Abstraction and the C++ machine model

Bjarne Stroustrup
Texas A&M University
(and AT&T Labs – Research)
http://www.research.att.com/~bs

Abstract

C++ was designed to be a systems programming language and has been used for embedded systems programming and other resource-constrained types of programming since the earliest days. This paper will briefly discuss how C++'s basic model of computation and data supports time and space performance, hardware access, and predictability. If that was all we wanted, we could write assembler or C, so I show how these basic features interact with abstraction mechanisms (such as classes, inheritance, and templates) to control system complexity and improve correctness while retaining the desired predictability and performance.

Ideals and constraints

C++ [ISO, 2003] [Stroustrup, 2000] is used in essentially every application areas, incl. scientific calculations, compilers, operating systems, device drivers, games, distributed systems infrastructure, animation, telecommunications, embedded systems applications (e.g. mars rover autonomous driving), aeronautics software, CAD/CAM systems, ordinary business applications, graphics, e-commerce sites, and large web applications (such as airline reservation). For a few examples of deployed applications, see http://www.research.att/~bs/applications.html.

How does C++ support such an enormous range of applications? The basic answer is: “by good use of hardware and effective abstraction”. The aim of this paper is to very briefly describe C++’s basic model of the machine and how it’s abstraction mechanisms map a user’s high-level concepts into that model without loss of time of space efficiency. To put this in context, we must first examine the general ideals for programming that C++ is designed to support:

- Work at the highest feasible level of abstraction

Code that is expressed directly using the concepts of the application domain (such as band diagonal matrices, game avatar, and graphics transforms) is more easy to get correct, more comprehensible, and therefore more maintainable than code expressed using low-level concepts (such as bytes, pointers, data structures, and simple loops). The use of "feasible" refers to the fact that the expressiveness of the programming language used, the availability of tools and libraries, the quality of optimizers, the size of available memory, the performance of computers, real-time constraints, the background of programmers, and many other factors can limit our adherence to this ideal. There are still applications that are best written in assembler or very-low-level C++. This, however, is not the ideal. The challenge for tool builders is to make abstraction feasible (effective, affordable, manageable, etc.) for a larger domain of applications.

By "abstract", I do not mean "vague" or "imprecise". On the contrary, the ideal is one-to-one correspondence between application concepts and precisely defined entities in the source code:

- Represent concepts directly in code
- Represent independent concepts independently in code
- Represent relationships among concepts directly in code
- Combine concepts freely in code when (and only when) combination makes sense

Examples of "relationships among concepts" are hierarchies (as used in object-oriented programming) parameterized types and algorithms (as used in generic programming).

This paper is about applying these ideas to embedded systems programming, and especially to hard-real time and high-reliability embedded systems programming where very low-level programming techniques traditionally have been necessary.

What's special about embedded systems programming? Like so many answers about programming, this question is hard to answer because there is no generally accepted definition of "embedded systems programming". The field ranges from tiny controllers of individual gadgets (such as a car window opener), through stereo amplifiers, rice cookers, and digital cameras, to huge telephone switches, and whole airplane control systems. My comments are meant to address all but the tiniest systems: there can be no ISO C++ on a 4-bit micro-processor, but anything larger than that could potentially benefit from the ideals and techniques described here. The keys from a system design view are

- The system is not just a computer
  - It's a "gadget"/system containing one or more computers
- Correctness
  - "but the hardware misbehaved" is often no excuse
- Reliability requirements
  - Are typically more stringent than for an "ordinary office application"
- Resources constraints
  - Most embedded systems suffer memory and/or time constraints

- Real time constraints
  - Hard or soft deadlines
- No operator
  - Just users of “the gadget”
- Long service life
  - Often a program cannot be updates for the years of life of its gadget
- Some systems can’t be taken down for maintenance
  - Either ever or for days at a time

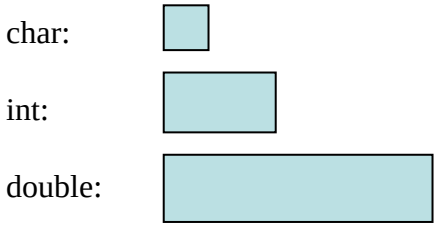
What does C++ have to offer in this domain that is not offered by assembler and C? In particular, what does the C++ abstraction mechanisms offer to complement the model of the machine that C++ shares with C? For a discussion of the relationship between C and C++, see [Stroustrup, 2002].

Machine model

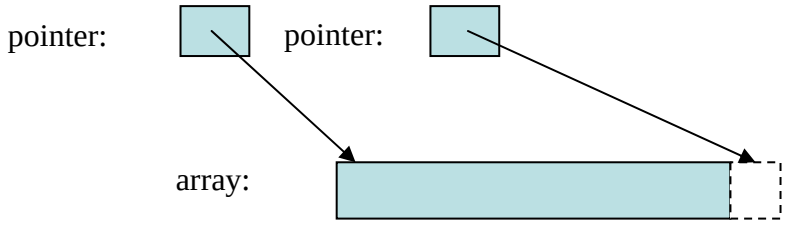
C++ maps directly onto hardware. Its basic types (such as **char**, **int**, and **double**) map directly into memory entities (such as bytes, words, and registers), most arithmetic and logical operations provided by processors are available for those types. Pointers, arrays, and references directly reflect the addressing hardware. There is no “abstract”, “virtual” or mathematical model between the C++ programmer’s expressions and the machine’s facilities. This allows relatively simple and very good code generation. C++’s model, which with few exceptions is identical to C’s, isn’t detailed. For example, there is nothing in C++ that portably expresses the idea of a 2nd level cache, a memory-mapping unit, ROM, or a special purpose register. Such concepts are hard to abstract (express in a useful and portable manner), but there is work on standard library facilities to express even such difficult facilities (see the ROMability and hardware interface sections of [ISO, 2005]). Using C++, we can get really close to the hardware, if that’s what we want.

Let me give examples of the simple map from C++ types to memory. The point here is not sophistication, but simplicity.

Basic arithmetic types are simply mapped into regions of memory of suitable size. A typical implementation would map a **cha**r to a byte, an **int** to a word, and a **double** to two words:



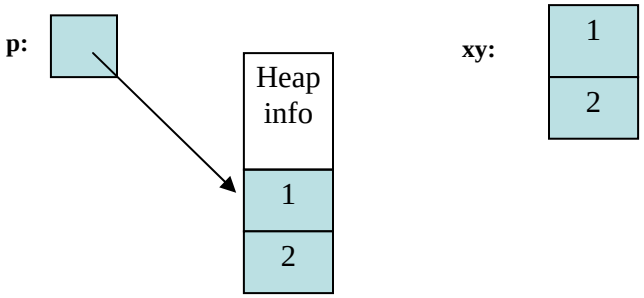
The exact map is chosen so as to be best for a given type of hardware. Access to sequences of objects is dealt with as arrays, typically accessed through pointers holding machine addresses. Often code manipulating sequences of objects deal with a pointer to the beginning of an array and a pointer to one-beyond-the-end of an array:



The flexibility of forming such addresses by the user and by the code generators can be important.

User-defined types are created by simple composition. Consider a simple type **Point**:

```
class Point { int x; int y; /* … */ };
Point xy(1,2);
Point* p = new Point(1,2);
```



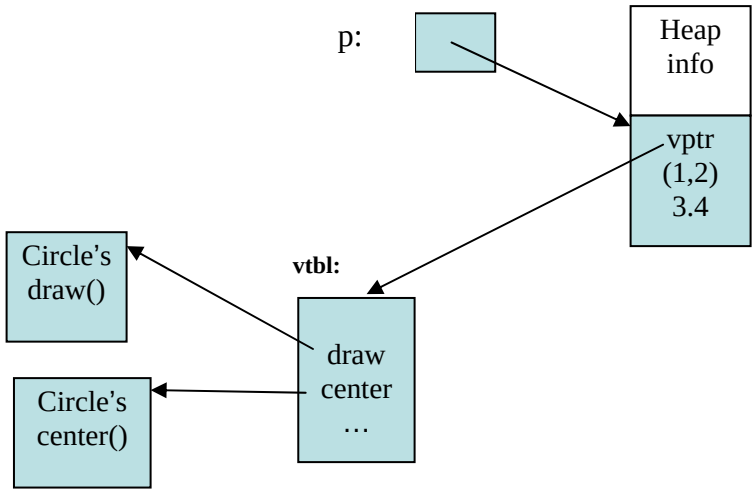
A **Point** is simply the concatenation of its data members, so the size of the **Point xy** is simply two times the size of an **int**. Only if we explicitly allocate a **Point** on the free store (the heap), as done for the **Point** pointed to by **p,** do we incur memory overhead (and allocation overhead). Similarly, basic inheritance simply involves the concatenation of members of the base and derived classes:

```
class X { int b; }
class Y : public X { int d; };
```

Only when we add virtual functions (C++'s variant of run-time dispatch supplying run-time polymorphism), do we need to add supporting data structures, and those are just tables of functions:

```
class Shape {
public:
        virtual void draw() = 0;
        virtual Point center() const = 0;
```

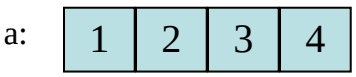
```
        // …
};

Class Circle : public Shape {
        Point c;
        double radius;
public:
        void draw() { /* draw the circle */ }
        Point center() const { return c; }
        // …
};

Shape* p = new Circle(Point(1,2),3.4);
```



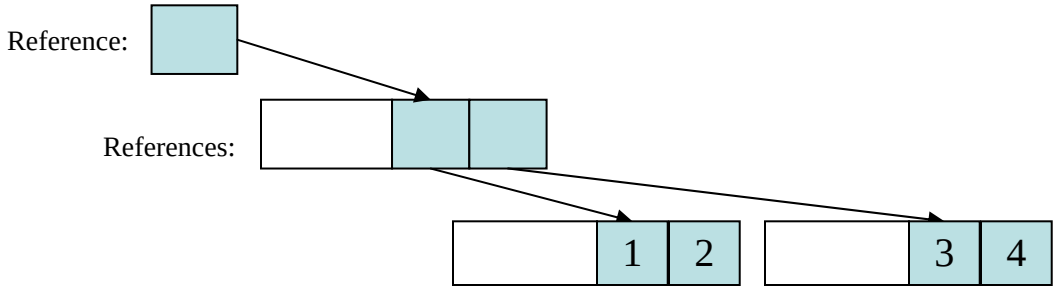
Naturally, this simple picture leaves out a lot, but when it comes to estimating time and space costs it's pretty accurate: What you see is what you get. For more details see [ISO, 2005]. In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [Stroustrup, 1994]. And further: What you do use, you couldn't hand code any better.

Please note that not every language provide such simple mappings to hardware and obeys these simple rules. Consider the C++ layout of an array of objects of a user-defined type:

```
class complex { double re, im; /* … */ };
complex a[ ] = { {1,2}, {3,4} };
```



The likely size is **4*sizeof(double)** which is likely to be 8 words. Compare this with a more typical layout from a "pure object-oriented language" where each user-defined object is allocated separately on the heap and accessed through a reference:



The likely size is **3*sizeof(reference)+3*sizeof(heap_overhead)+4*sizeof(double)**. Assuming a reference to be one word and the heap overhead to be two words, we get a likely size of 17 words to compare to C++'s 8 words. This memory overhead comes with a run-time overhead from allocation and indirect access to elements. That indirect access to memory typically causes problems with cache utilization and limits ROMability.

Myths and limitations

It is not uncommon to encounter an attitude that "if it's elegant, flexible, high-level, general, readable, etc., it must be slow and complicated". This attitude can be so ingrained that someone rejects essentially every C++ facility not offered by C without feeling the need for evidence. This is unfortunate because the low-level alternative involves more work at a lower level of abstraction, more errors, and more maintenance headaches. Bit, byte, pointer, and array fiddling should be the last resort rather than the first choice. C++ balances costs with benefits for "advanced features", such as classes, inheritance, templates, free store (heap), exceptions, and the standard library. If you need the functionality offered by these facilities, you can rarely (if ever) provide better hand-coded alternatives. The ISO C++ standard committee's technical report on performance [ISO, 2005] is provides data and arguments for that proposition.

Obviously, we should not use every feature of C++ for every problem. In particular, not every feature is suitable for hard real time because their performance is not 100% predictable (that is, we can't state in advance exactly how much an operation cost without knowing how it is used and the/or state of the program when it is used). The operations with this problem are:

- Free store (**new**/**delete**): The time needed for an allocation depends on the amount of available free memory and fragmentation can cause deterioration of performance over time. This implies that for many systems, free store cannot be used or can be used only at startup time (no deallocation implies no fragmentation). Alternatives are static allocation, stack allocation, and use of storage pools.
- RTTI (**dynamic_cast**/**typeid**): This is rarely needed in small embedded systems, so just don't use it for such systems. It is possible to implement **dynamic_cast** to

be fast and predictable [Gibbs, 2005] but current implementations don't implement this refinement.

- Exceptions (**throw**/**catch**): The time needed to handle an exception depends on the distance (measured in function calls) from the throw-point to the catch-point and the number of objects needed to be destroyed on the way. Without suitable tools that's very hard to predict, and such tools are not available. Consequently, I can't recommend exceptions for hard real time; doing so is a research problem, which I expect to be solved within the decade. For now, we must use more conventional error-handling strategies when hard real time is needed, and restrict the use of exceptions to large embedded systems with soft real time requirements.

The rest of C++ (including classes, class hierarchies, and templates) can be used and has been used successfully for hard real time code. Naturally, this requires understanding of the facilities and their mapping to hardware, but that's no different from other language constructs. Writing code for hard-real-time or high-reliability systems also requires caution and a good compiler (see http://www.research.att/~bs/compilers.html). It is worth noting that for many styles of usage, modern exception implementations are within 5% of the performance of non-exception code – and that non-exception code must be augmented with alternative exception-handling code (returning error codes, explicit tests, etc.). For systems where exceptions can be used, I consider them the preferred basis for an error-handling strategy [Stroustrup, 2000].

Compilers used for embedded systems programming have switches to disable features where they are undesirable (e.g. in a hard-real time application). Anyway, their use is obvious from the source code.

Abstraction mechanisms

The main abstraction mechanisms provided by C++ are classes, inheritance of classes, and templates. Here, I'll concentrate on templates because they are the key tool for modern statically type-safe high-performance code. Templates are a compile-time composition mechanism implying no runtime or space cost compared to equivalent hand-written code. Templates allow you to parameterize classes and functions with types and integers. If you like fancy words, they provide parametric polymorphism complementing the ad-hoc polymorphism offered by class hierarchies. Generally, systematic use of templates is called "generic programming" which complements the "object-oriented programming" that you get from systematic use of class hierarchies. Both programming paradigms rely on classes.

I will first present some simple "textbook examples" to illustrate the general techniques and tradeoffs. After that, I'll show some real code from a large marine diesel engine using those same facilities to provide reliability, safety, and performance.

Here is a slightly simplified version of the C++ standard library complex type. This is a template class parameterized by the scalar type used:

```
template<class Scalar>
class complex {
    Scalar re, im;
public;
    complex() { }
    complex(Scalar x) : re(x) { }
    complex(Scalar x, Scalar y) : re(x), im(y) { }

    complex& operator+=(complex z) { re+=z.re; im+=z.im; return *this; }
    complex& operator+=(Scalar x) { re+=x; return *this; }

    // …
};
```

This is a perfectly ordinary class definition, providing data members (defining the layout of objects of the type) and function members (defining valid operations). The **template<class Scalar>** says that **complex** takes a type argument (which it uses as its scalar type). Given that definition – and nothing else – we can write

```
complex<double> z(1,2);     // z.re=1; z.im=2;
complex<float> z2 = 3;      // z2.re=3;

z += z2;                    // z.re=z.re+z2.re; z.im=z.im+z2.im;
```

The comments indicate the code generated. The point is that there is no overhead. The operations performed are at the machine level exactly those required by the semantics. A **complex<double>** is allocated as two **double**s (and no more) whereas a **complex<float>** is allocated as two **float**s. A **complex<int>** would make a rather good **Point** type. No code or space is generated for the template class **complex** itself and since we didn't use the **+=** operation taking a scalar argument, no code is generated for that either. Given a decent optimizer, no code is laid down for the used **+=** operation either. Instead, all the operations are inlined to give the code indicated in the comments.

There are two versions of += to ensure optimal performance without heroic efforts from the optimizer. For example, consider:

```
z+=2;       // z.re+=2
z+=(2,0);   // z.re+=2; z.im+=0;
```

A good optimizer will eliminate the redundant **z.im+=0** in the second statement. However, by providing a separate implementation for incrementing only the real part, we don't have to rely on the optimizer to be that clever. In this way, overloading can be a tool for performance.

We can use the += operation to define a conventional binary +:

```
template<class S>
complex<S> operator+(complex<S> x, complex<S> y)
{
        complex<S> r = x;    // r.re=x.re; r.im=y.im;
        r+=y;                // r.re+=y.re; r.im+=y.im;
}

// define complex variables x and y
complex<double> z = x+y;   // z.re=x.re+y.re; z.im=x.im+y.im;
```

Again the comments indicate the optimal code that existing optimizers generate for this. Basically, the templates map to the implementation model for classes described above to give good use of memory and inlining of simple function calls ensures good use of that memory. By "good" I mean "optimal given a good optimizer" and optimizers that good are not uncommon. The example above might make a simple first test of your compiler and optimizer if you want to see whether it is suitable for an application.

The performance of this code depends on inlining of function calls. It has correctly been observed that inlining can lead to code bloat when a large function is inlined many times (either for many different calls or for s few calls but with different template arguments). However, that argument does not apply to small functions (such as, the += and + defined for **complex**) where the actual operation is smaller and faster than the function preamble and value return. In such cases, inlining provides improvements in both time and space compared with ordinary functions and ordinary function calls. In fact, a popular use of class objects and inline function is to implement parameterization where the parameter can be a single machine instruction, such as < [Stroustrup, 1999].

Inlining a large function is usually a very bad idea. Doing so typically indicates carelessness on behalf of the programmer or a poorly tuned optimizer.

In sharp contrast to the claim that templates cause code bloat, it so happens that templates can be used to save code space. A C++ compiler is not allowed to generate code for an unused template function. This implies that if a program uses only 3 of a template class' 7 member functions, only those three functions will occupy space in memory. The equivalent optimization for non-template classes is not common (the standard doesn't require it) and extremely hard to achieve for virtual functions.

The perfect inlining of small member functions and the guarantee that no code is generated for unused functions is the reason that function objects have become the preferred way of parameterizing algorithms. A function object is an object of a class with the application operator () defined to perform a required action. For example

```
template<class T> struct less {
        bool operator()(const T& a, const T& b) const { return a<b; }
};
```

This function object, **less**, is used by most standard library facilities that need to perform a comparison. The result can be factors of improvement in run time compared to parameterization with a function pointers for algorithms such as **sort()** [Stroustrup, 1999].

Most uses of templates are described as “generic programming” or “template meta-programming”. Both are based on overloading where we let the compiler pick the right implementation based on types (and integer values). The simplest and most familiar example is the compiler choosing the right implementation of + when we add **int**, **double**, **complex**, etc. values. The compiler can pick the right function (or basic operation) based on argument types. Similarly, the compiler will pick the right type for an object based on template arguments.

The selection of types and operations is done at compile time and can lead to major improvements. For example, in an embedded application the indirection through pointers to manipulate device drivers turned out to be the bottleneck. The solution was to replace hand-optimized low-level C with templates parameterized on the device register addresses and object types; a 40% improvement in performance was achieved that way. The resulting code was also much shorter and easier to maintain [O’Riorden, 2004]. Section 5 of [ISO, 2005] contains code illustrating such techniques; the examples there relate to a standard interface to special-purpose registers.

It’s amazing what you can do using these techniques. One place to look for techniques and examples is the STL (the C++ standard library’s framework for containers and algorithms) [Stroustrup, 2000]. Since the STL relies on free store it may not be applicable to your particular embedded application, but the techniques are general. For more advanced/extreme uses labeled “template metaprogramming”, see [Abrahams, 2005] and for lots of examples see the Boost collection of libraries [Boost, 2005].

For generality, it is important that templates can have integer arguments. In particular, you can do arbitrary computations at compile time; compile-time constant folding is just the simplest example.

Code examples

Consider briefly a problem faced by the designers of control and monitoring software for large (100,000Hp+) marine diesel engines at MAN B&W Diesel A/S. These engines simply can’t be allowed to fail (or a huge ship is adrift), the engine computers must potentially work for years without maintenance, and programs must be portable to new generations of computers (since computer generations are shorted than engine generations) [Hansen, 2004].

How can we compute accurately and safely? Using numbers of different accuracies? And detect errors such as dived by zero and overflow? Fast enough for hard-real time? (on rugged hardware based on 25MHz Motorola 68332 processors used for electronic fuel injection). The solution chosen and now running on huge ships on the high seas involves:

- Make a template class for fixed-point arithmetic
  - Fixed point is completely portable
  - Fixed point is most efficient on the relevant processors
- Use template specializations where needed

As expected and required, this solution has zero overhead in time and space.

Consider first an example of a function that performs a critical computation. I have done nothing to this code except adjusting the indentation. I am told that it is easy to read if you understand about the engine. Having seen far worse looking code for far simpler problems, I have no trouble believing that:

```
StatusType<FixPoint16> EngineClass::InternalLoadEstimation(
                            const StatusType<FixPoint16>& UnsigRelSpeed,
                            const StatusType<FixPoint16>& FuelIndex)
{
        StatusType<FixPoint16> sl =UnsigRelSpeed*FuelIndex;

        StatusType<FixPoint16> IntLoad =
          sl*(PointSevenFive+sl*(PointFiveFour-PointTwoSeven*sl))
          - PointZeroTwo*UnsigRelSpeed*UnsigRelSpeed*UnsigRelSpeed;

        IntLoad=IntLoad*NoFuelCylCorrFactor.Get();

        if (IntLoad.GetValue()<FixPoint16ZeroValue)
                IntLoad=sFIXPOINT16_0;

        return IntLoad;
}
```

The 16-bit fixed point type is just an ordinary class:

```
stuct FixPoint16 {
        FixPoint16();
        FixPoint16(double aVal);

        bool  operator==(const FixPoint16& a) const { return val==a.val; }
        bool  operator!=(const FixPoint16&) const;
        bool  operator>(const FixPoint16&) const;
        bool  operator<(const FixPoint16&) const;
        bool  operator>=(const FixPoint16&) const;
        bool  operator<=(const FixPoint16&) const;

        short    GetShort() const;
        float    GetFloat() const;
```

```
        double  GetDouble() const;
private:
        long    val;    // e.g. 16.16
};
```

The real computation (of engine status) takes place on status types (parameterized by arithmetic types, such as **FixPoint16**):

```
template <class T>
struct StatusType {
  StatusType();
  StatusType(const StatusType&);
  StatusType(const T aVal,const unsigned long aStat);

  // Member Compound-assignment operator functions:
  StatusType& operator+=(const StatusType&);

  // Miscellaneous:
  const T& GetValue() const;

  // Access functions for status bits:
  bool isOk() const;
  bool IsValid() const;
private:
  T value;
  unsigned long  fpStatus;  // Bit codes defined by type tagFixPoint16Status
};
```

This template class is designed and implemented using the techniques we saw for **complex**. For its time and space performance, it relies on the same techniques and optimizations. This implies that the techniques (and the tools that supports them) are effective in real-world embedded systems contexts.

The low-level details of the engine and the processor are encoded in constants and encapsulated in functions relying on such constants:

```
template<class T>
inline bool StatusType<T>::IsValid() const
{
  return (bool)((fpStatus & 0x0000FFFF) ==VS_VALID);
}


template <>
StatusType<long>&
        StatusType<long>::operator+=(const StatusType<long>& rhs)
```

```
{
    long sum = value + rhs.value;

    if ((value ^ sum) & (rhs.value ^ sum) & LONG_MSB)  { // overflow
        AppendToStatus(VS_OVERFLOW);
        value = (sum & LONG_MSB ? LONG_MAX : LONG_MIN);
    }
    else {
        value = sum;
    }

    AppendToStatus(rhs.GetStatus());

    return (*this);
}
```

The designers of this software emphasize (my translation from Danish):

- C++ is not just used as ”A better C”
  - Our results far exceeded our outside consultants experience with comparable C-based projects.
- Heavy use of object-oriented techniques
  - Including class hierarchies and virtual functions
- Heavy use of generic programming and templates
  - Essential to avoid code duplication
  - Essential to achieve optimal performance
  - Object-oriented and generic programming used in combination
- A good tool chain is essential

The code does not use exceptions (since it is a hard-real-time program) and free store allocation is only used during startup where memory exhaustion and fragmentation cannot occur.

Acknowledgements

Special thanks to Mogens Hansen and Martin O’Riorden for making their examples available to me and educating me in some newer techniques used in performance-critical and safety-critical embedded systems programming. Also thanks to the members of the ISO C++ standard committee’s Performance working group who collected the information for [ISO, 2005].

References

[Abrahams, 2005] David Abrahams and Aleksey Gurtovoy: “C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond”. Addison Wesley. 2005. ISBN 0-321-22725-5.

[Boost, 2005] www.boost.org.

[Gibbs, 2005] Michael Gibbs and Bjarne Stroustrup: “Fast Dynamic Casting”. Software-Practice&Experience. Wiley. To appear 2005.

[Hansen, 2004] Mogens Hansen: “C++ I embedded systemer”. Elektronik 04. Odense Congress Center. September 2004. And personal communication.

[O’Riorden, 2004] Martin J. O’Riordan: “C++ For Embedded Systems”. And personal communications.

[ISO, 2003] “The C++ Standard” (ISO/IEC 14882:2002). Wiley 2003. ISBN 0 470 84674-7.

[ISO, 2005] “Technical Report on C++ Performance”. ISO.IEC PDTR 18015. (http://www.research.att.com/~bs/performanceTR.pdf).

[Stroustrup, 1994] Bjarne Stroustrup: “The Design and Evolution of C++”. Addison Wesley, 1994. ISBN 0-201-54330-3.

[Stroustrup, 1999] Bjarne Stroustrup: “Learning standard C++ as a new language”. C/C++ Users Journal. May 1999

[Stroustrup, 2000] Bjarne Stroustrup: “The C++ Programming Language”. Addison Wesley. 2000. ISBN 0-201-70073-5.

[Stroustrup, 2002] B. Stroustrup: “C and C++: Siblings”, “C and C++: A Case for Compatibility”, “C and C++: Case Studies in Compatibility”. The C/C++ Users Journal. July, August, and September 2002.