read the value of the time stamp counter before and after executing a critical piece of code then you can get the exact time consumption as the difference between the two clock counts.

The value of the time stamp counter can be obtained with the function `ReadTSC` listed below in example 14.1. This code works only for compilers that support intrinsics. Alternatively, you can get `ReadTSC` as a library function available in www.agner.org/optimize/alib.zip.

```
// Example 14.1
#include <intrin.h>          // Or #include <ia32intrin.h> etc.

__int64 ReadTSC() {          // Returns time stamp counter
   int dummy[4];             // For unused returns
   __int64 clock;            // Time
   __cpuid(dummy, 0);        // Serialize
   clock = __rdtsc();        // Read time
   __cpuid(dummy, 0);        // Serialize again
   return clock;
}
```

You can use this function to measure the clock count before and after executing the critical code. A test setup may look like this:

```
// Example 14.2
// Link with appropriate version of alib

#include <stdio.h>
#include <alib.h>            // Use ReadTSC() from library alib..
                             // or from example 14.1
void CriticalFunction();     // This is the function we want to measure

const int NumberOfTests = 10;          // Number of times to test
int i; __int64 time1;
__int64 timediff[NumberOfTests];       // Time difference for each test
for (i = 0; i < NumberOfTests; i++) { // Repeat NumberOfTests times
   time1 = ReadTSC();                  // Time before test

   CriticalFunction();                 // Critical function to test

   timediff[i] = ReadTSC() - time1;    // (time after) - (time before)
}
printf("\nResults:");                  // Print heading
for (i = 0; i < NumberOfTests; i++) { // Loop to print out results
   printf("\n%2i  %10I64i", i, timediff[i]);
}
```

The code in example 14.2 calls the critical function ten times and stores the time consumption of each run in an array. The values are then output after the test loop. The time that is measured in this way includes the time it takes to call the `ReadTSC` function. You can subtract this value from the counts. It is measured simply by removing the call to `CriticalFunction` in example 14.2.

The measured time is interpreted in the following way. The first count is always higher than the subsequent counts. This is the time it takes to execute `CriticalFunction` when code and data are not cached. The subsequent counts give the execution time when code and data are cached as good as possible. The first count and the subsequent counts represent the "worst case" and "best case" values. Which of these two values is closest to the truth depends on whether `CriticalFunction` is called once or multiple times in the final program and whether there is other code that uses the cache in between the calls to `CriticalFunction`. If your optimization effort is concentrated on CPU efficiency then it is the "best case" counts that you should look at to see if a certain modification is profitable. On the other hand, if your optimization effort is concentrated on arranging data in order to improve cache efficiency, then you may also look at the "worst case" counts. In any event, the clock counts should be multiplied by the clock period and by the number of times `CriticalFunction` is called in a typical application to calculate the time delay that the end user is likely to experience.

Occasionally, the clock counts that you measure are much higher than normal. This happens when a task switch occurs during execution of `CriticalFunction`. You cannot avoid this in a protected operating system, but you can reduce the problem by increasing the thread priority before the test and setting the priority back to normal afterwards.

An alternative to the test setup in example 14.2 is to use one of the test tools that I have made available at www.agner.org/optimize/testp.zip. These test tools are based on the same principle as example 14.2, but they can give additional information about cache misses, misaligned memory references, branch mispredictions, floating point instructions, etc. This information is based on the performance monitor counters that are built into the CPU hardware. A test is performed by inserting the critical function in the test program and compiling it. My test tools are intended for testing small critical pieces of code, not for testing whole programs.

15 Some useful templates

Below are some useful container class templates which are particularly fast because they do not use dynamic memory allocation and have only the most necessary functionality.

15.1 Array with bounds checking

It is useful to make a general template class to encapsulate arrays with automatic checking that the index is within the allowed range. This overcomes the most important security problem in the C++ language.

```
// Example 15.1a. Template for safe array with bounds checking
template <class T, int N> class CSafeArray {
protected:
   T a[N];                     // Array with N elements of type T
public:
   CSafeArray() {              // Constructor
      memset(a, 0, sizeof(a)); // Initialize to zero
   }
   int size() {                // Return the size of the array
      return N;
   }
   T & operator[] (unsigned int i) {  // Safe [] array index operator
      if (i >= N) {
         // Index out of range. The next line provokes an error
         return *(T*)0;  // Return a null reference to provoke error
      }
      // No error
```

```
        return a[i];     // Return reference to a[i]
    }
};
```

An array using this template class is declared by specifying the type and size as template parameters, as example 15.1b below shows. It is accessed with a square bracket index, just as a normal array. The constructor sets all elements to zero. You may remove the `memset` line if you don't want this initialization, or if the type `T` is a class with a default constructor that does the necessary initialization. The compiler may report that `memset` is deprecated. This is because it can cause errors if the size parameter is wrong, but it is still the fastest way to set an array to zero. The `[]` operator will detect an error if the index is negative or too high. An error message is provoked here in a rather unconventional manner by returning a null reference. This is sure to provoke an error message in a protected operating system, and the error is easy to trace with a debugger. You may replace this line by any other form of error reporting. For example, in Windows, you may write `FatalAppExitA(0,"Array index out of range");` or better, make your own error message function.

The following example illustrates how to use `CSafeArray`:

```
// Example 15.1b
CSafeArray <float, 100> list;           // Make array of 100 floats
for (int i = 0; i < list.size(); i++) { // Loop through array
   cout << list[i] << endl;             // Output array element
}
```

15.2 FIFO list

The following example is a template for a simple First-In-First-Out queue using a circular buffer. It can be used for simple types as well as for structure and class objects. The maximum number of items that the list can hold must be specified as a constant in order to avoid dynamic memory allocation. This template is much faster than a linked list with dynamic memory allocations.

```
// Example 15.2a. Template for FIFO list
template <class OBJTYPE, int MAXSIZE>
class Cfifo {
   protected:
   OBJTYPE * head, * tail;   // Pointers to current head and tail
   int n;                    // Number of objects in list
   OBJTYPE list[MAXSIZE];    // Circular buffer
   public:

   Cfifo() {                 // Constructor
      head = tail = list;    // Initialize
      n = 0;}

   bool put(OBJTYPE const & x) { // Put object into list
      if (n >= MAXSIZE) {
         return false;}      // Return false if list full
      n++;                   // Increment count
      *head = x;             // Copy x to list
      if (++head >= list + MAXSIZE) { // Increment head pointer
         head = list;}       // Wrap around
      return true;}          // Return true if success

   OBJTYPE get() {           // Get object from list
      if (n <= 0) {
         // Error: list empty.
         // ... Put an error message here or return an empty object !
         exit(1);}
      n--;                   // Decrement count
```

```
      OBJTYPE * p = tail;     // Pointer to object
      if (++tail >= list + MAXSIZE) { // Increment tail pointer
         tail = list;}        // Wrap around
      return *p;}             // Return object

   int NumObjects() {         // Tell number of objects in list
      return n;}              // Return number of objects
};
```

To use this template, make an object of type `Cfifo<MyType,MySize>` where `MyType` is the type of objects to store in the list, and `MySize` is the maximum number of objects that the list can contain at the same time. `MyType` can be a simple type such as `int` or a structure or class. `MySize` must be an integer constant.

```
// Example 15.2b. Use of FIFO template
Cfifo <int,1000> MyFIFOList;          // Make list of max 1000 int
MyFIFOList.put(10);                   // Put 10 into the list
MyFIFOList.put(20);                   // Put 20 into the list
while (MyFIFOList.NumObjects() > 0) { // While list not empty
   cout << MyFIFOList.get() << " ";   // Get item from list and print
}                                     // Will print "10 20 "
```

15.3 LIFO list

The following example is a template for a simple Last-In-First-Out list using a linear buffer. It is used in exactly the same way as the FIFO template in example 15.2.

```
// Example 15.3. Template for LIFO list
template <class OBJTYPE, int MAXSIZE>
class Clifo {
   protected:
   OBJTYPE * top;             // Pointer to top of stack
   int n;                     // Number of objects in list
   OBJTYPE list[MAXSIZE];     // Data buffer
   public:

   Clifo() {                  // Constructor
      top = list;             // Initialize
      n = 0;}

   bool put(OBJTYPE const & x) { // Put object into list
      if (n >= MAXSIZE) {
         return false;}       // Return false if list full
      n++;                    // Increment count
      *(top++) = x;           // Copy x to list
      return true;}           // Return true if success

   OBJTYPE get() {            // Get object from list
      if (n <= 0) {
         // Error: list empty.
         // ... Put an error message here or return an empty object !
         exit(1);}
      n--;                    // Decrement count
      top--;                  // Decrement pointer
      return *top;}           // Return object

   int NumObjects() {         // Tell number of objects in list
      return n;}              // Return number of objects
};
```

15.4 Searchable list

The following template is a sorted list. The records in the list are kept sorted at all times. Adding and removing records is slower than in the FIFO and LIFO lists (example 15.2 and 15.3) because the list has to be reordered every time it is changed. But searching for a record is faster because it can use binary search. The sorted list is useful when you need a small list where searches in the list are more frequent than adding or removing records.

A sorted list is not the optimal solution for large lists. A hashed list is much faster. But a sorted list is simpler than a hashed list and therefore a faster solution for small lists. I will not provide the details for a hashed list here. You have to look in textbooks on data structures and algorithms for the theory of hashing. The hashed list should preferably allocate a reasonable size for each bucket before elements are added and have an efficient way of extending buckets if they become too small.

The template class below has three parameters, `OBJTYPE`, `KEYTYPE` and `MAXSIZE`. `OBJTYPE` is a class or structure that defines the type of records in the list. `OBJTYPE` must have a member function named `Key()` that returns the type `KEYTYPE` which is the type used for searching. `OBJTYPE` should not have any explicit copy constructor or destructor. `KEYTYPE` must be a type for which the operators `<` and `==` are defined. These operators are used for sorting and searching in the list. `MAXSIZE` is the maximum number of records in the list. The best performance is obtained when `OBJTYPE` is small. The size of `OBJTYPE` may preferably be a power of 2.

```
// Example 15.4a. Template for sorted list
#include <string.h>        // Header file for memcpy and memmove

template <class OBJTYPE, class KEYTYPE, int MAXSIZE>
class CSortedList {
protected:
   int n;                           // Number of objects in list
   OBJTYPE list[MAXSIZE];           // Storage buffer

   // This function is used internally for finding a record
   // or storage place. It returns an index to first entry with
   // key bigger than k.
   int BinSearch(KEYTYPE const k) {
      unsigned int a = 0;           // Start of search interval
      unsigned int b = n;           // End of search interval + 1
      unsigned int c = 0;           // Middle of search interval

      // Binary search loop:
      while (a < b) {
         c = (a + b) / 2;
         if (k < list[c].Key()) {
            b = c;}
         else {
            a = c + 1;}
      }
      return a;
   }
public:                             // Public member functions

   // Constructor
   CSortedList() {
      n = 0;}                       // Initialize

   // This function adds a new record to the list.
   // Returns true if success, false if the list is full.
   bool Put(OBJTYPE const & x);

   // This function removes a record from the list.
```

```
   // Returns false if no record with the specified key is found.
   bool Remove(KEYTYPE const key);

   // This function gets a record and removes it from the list.
   // A copy of the record is stored at destination before it
   // is removed from the list.
   // Returns false if no record with this key is found.
   bool GetAndRemove(OBJTYPE * destination, KEYTYPE const key);

   // This function searches for a record in the list.
   // It returns a pointer to the record if found,
   // or returns a NULL pointer if not found.
   // Remember to check if the pointer is NULL before using it.
   // If more than one record has the same key then the one that
   // was added last is found.
   OBJTYPE * Find(KEYTYPE const key);

   // This function gets the number of records in the list
   int NumRecords() {return n;}

   // This function gets a pointer to the first record in the list.
   // It is used for looping through the list.
   // The pointer does not point to valid data if the list is empty.
   OBJTYPE * IteratorBegin() {return list;}

   // This function gets a pointer to the end of the list.
   // It is used for looping through the list.
   // Note that IteratorEnd() does not point to a valid record.
   // The last valid record can be accessed as IteratorEnd()-1 if
   // the list is not empty.
   OBJTYPE * IteratorEnd() {return list + n;}
};

// Member function Put. Adds a record to the list:
template <class OBJTYPE, class KEYTYPE, int MAXSIZE>
bool CSortedList<OBJTYPE, KEYTYPE, MAXSIZE>::Put(OBJTYPE const & x) {
   if (n >= MAXSIZE) {
      return false;}                // List full
   int a = BinSearch(x.Key());      // Find correct storage place
   // Move all records from a and above one place up to make
   // space for the new record:
   if (n - a > 0) {
      memmove(list+a+1, list+a, (n-a) * sizeof(OBJTYPE));}
   // Copy record into the right place in list:
   list[a] = x;
   n++;                             // Increment count
   return true;                     // Success
}

// Member function Remove. Removes a record from the list.
template <class OBJTYPE, class KEYTYPE, int MAXSIZE>
bool CSortedList<OBJTYPE, KEYTYPE, MAXSIZE>::Remove(KEYTYPE const key)
{
   int a = BinSearch(key) - 1;      // Index to record
   // Check that 0 <= a < n and that key matches a record:
   if ((unsigned int)a < (unsigned int)n && key == list[a].Key()) {
      // Found. Remove record:
      n--;
      // Call destructor if any:
      // Uncomment the following line if OBJTYPE has a destructor !
      // list[a].~OBJTYPE();

      // Move the following records one place down:
      if (n - a > 0) {
         memcpy(list+a, list+a+1, (n-a) * sizeof(OBJTYPE));}
```

```
      return true;}                  // Success
   else {
      // Not found. Return false:
      return false;}
}

// Member function GetAndRemove.
// Retrieves a record and removes it from the list.
template <class OBJTYPE, class KEYTYPE, int MAXSIZE>
bool CSortedList<OBJTYPE, KEYTYPE, MAXSIZE>::GetAndRemove(OBJTYPE *
destination, KEYTYPE const key) {
   int a = BinSearch(key) - 1;       // Index to record
   // Check that 0 <= a < n and that key matches a record:
   if ((unsigned int)a < (unsigned int)n && key == list[a].Key()) {
      // Found. Copy record to destination before it is removed:
      * destination = list[a];
      // Remove record:
      n--;
      // Call destructor if any:
      // Uncomment the following line if OBJTYPE has a destructor !
      // list[a].~OBJTYPE();

      // Move the following records one place down:
      if (n - a > 0) {
         memcpy(list+a, list+a+1, (n-a) * sizeof(OBJTYPE));}
      return true;}                  // Success
   else {
      // Not found. Return false:
      return false;}
}

// Member function Find.
// Searches for a record with the specified key.
// Returns a pointer to the record if found,
// returns a NULL pointer if not found.
template <class OBJTYPE, class KEYTYPE, int MAXSIZE>
OBJTYPE * CSortedList<OBJTYPE, KEYTYPE, MAXSIZE> ::
Find(KEYTYPE const key) {
   int a = BinSearch(key) - 1;       // Index to record
   // Check that 0 <= a < n and that key matches a record:
   if ((unsigned int)a < (unsigned int)n && key == list[a].Key()) {
      // Found. Return pointer to record
      return list + a;}
   else {
      // Not found. Return NULL pointer
      return 0;}
}
```

The following example shows how to use this template to make a sorted list of names and addresses. The records are sorted by name.

```
// Example 15.4b

// The sorting key for our records is a pointer to a zero-terminated
// ASCII string. We must wrap this pointer into class CKey so that we
// can define the operators < and == for comparing strings:
class CKey {
protected:
   char const * s;                   // Zero-terminated ASCII string
public:
   CKey(char const * k) : s(k) { }  // Constructor

   // Define operators for comparing strings:
```

```
   // Operator < returns true if a comes before b:
   friend bool operator < (CKey const a, CKey const b) {
      // Use strcmp below for case sensitive searching,
      // Replace strcmp with _stricmp for case insensitive searching:
      return strcmp(a.s, b.s) < 0;
   }

   // Operator == returns true if strings a and b are identical:
   friend bool operator == (CKey const a, CKey const b) {
      // Use strcmp below for case sensitive searching,
      // Replace strcmp with _stricmp for case insensitive searching:
      return strcmp(a.s, b.s) == 0;
   }
};

// Define lengths of text strings in records:
const int NameLength    = 32;
const int AddressLength = 32;

// Define the structure of records in our list:
struct CRecord {
   char name[NameLength];        // Name. Zero-terminated ASCII string
   char address[AddressLength];  // Address. Zero-termin. ASCII string

   // Default constructor:
   CRecord() {
      // Make empty strings:
      name[0] = 0;  address[0] = 0;}

   // Construct from text strings:
   CRecord(char const * Name, char const * Address) {
      // Copy string, truncating if necessary:
      strncpy(name, Name, NameLength);
      // Make sure string is zero-terminated:
      name[NameLength-1] = 0;
      // Copy string, truncating if necessary:
      strncpy(address, Address, AddressLength);
      // Make sure string is zero-terminated:
      address[AddressLength-1] = 0;
   }

   // Template CSortedList requires a member function named Key() to
   // return an object of class CKey for sorting and searching:
   CKey Key() const {return name;}
};

// Main function. Test sorted list:
int main() {
   // Define a sorted list with records of type CRecord,
   // keys of type CKey, and a maximum of 1000 records:
   CSortedList<CRecord, CKey, 1000> list;

   // Put some silly test data into the list:
   list.Put(CRecord("Dennis", "Nowhere"));
   list.Put(CRecord("Elsie", "Elsewhere"));
   list.Put(CRecord("Brad", "There"));
   list.Put(CRecord("Celia", "Somewhere"));
   list.Put(CRecord("Anna", "Here"));

   // Output the number of records in the list:
   printf("\n\n%i records:", list.NumRecords());

   // Loop through the list and print all records.
   // The records will be printed in alphabetical order.
   // Use a pointer as iterator.
```

```
   CRecord * p;
   for (p = list.IteratorBegin(); p != list.IteratorEnd(); p++) {
      printf("\n%-32s %-32s", p->name, p->address);
   }

   // Try to search for a name in the list:
   char * NameToFind = "Celia";
   CRecord * found = list.Find(NameToFind);
   // Check that pointer found is not NULL:
   if (found) {
      // Found name. Print record:
      printf("\n\nFound %s %s", found->name, found->address);}
   else {
      // Name not found:
      printf("\n\n%s Not found", NameToFind);}

   // Try to retrieve a record and remove it from the list:
   CRecord Removed;  // Object to copy record into
   char * NameToRemove = "Anna"; // Name to search for
   // Call GetAndRemove:
   if (list.GetAndRemove(&Removed, NameToRemove)) {
      // Found and removed successfully
      printf("\n\nRemove and get %s %s",
         Removed.name, Removed.address);}
   else {
      // Not found
      printf("\n\nCan't remove %s", NameToRemove);}

   // Try to remove a record from the list without retrieving it:
   NameToRemove = "Dennis"; // Name to search for
   if (list.Remove(NameToRemove)) {
      // Found and deleted successfully
      printf("\n\nRemoved %s", NameToRemove);}
   else {
      // Not found
      printf("\n\nCan't remove %s", NameToRemove);}

   // Loop through the list and print the records that remain
   // after removing Anna and Dennis:
   printf("\n\n%i records:", list.NumRecords());
   for (p = list.IteratorBegin(); p != list.IteratorEnd(); p++) {
      printf("\n%-32s %-32s", p->name, p->address);
   }
   return 0;
}
```

16 Overview of compiler options

Table 16.1. Command line options relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Optimize for speed	`/O2 or /Ox`	`-O3`	`/O3`	`-O3`
Interprocedural optimization	`/Og`			
Whole program optimization	`/GL`	`--combine -fwhole-program`	`/Qipo`	`-ipo`
No exception handling	`/EHs-`			
No stack frame	`/Oy`	`-fomit-frame-pointer`		`-fomit-frame-pointer`
No runtime type identification (RTTI)	`/GR–`	`-fno-rtti`	`/GR-`	`-fno-rtti`
Assume no pointer aliasing	`/Oa`			`-fno-alias`
Non-strict floating point		`-ffast-math`	`/fp:fast /fp:fast=2`	`-fp-model fast, -fp-model fast=2`
Simple member pointers	`/vms`			
Fastcall functions	`/Gr`			
Function level linking (remove unreferenced functions)	`/Gy`	`-ffunction-sections`	`/Gy`	`-ffunction-sections`
SSE instruction set (128 bit float vectors)	`/arch:SSE`	`-msse`	`/arch:SSE`	`-msse`
SSE2 instruction set (128 vectors of integer or double)	`/arch:SSE2`	`-msse2`	`/arch:SSE2`	`-msse2`
SSE3 instruction set		`-msse3`	`/arch:SSE3`	`-msse3`
SSE4 instruction set				
Automatic CPU dispatch			`/QaxB, etc.`	`-axB, etc.`
Automatic vectorization			(requires no specific option)	
Automatic parallelization by multiple threads			`/Qparallel`	`-parallel`
Parallelization by OpenMP directives			`/Qopenmp`	`-openmp`
32 bit code		`-m32`		
64 bit code		`-m64`		
Static linking (multithreaded)	`/MT`	`-static`	`/MT`	`-static`
Generate assembly listing	`/FA`	`-S -masm=intel`	`/FA`	`-S`
Generate map file	`/Fm`			
Generate optimization report			`/Qopt-report`	`-opt-report`

Table 16.2. Compiler directives and keywords relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Align by 16	`__declspec(align(16))`	`__attribute((aligned(16)))`	`__declspec(align(16))`	`__attribute((aligned(16)))`
Assume pointer is aligned			`#pragma vector aligned`	`#pragma vector aligned`

Assume pointer not aliased	`#pragma optimize("a", on)`
	`__restrict`
		`__restrict`	`__declspec(noalias)`
			`__restrict`
			`#pragma ivdep`
				`__restrict`
				`#pragma ivdep`

Assume function is pure		`__attribute((const))`		`__attribute((const))`
Assume function does not throw exceptions	`throw()`	`throw()`	`throw()`	`throw()`
Assume function called only from same module	`static`	`static`	`static`	`static`
Assume member function called only from same module		`__attribute__((visibility("internal")))`		`__attribute__((visibility("internal")))`
Vectorize			`#pragma vector always`	`#pragma vector always`
Optimize function	`#pragma optimize(...)`			
Fastcall function	`__fastcall`	`__attribute((fastcall))`	`__fastcall`	
Noncached write			`#pragma vector nontemporal`	`#pragma vector nontemporal`

Table 16.3. Predefined macros

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Compiler identification	`_MSC_VER` and not `__INTEL_COMPILER`	`__GNUC__` and not `__INTEL_COMPILER`	`__INTEL_COMPILER`	`__INTEL_COMPILER`
16 bit platform	not `_WIN32`	n.a.	n.a.	n.a.
32 bit platform	not `_WIN64`		not `_WIN64`	
64 bit platform	`_WIN64`	`_LP64`	`_WIN64`	`_LP64`
Windows	`_WIN32`		`_WIN32`	

platform				
Linux platform	n.a.	`__unix__` `__linux__`		`__unix__` `__linux__`
x86 platform	`_M_IX86`		`_M_IX86`	
x86-64 platform	`_M_IX86` and `_WIN64`		`_M_X64`	`_M_X64`

17 Literature

Other manuals by Agner Fog

The present manual is number one in a series of five manuals. See page 3 for a list of titles.

Literature on code optimization

Intel: "IA-32 Intel Architecture Optimization Reference Manual". developer.intel.com.
Many advices on optimization of C++ and assembly code for Intel CPU's. New versions are produced regularly.

AMD: "Software Optimization Guide for AMD64 Processors". www.amd.com.
Advices on optimization of C++ and assembly code for AMD CPU's. New versions are produced regularly.

Intel: "Intel® C++ Compiler Documentation". Included with Intel C++ compiler, which is available from www.intel.com.
Manual on using the optimization features of Intel C++ compilers.

Wikipedia article on compiler optimization. en.wikipedia.org/wiki/Compiler_optimization.

OpenMP. www.openmp.org. Documentation of the OpenMP directives for parallel processing.

Stefan Goedecker and Adolfy Hoisie: "Performance Optimization of Numerically Intensive Codes", SIAM 2001.
Advanced book on optimization of C++ and Fortran code. The main focus is on mathematical applications with large data sets. Covers PC's, workstations and scientific vector processors.

Michael Abrash: "Zen of code optimization", Coriolis group books 1994.
Covers optimization of C and assembly code for the DOS platform. Mostly obsolete.

Rick Booth: "Inner Loops: A sourcebook for fast 32-bit software development", Addison-Wesley 1997.
Covers optimization of assembly code for Pentium - Pentium II. Mostly obsolete.

Microprocessor documentation

Intel: "IA-32 Intel Architecture Software Developer's Manual", Volume 1, 2A, 2B, and 3A and 3B. developer.intel.com.

AMD: "AMD64 Architecture Programmer's Manual", Volume 1 - 5. www.amd.com.

Literature on usability

Jenny Preece (ed): "Human-Computer interaction". Addison-Wesley 1994.
University-level textbook. Theoretical but easy to understand, with many illustrations and exercises.

Ben Shneiderman & Catherine Plaisant: "Designing the User Interface: Strategies for Effective Human-Computer Interaction". 4th ed. Addison Wesley 2004.
Comprehensive university-level textbook on the design of human/computer interface.

Ernest McCormick: "Human factor in engineering and design". McGraw-Hill 1976

Theoretical textbook about input/output devices, ergonomics, cognition, psychology.

W. M. Newman & M. G. Lamming: "Interactive System Design". Addison-Wesley 1995.
Textbook with the main focus on user psychology and cognition, including user study, modeling user activity, and systems analysis.

Jef Raskin: "The Humane Interface: New Directions for Designing Interactive Systems". Addison-Wesley 2000.
Theoretical book on human/computer interface and cognitive psychology with detailed discussion of commands, displays, cursors, icons, menus, etc.

W3C: "Web Content Accessibility Guidelines 1.0". 1999. www.w3.org/TR/WAI-WEBCONTENT/.
Guidelines for handicap-friendly web user interfaces. Some of the advices are also applicable to other software interfaces.

Internet forums

Several internet forums and newsgroups contain useful discussions about code optimization. See www.agner.org/optimize and the FAQ for the newsgroup comp.lang.asm.x86 for some links.