Designing and Building Portable Systems in C++

Günter Obiltschnig, Applied Informatics
guenter.obiltschnig@appinf.com

Abstract. C++ covers the whole range from low-level to high-level programming, making it ideally suited for writing portable software. However, code portability is often neglected in embedded systems engineering. With software becoming ever more complex, and hardware becoming ever more interchangeable, this oversight can turn into a problem when software must be ported to a new platform. This paper shows tools and techniques to design and build portable software in C++. It shows how to use C++ features to encapsulate platform-dependent parts (compiler/language differences, operating system interfaces, input/output) of programs, thus ensuring portability of the resulting system.

1 Introduction

Software for embedded systems is becoming ever more complex. At the same time, the significance of the hardware in embedded systems, at least when it comes to CPUs and controllers, is steadily decreasing. Custom hardware designs are replaced by off-the-shelf components and designs. The value of embedded systems is moving from the hardware to the software side.

1.1 The Significance of Software

Historically, software for embedded systems has been written specific to the underlying hardware. This is frequently still so today, especially in 8-bit and 16-bit systems, where the limits of the hardware restrict the size and complexity of the software. However, 32-bit (and more so, 64-bit) platforms allow for nearly unlimited complexity. The know-how and intellectual property hidden in embedded software grows enormously, and software starts to outgrow or outlive the underlying hardware platforms. Therein lies the need to design and implement software in a way that makes it easy to transfer it to a new hardware platform.

1.2 RTOS to the Rescue?

32-bit platforms mandate the use of an embedded/real-time operating system (RTOS) that provides services like task/process or thread scheduling and synchronization, memory management, input/output and inter-process communication to application software. An RTOS, to some degree, also decouples application software from the specifics of the underlying hardware, like I/O port addresses and hardware interrupts. A first step to designing portable software is to choose an operating system that is available for multiple hardware platforms. This provides the possibility to switch, relatively painlessly, to a more powerful hardware platform should the need ever arise. The respective RTOS has to be available for that platform, however. There is still the problem, of course, what happens if it is the RTOS

that does no longer suit the requirements. The move to a different RTOS can be a real pain if the application programming interfaces (APIs) of the operating systems differ. Unless the APIs of both operating systems conform to the POSIX specification, incompatible APIs will certainly be the norm and all code that interacts with the operating system needs to be rewritten. This is non-trivial and error-prone. Fortunately, many embedded operating systems already support POSIX (for example, QNX, Integrity, embedded Linux) or at least provide a wrapper library that provides POSIX APIs on top of the operating system's native API (for example, VxWorks). Still, there are major embedded operating systems (like Windows CE) that do not support POSIX. This issue will be discussed later in the paper.

1.3 The Programming Language Choice

C++ is slowly but steadily replacing C as the programming language of choice for embedded software development. While C++ has long been considered prohibitively resource intensive for embedded devices, with today's available powerful embedded hardware this is certainly no longer the case. For example, today's smart phones have roughly the same computing power (in terms of processor performance and memory capacity) as engineering workstations from 12 years ago, and 32-bit RISC microprocessors are state-of-the-art for embedded applications.

What makes C++ unique among programming languages is that it covers the full range from low-level programming (interrupt service routines, direct hardware access) to high-level programming (object orientation, generic programming) in one language, without the need to connect both ends via awkward artificial interfacing mechanisms. With the software for embedded systems becoming ever more sophisticated, the high level abstractions of C++ are certainly needed in addition to its suitability for low-level programming.

1.3.1 A Short History of C++

The history of C++ started in 1979, when Bjarne Stroustrup of Bell Labs started to extend the C programming language with object-oriented features [1]. Stroustrup, who at the time worked as an operating system researcher, was in search for a tool that would let him write simulators, operating systems or do systems programming in general. His wish list for such a tool included good support for program organization (i.e. classes), strong type checking and concurrency support. The programs produced by such a tool should run as fast as possible and it should be possible to combine separately compiled units, even those written in other languages, such as C or Fortran, into programs. Finally, the tool should allow for highly portable implementations. As Stroustrup could not find such a tool on the market, he decided to build his own. He also chose not to completely start from scratch and to build an all-new language, but to take an existing language as starting point. C was the language of choice for this purpose, because of its flexibility, efficiency, availability and portability.

1.3.2 The Early Days

As a superset of C, the new language provided "flexible and efficient facilities for defining new types", as Bjarne Stroustrup wrote in the first edition of The C++ Programming Language. Inspired by the object-oriented features of the Simula programming language, Stroustrup enhanced C with objects, classes, inheritance and, later, virtual functions, while still maintaining backwards compatibility with the old language. This first version of what would later become C++ was a quite simple, yet still powerful language called C with Classes. The language was implemented as a pre-processor for C, called Cpre.

1.3.3 Evolution

From 1982 to 1984, C with Classes gradually evolved into C++. The name C++, suggested by Rick Mascitti, first appeared in December 1983. Prior to this, the language was called C84 for a few months. A new compiler front-end, Cfront, was designed and implemented from 1982 to 1983.

The first commercial C++ compiler appeared in 1985. In the same year, the first edition of The C++ Programming Language was published. From the beginning on, the language was steadily improved and new features were being added. Cfront served as the reference implementation of C++, and the major releases corresponded to the new features added to the language. Release 2.0 (1989) brought multiple inheritance. Release 3.0 (1991) added templates and release 4.0 (scheduled for 1993) was supposed to add exception handling, but was never released due to the overwhelming complexity of implementing it in the existing code base.

1.3.4 Standardization

The standardization process for C++, driven by ANSI and ISO, began in 1991 and the final standard was published in September 1999, as ISO/IEC 14882. A major addition to the standard was the Standard Template Library (STL) in 1994. Developed by Alexander Stepanov and Meng Lee at Hewlett Packard Laboratories, the STL brought ready-made generic algorithms and container classes to C++. It also introduced the concept of generic programming to a large audience outside of universities and research laboratories.

During all these years, C++ has turned into a quite complex, but at the same time extremely powerful language. The complexity of C++ manifested itself best in the long time it took compiler vendors to bring standards-compliant (and largely bug free) compilers to market. When Andrei Alexandrescu released Modern C++ Design, in which he wrote about advanced uses of templates, in 2001, almost no commercially available compiler could handle the standard compliant code presented therein.

Porting source code that used advanced C++ features to different compilers was a painful experience for a long time. Finally, today, it is possible to write both standard compliant and portable C++ code for a large number of platforms.

2 C++ Cross-Platform Programming Issues

Despite being designed as a platform-neutral language, writing cross-platform software in C++ is a not-so-easy task, especially when it comes to functionality not covered by the C++ Standard Library. But even a few features (or lack thereof) of the language itself can be a source of problems.

2.1 Compiler Bugs and Implementation Issues

Among the major problems in writing portable C++ code are the dissimilarities in the implementation of different vendor's C++ compilers. While the C++ standard describes the language in great detail, there still seems to be some degree of misinterpretation in the C++ compiler-writer-community. Another source of grief lies in the sheer complexity of C++ and the trouble this causes to its implementers.

2.1.1 The Complexity of C++

Especially when it comes to templates, many compilers still have trouble handling C++ correctly. An early version of the DEC C++ (5.6) compiler for OpenVMS could not even

parse some of its own standard library header files. The automatic instantiation of templates in conjunction with shared libraries caused nightmares on many platforms. Nesting of templates, member templates, and partial specialization can cause problems even today. Furthermore, the implementations of the standard library that shipped with the various compilers had bugs on their own, up to a degree that made them practically unusable for use in real-world applications (the implementation that came with Microsoft Visual C++ 6 with its multithreading issues comes to mind here). A good alternative in such a case is STLport, an open-source, portable, high-quality implementation of the C++ standard library.

Life was — and even sometimes still is — quite hard for developers wanting to write both standards-compliant and portable C++ code. Fortunately, with the latest C++ compiler releases, things are finally getting better.

2.1.2 Implementation-Defined, Unspecified and Undefined Behavior

The C++ standard specifies the exact (required) behavior for most, but not for all language elements. Certain language constructs are described as *implementation-defined*, *unspecified* or *undefined*.

Implementation-defined means that the compiler writer is free to implement a certain language construct in any way he sees appropriate, as long as the exact behavior is consistent, documented, and the compilation succeeds. The standard may specify a number of allowable behaviors from which to choose one, or it may leave it entirely up to the compiler writer.

Unspecified is similar, except that the behavior of the implementation need not be documented and need not even be consistent.

Undefined behavior means that the standard does not place any requirements whatsoever on the implementation. The compiler may even fail when compiling such a language construct, or the program may silently produce incorrect results.

The rationale behind all this is that it allows the compiler writer to implement certain constructs in a way that works best for the target platform. The downside of all this is that it opens up opportunities for portability problems. Code that works as expected on one platform can produce different, unexpected results on another platform. Even more so, the behavior can be different depending on whether the compiler's optimizer has been used, or not.

An example for unspecified behavior is the order in which function arguments are evaluated in a function call. Consider the following example:

```
void f(int a, int b, int c)
{
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
}

int main(int argc, char** argv)
{
    int i = 0;
    f(i++, i++, i++);
    return 0;
}
```

This example will yield surprisingly different results on different platforms. For example, on Mac OS X, using GCC 3.3, the program fragment yields the following output

```
a = 0, b = 1, c = 2
```

This is what one would intuitively expect, assuming left-to-right argument evaluation. Compiled with HP ANSI C++ A.03.57 on HP-UX 11.11, the result is the same. However, when compiled with Compaq C++ 6.5 on HP Tru64 5.1, the program yields:

```
a = 0, b = 0, c = 0
```

which may be surprising. However, with regard to the C++ standard, both results are correct. Firstly, the compiler is free to evaluate the function arguments in any order it likes – from left to right, from right to left, in order of decreasing expression complexity, or even in random order. Secondly, the compiler is free to delay the increment operation until all arguments have been evaluated.

2.1.3 Language Extensions and Syntactical Freedom

Especially Microsoft Visual C++ has many "features" that make writing portable code unnecessarily hard for developers not paying enough attention to correct syntax. There is a compiler setting to turn off language extensions, but with this setting enabled, one can no longer use the Windows Platform SDK, as its header files rely on non-standard language features. The most common problem cases are presented below.

Method declarations

The following class definition is valid in Microsoft Visual C++, but causes errors on standards conformant compilers.

```
Class A
{
    void A::m();
};
```

In this example, the class name is part of the method declaration, which is both redundant and not allowed by the standard. Mistakes of this kind happen very easily if a class is extended with a new method. The developer starts by implementing the new method in the implementation file, and then adds the corresponding method declaration to the header file by copy-and-paste, forgetting to remove the class scope from the declaration. The code compiles fine with Microsoft Visual C++, but compilers that are more stringent will produce error messages when being given such code.

Pointers to members

Microsoft Visual C++ allows a simplified syntax for passing pointers to members functions. The following class declaration defines a pointer to a method of B taking no arguments, as well as methods using this pointer type.

```
Class B
{
public:
      typedef void (B::*M)(); // pointer to a method of B taking no
args
    void invoke(M m);
    void do();
    void test();
};
```

The methods `invoke()` and `do()` can be implemented as follows:

```
void B::invoke(B::M m)
{
    (this->*m)();
}
```

```
void B::do()
{
    // do something interesting here
}
```

Using Microsoft Visual C++ (with language extensions enabled), it is possible to implement `test()` in the following way, which does not seem to be wrong at first glance:

```
void B::test()
{
    invoke(do);
}
```

Well-behaved compilers will not accept this code, as the correct way to specify a pointer to a non-static member is `&Class::Method`. Therefore, for portable code, one needs to write:

```
void B::test()
{
    invoke(&B::do);
}
```

Platform-specific types

It can often be seen that developers use platform-specific or user-defined types where a standard type is more appropriate. Frequently this happens with the Boolean type. There were times when C and C++ did not have a built-in Boolean data type, and in these times it was common for every project to define its own Boolean type, usually called `BOOL`, along with its values `TRUE` and `FALSE` defined as preprocessor macros. Apparently, many developers got so used to these workarounds that they use them even today. Such kludges have no business in portable code. C++ has the `bool` type, so use it!

The situation is similar with certain platform-specific types (`DWORD`, `TCHAR`, `LPCTSTR`), which are often used in situations where they are inappropriate.

The last line

The C++ standard requires the last line in a C++ source file to be terminated by a newline character. This rule is often violated since many compilers do not enforce it. There are, however, compilers that do enforce this rule, and these compilers will produce warnings or errors when being fed non-conforming source files.

2.2 Fixed-Size Data Types

C++ does not make any guarantees about the size of fundamental types, except for vague requirements such as that a certain type must have at least the size of another type (for example, `int` must have at least the size of `short`) and that an `int` has the natural size suggested by the system architecture. So what can be done if a fixed-size data type (a 32-bit unsigned integer, for example) is needed in portable code? The solution is to put type definitions for all such fixed-sized types in a header file. The actual type definitions for a specific platform (or compiler), depend on the platform. Here comes the hard part. One has to dig out the reference manuals for all the compilers to be used, and look up the sizes for the fundamental types. The size of some types will be different depending on whether compiling for 32-bit mode or for 64-bit mode. What makes things even more difficult is that some compilers provide fundamental types over and above those defined in the C++ standard. Most compilers for 32-bit platforms also have 64-bit signed and unsigned integer types. Yet, on most 32-bit platforms, `int` and `long` are 32-bit types, so these compilers need non-standard

type names for their 64 bit integer types. Most compilers name these types `long long` and `unsigned long long`. Microsoft Visual C++ calls them `__int64` and `unsigned __int64`.

When working with character types, it must be kept in mind that `char`, `unsigned char` and `signed char` are all different types. This is different for integer types, where, for example, `int` and `signed int` are the same. Furthermore, the standard leaves it up to the implementation whether the plain `char` type is signed or unsigned.

The C++ standard also defines a distinct wide character type, `wchar_t`. However, its characteristics make it unsuitable for use in cross-platform code, as one cannot make any general assumptions about its size or the character encoding it supports.

If one absolutely must store pointer values in integer types, it must be kept in mind that on some 64-bit platforms a pointer might not fit into a `long` variable.

The ISO/IEC 988:1999 standard for C specifies a set of fixed-size integer types, defined in the standard header file `<stdint.h>`. These types are `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, minimum-width and fastest-width variants of them, as well as `intptr_t` and `uintptr_t` for holding pointer values. However, this header file is not part of standard C++, so those types are not generally available to C++ programs.

2.3 Byte Order

Integer values made up of more than one byte can be stored in two ways. If a 16-bit (2 byte) integer is taken as an example, it can be stored either with the low-order byte at the starting address, or with the high-order byte at the starting address. The first alternative is known as *little-endian* byte order, the second is known as *big-endian* byte order. Figure 1 illustrates these two formats.

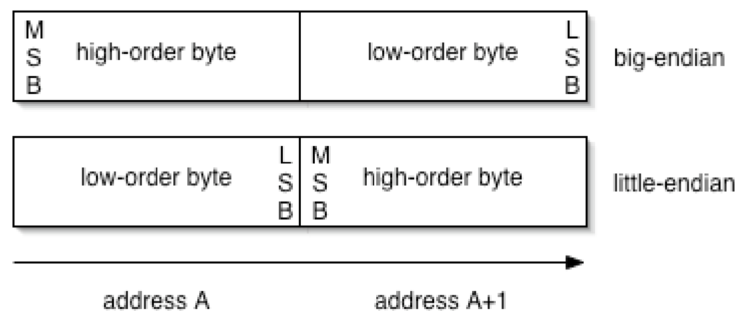

Figure 1. Little-endian vs. big-endian.

For big-endian, the higher-order bytes are stored on lower memory addresses and the lower-order bytes are stored on higher memory addresses. For little-endian, the opposite approach is used. In a multi-byte binary number, the leftmost bit, bit 0 in the lowest-order byte, is called the *least significant bit* (LSB). The rightmost bit, bit 7 in the highest-order byte, is called the *most significant bit* (MSB). Big-endian corresponds to the natural writing order for binary numbers and values are more easily readable in a memory dump. In little-endian, multi-byte values look somewhat awkward in a memory dump, as can be seen by the position of the MSB and LSB in Figure 1. In the example a 16-bit integer is shown, but the same concept of course applies to 32-bit and 64-bit integers as well.

There is no standard byte order scheme, so different systems use different byte orders, and this leads to potential problems when binary data (via a file or a network connection) is exchanged between systems. The inventors of the Internet protocols faced the same problem and they solved it by specifying a network byte order, which all protocol data sent over the

network must obey. Network byte order is defined to be big-endian, and every operating system supporting the Internet protocols provides functions for converting values from host byte order to network byte order and back.

2.4 Data Alignment

Data alignment violations are a frequent source of problems when porting code from one platform to another. Let's take as an example the following imaginary piece of code, which would work fine under Windows NT on Intel hardware.

```
struct Header
{
    UInt32 size;
    UInt32 checksum;
};
...
void handleData(void* pData)
{
    Header* pHeader = reinterpret_cast<Header*>(pData);
    for (int i = 0; i < pHeader->size; ++i)
    ...
}
```

What is wrong with this code? Well, the trouble starts when a program using this code is compiled and run on, say, Sparc Solaris. One may suddenly get a nasty *Bus error*. Upon further examination of the cause, one would get this bus error whenever the address `pData` points to is not on a four-byte boundary. Or, in other words, the address is not properly aligned. On most machines, a memory object of size s bytes must be located on a memory address `A` such as that `A mod s = 0`.

What is the reason for the alignment requirement just described? Accessing values at misaligned addresses causes hardware complications, since memory is typically aligned on word boundaries. To reduce hardware complexity, many modern architectures do not provide the additional logic required to support such addresses. On these architectures, an attempt to access misaligned data will result in a hardware trap, which causes the bus error one would experience on Solaris. To support misaligned memory access, the hardware must translate every misaligned access into two separate, aligned memory accesses. The result of these two operations must then be combined to yield the desired result, as depicted in Figure 2. Instead of implementing misaligned accesses directly in hardware, system designers can implement such accesses in software by providing a handler routine for the hardware trap, which then carries out the necessary operations. Regardless of the approach taken, every misaligned memory access, even if supported by hardware, is many times slower than an aligned access.

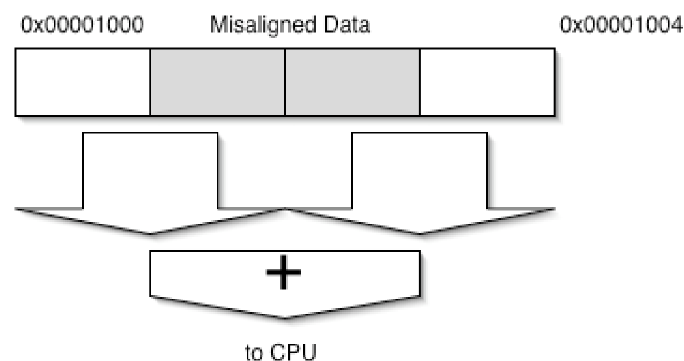

Figure 2. Accessing misaligned data. Two memory accesses must be carried out and the result combined.

2.5 Floating Point Types

Similar to integer types, the C++ standard does not specify any particular binary representation for floating point numbers. The standard defines three floating point types: `float`, `double` and `long double`. Type `double` must provide at least as much precision as `float`, and `long double` must provide at least as much precision as `double`. Furthermore, the set of values of the type `float` must be a subset of the values of type `double` and the set of values of the type `double` must be a subset of the values of type `long double`. Although not required by the standard, the implementation of floating point arithmetic used by most C++ compilers conforms to a standard, IEEE 754-1985, at least for types float and double. This is directly related to the fact that the floating point units of modern CPUs also implement this standard. The IEEE 754 standard specifies the binary format for floating point numbers, as well as the semantics for floating point operations.

Byte order also applies to floating point types, so on a big-endian machine, the constant π = 3.1415926... (or, strictly speaking, an approximation of it) as a `float` value, will be stored as `0x40 0x49 0x0F 0xDB`, while on a little-endian machine it will be stored as `0xDB 0x0F 0x49 0x40`. Of all three floating point types, long double is the least portable. Some systems do not support long double directly; on these systems, it is the same size as a double. Byte order issues aside, `float` and `double` values usually can be exchanged in binary format between platforms; `long double` values cannot.

The IEEE 754 standard has some special features, and they are a major reason for problems when porting floating point code between platforms. *NaN* (Not a Number) is a special value representing the result of taking the square root of a negative number. *Infinity* is the result of a divide by zero. *Signed zero* means there are two representations for zero: +0 and –0. There are also *gradual underflow* and *denormal numbers*, but these two features are not universally available. Finally, there are various *rounding modes*, *floating point exceptions* and *flags*, which may be useful to some applications. C++ does not provide support for working with these features (for example, checking for NaN or Infinity), but there are usually non-standard functions provided by runtime libraries. Also, the C99 standard provides some support here.

Neither the IEEE 754 nor the C++ standard specify how transcendental functions like `sin()`, `cos()`, `exp()`, etc. have to be implemented. Therefore, for the same arguments, the results that these functions return on various platforms might not be the same. This is another reason for getting non-identical results for the same expression on different platforms. The standard also does not specify how the conversion from decimal floating point values to their binary representation must be implemented.

3 Operating System APIs

Modern general-purpose operating systems give application and system programmers a variety of programming interfaces to work with. At the lowest level, every operating system provides so-called system calls, system services or executive services – functions directly implemented in the operating system kernel, and invoked via some kind of software interrupt or trap mechanism. On a higher level are library functions, such as those from the C library on Unix. Windows NT goes one step further and has so-called personalities or subsystems – different sets of programming interfaces that are all implemented in terms of the low-level system services. The Win32 API is not, as one might easily guess, the native programming interface of Windows NT, but rather a higher-level API built atop the largely undocumented executive services.

System calls and library functions differ significantly between operating systems. The system services of OpenVMS have not much in common with the Unix system calls, and the Windows executive services, although conceptually similar to OpenVMS system services, are an altogether different thing again. Parts of the Unix C library have been standardized in the ANSI and ISO standards for C, and many C compilers for non-Unix platforms provide some degree of Unix compatibility at the library level.

The situation is no different for embedded/real-time operating systems. While the APIs offered are more lightweight than that of their general-purpose operating system counterparts, the issues of everyone cooking up its own soup are the same.

3.1 Namespace Pollution

Some operating systems use macros in their system header files in a way that can lead to nasty troubles for C++ programmers. Macros do not respect C++ namespaces, and this is where the problem originates. Windows platforms provide two versions of most API calls. One variant that works with Unicode strings, where each character is 16 bits wide, and a second variant that works with 8-bit characters. For example, there is a function `CreateFileW` that takes an Unicode string as first argument, and a function `CreateFileA` that takes an 8-bit string as first argument. What most programmers use, however, is `CreateFile`. A look at the Windows header file `<winbase.h>`, reveals following definitions:

```
WINBASEAPI
HANDLE
WINAPI
CreateFileA(
    IN LPCSTR lpFileName,
    IN DWORD dwDesiredAccess,
    IN DWORD dwShareMode,
    IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    IN DWORD dwCreationDisposition,
    IN DWORD dwFlagsAndAttributes,
    IN HANDLE hTemplateFile
    );
WINBASEAPI
HANDLE
WINAPI
CreateFileW(
    IN LPCWSTR lpFileName,
    IN DWORD dwDesiredAccess,
    IN DWORD dwShareMode,
    IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    IN DWORD dwCreationDisposition,
    IN DWORD dwFlagsAndAttributes,
    IN HANDLE hTemplateFile
    );
#ifdef UNICODE
#define CreateFile  CreateFileW
#else
#define CreateFile  CreateFileA
#endif // !UNICODE
```

Depending on whether the `UNICODE` macro is defined during compilation, or not, `CreateFile` will either expand to `CreateFileW` or `CreateFileA`. Of course, the preprocessor will expand every instance of `CreateFile` it finds, no matter where it is, as soon as `<winbase.h>` is included. It's easy to see where this leads to if there is a class with a member function named `CreateFile`.

While this namespace pollution issue is worst on Windows – many candidates for good method names are already names of Windows API functions – it can also happen on other platforms. A good way to avoid this problem as much as possible is to use method names that start with a lower case letter.

3.2 POSIX

The name POSIX, pronounced *pahz-icks*, is an acronym for Portable Operating System Interface. It originally referred to IEEE Std 1003.1-1988, the first outcome of a standardization effort started in 1985 by a bunch of Unix vendors, but nowadays refers to a whole family of standards: IEEE Std 1003.*n*. To avoid ambiguities, the 1003.1 standard is often referred to as POSIX.1. It describes the programming interface to the operating system, and thus is the only standard interesting from a system programming point of view.

POSIX.1 is also an international standard of the International Organization for Standardization, published as ISO/IEC 9945. In other words, the corresponding revisions of IEEE Std 1003.1 and ISO/IEC 9945 are the same.

Over the years POSIX.1 underwent several revisions, in 1990 (incorporating ANSI C), 1992, 1996, 2001 and 2003. A few amendments were created as well, which were later on merged back into the main text. Examples are POSIX.1b-1993 Realtime Extension, and POSIX.1c-1995 Threads.

As implied by its name, POSIX does not describe the implementation of an operating system, but rather the programming interfaces an operating system must provide to its applications. Therefore, a POSIX compliant operating system need not have anything to do with the classic Unix System V or BSD implementations. For example, Windows NT had a POSIX subsystem (that later was removed with the release of Windows XP), and QNX, although technically not a Unix system, is fully POSIX compliant.

Among other things, POSIX specifies the name, signature and semantics of well over 1000 functions, as well as the name and location of the corresponding C header files. It does not distinguish between system calls and standard library functions, so implementations are free to implement the functions in the way that best suits them.

The POSIX standard has been widely adopted by vendors of embedded/real-time operating systems. Some of these operating systems offer native POSIX-compliant APIs, like QNX or Integrity. Others, for example VxWorks, provide compatibility libraries that implement POSIX interfaces on top of the native operating system APIs. Nevertheless, there are operating systems without any support for POSIX.

3.3 An Object-Oriented Operating System API

For pure C development, the POSIX APIs may be sufficient as a portable interface to the operating system, even if it means that a POSIX compatibility layer may need to be written for some systems. In contrast, for C++ development, an object-oriented interface to the operating system that uses C++ features to their full extent (classes, exception handling) is desirable, as it significantly reduces the complexity of working with an operating system API, in addition to acting as an abstraction layer that improves code portability.

3.3.1 Abstracting Operating System Facilities

As an example that shows how an operating system facility can be wrapped in a C++ class, a `Mutex` class implementing a mutual-exclusion object or binary semaphore is used in the remainder of this section. The `Mutex` class is straightforward and looks as follows:

```
class Mutex
{
public:
    Mutex();
    ~Mutex();
    void lock();
    void unlock();

private:
    Mutex(const Mutex&);
    Mutex& operator = (const Mutex&);
};
```

A typical usage of a `Mutex` is to implement a critical section:

```
static Mutex mtx;
mtx.lock();
// some code that needs to be in a critical section
mtx.unlock();
```

The code above has a problem: it is not exception safe. If an exception is thrown from within the critical section, the mutex remains locked forever, leading to a hanging program the next time the critical section is to be executed. The correct way to implement the critical section thus would be:

```
static Mutex mtx;
mtx.lock();
try
{
    // some code that needs to be in a critical section
}
catch (...)
{
    mtx.unlock();
    throw;
}
mtx.unlock();
```

The additional error handling code does not look very nice – after all, C++ exceptions are meant to reduce to amount of explicit error handling code. Furthermore, it is very easy to forget a call to `unlock()`. A better solution is strongly desired. This is where the power of C++ kicks in. A `ScopedLock` class can be easily implemented, that guarantees that a mutex is always unlocked appropriately.

```
class ScopedLock
{
public:
    ScopedLock(Mutex& mutex): _mutex(mutex)
    {
        _mutex.lock();
    }
    ~ScopedLock()
    {
        _mutex.unlock();
    }

private:
    Mutex& _mutex;

    ScopedLock();
    ScopedLock(const ScopedLock&);
    ScopedLock& operator = (const ScopedLock&);
};
```

With the ScopedLock class, instead of the error-prone 12 lines above, the following elegant code achieves the same result:

```
static Mutex mtx;
{
    ScopedLock lock(mtx);
    // some code that needs to be in a critical section
}
```

Putting the `ScopedLock` object into a block ensures that the mutex will always be locked upon entering, and unlocked upon leaving the block, no matter whether the block is left regularly, or as the consequence of throwing an exception.

3.3.2 Implementing an Abstraction Layer

The `Mutex` class must now be implemented for all the supported platforms. This leads to an interesting problem. Different implementations of the same interface are needed for the various platforms. How shall these different implementations be organized? First, a common header file is needed that contains the definition of the `Mutex` class. However, the class definition will not be the same for all platforms. This can be accounted for by using macros and conditional compilation which, in turn, leads to hard readable and thus hard maintainable code. Conditional compilation can also be used to pull in the correct class definition for the respective platform from another header file. This approach is somewhat better, but still has two drawbacks. First, there is no reference definition of the interface. One might someday put additional methods into the class, but it is very easy to forget to put these into the implementations for other platforms as well. Second, one might want to look at the class definition to look up some information — the class definition is, after all, still the most authoritative reference. Using this approach, one has to look for the class definition in a platform specific header file, not in the common header file actually used in the source code.

C++ provides us with a facility that is the solution to our problem: *private inheritance*. If class B is a private base class of class C, then only methods of C (and its friends) can access the public and protected members of B. Subclasses of C or other unrelated classes cannot access them. This is ideal for the intended purpose.

For every platform, a class `MutexImp`, having the interface shown in the following example, is implemented:

```
class MutexImp
{
protected:
    MutexImp();
    ~MutexImp();
    void lockImp();
    void unlockImp();
};
```

All members of `MutexImp` are protected, so this class cannot be used on its own. It can, however, be used as a base class for `Mutex`. Inheriting private from `MutexImp` ensures that no one but `Mutex` gets access to the members of `MutexImp`. The definition and implementation of `Mutex` therefore looks as follows:

```
class Mutex: private MutexImp
{
public:
    Mutex();
    ~Mutex();
    void lock();
    void unlock();
```

```
};

...

inline void Mutex::lock()
{
    lockImp();
}

inline void Mutex::unlock()
{
    unlockImp();
}
```

The `lock()` and `unlock()` methods are implemented as inline methods, to mitigate for the additional indirection inducted by the call to the implementation method. It now only must be made sure that the correct implementation of `MutexImp` for the platform gets included, using conditional compilation.

An implementation of `MutexImp` for platforms supporting POSIX threads could be like this:

```
class MutexImp
{
protected:
    MutexImp();
    ~MutexImp();
    void lockImp();
    void unlockImp();

private:
    pthread_mutex_t _mutex;
};

MutexImp::MutexImp()
{
    if (pthread_mutex_init(&_mutex, 0))
        throw SystemException("cannot create mutex");
}

MutexImp::~MutexImp()
{
    pthread_mutex_destroy(&_mutex);
}

inline void MutexImp::lockImp()
{
    if (pthread_mutex_lock(&_mutex))
        throw SystemException("cannot lock mutex");
}

inline void MutexImp::unlockImp()
{
    if (pthread_mutex_unlock(&_mutex))
        throw SystemException("cannot unlock mutex");
}
```

3.4 Using Third-Party Class Libraries

There are class libraries available, both from open-source projects and commercial vendors, that support cross-platform development. The C++ Portable Components, written by the author of this paper, are just one example [2]. An exhaustive resource for finding such a library is the Available C++ Libraries FAQ [3]. However, most of these libraries only target

Windows and Unix platforms, and support for embedded/real-time operating systems is limited, if available at all. It most be noted that using such a library only brings one half the way towards true cross-platform programming. The other half is a solid understanding of all the C++ cross-platform programming issues, as discussed in this paper.

4 Interfacing Hardware

Using the object-oriented features of C++, hardware can be easily represented by classes. The same technique that has been used for abstracting operating system facilities can be used for abstracting hardware. Alternatively, public inheritance together with protected virtual member functions can be used to implement different versions of devices with the same interface.

4.1 Input/Output Ports

The methods for accessing input/output ports are highly platform specific. Not even the POSIX specification defines a standard API for this. At the lowest level, an I/O register can be represented as a class. For example, using the private inheritance technique:

```
class IOReg16: private IOReg16Imp
{
public:
    IOReg16(UIntPtr addr);
    void set(UInt16 value);
    UInt16 get() const;
};
```

This, again, can be used as a building block for a higher-level abstraction, for example a digital-to-analog converter (DAC):

```
class DAC
{
public:
    DAC(IOReg16& reg);
    void set(float voltage);
    const float MAX_VOLTAGE = 2.4;

private:
    IOReg16& _ioReg;
};

DAC::DAC(IOReg16& reg): _ioReg(reg)
{
}

inline void DAC::set(float voltage)
{
    _ioReg.set((UInt16) (voltage*0xFFFF/MAX_VOLTAGE));
}
```

4.2 Interrupts

Interrupt service routines (ISRs) can be written in C++, provided the RTOS has some basic support for writing ISRs in C++ or C. If not, the use of assembly language or platform-specific language extensions might be required to use C++ code in an ISR, due to incompatible calling conventions. The code implementing an ISR should be as short as possible. The non time-critical parts of handling an interrupt should be loaded off to a separate thread. An event queue (implemented as a static ring buffer, since no memory can be allocated in an ISR) can be used to pass data from the ISR to the handler thread. It is then

possible to write the code for the handler thread in a portable way. ISR code should be clearly separated from the other, portable parts of a program, to ensure maintainability.

5 Conclusions

Designing and building portable software in C++ is not a trivial task. Doing it successfully requires experience, as well as careful planning and extra thoughts when designing the system. Nevertheless, portable software can be built quite successfully with C++. As a side effect, software designed and implemented with portability in mind brings with it a cleaner design and improved maintainability. The benefits of building C++ software in a portable way can by far outweigh the additional effort required.

References

[1] Stroustrup, B., “A History of C++: 1979-1991”, in ACM SIGPLAN Notices, Volume 28, No. 3, 1993

[2] Obiltschnig, G., “The C++ Portable Components for Embedded Development”, 2005, at `http://www.appinf.com/poco/index.html`

[3] N. Lokke, “Available C++ Libraries FAQ”, 2005, at `http://www.trumphurst.com/cpplibs/cpplibs.phtml`