Foundations of C++

Bjarne Stroustrup

Texas A&M University
bs@cs.tamu.edu

Abstract. C++ is a large and complicated language. People get lost in details. However, to write good C++ you only need to understand a few fundamental techniques – the rest is indeed details. This paper presents fundamental examples and explains the principles behind them. Among the issues touched upon are type safety, resource management, compile-time computation, error-handling, concurrency, performance, object-oriented programming, and generic programming. The presentation relies on and introduces a few features from the recent ISO C++ standard, C++11, that simplify the discussion of C++ fundamentals and modern style.

Keywords: C++, programming style, fundamental techniques.

1 Introduction

A programming language – any programming language – has a few fundamental constructs, techniques, and underlying models. Understand those and you have a good idea of what can be expressed in the language, and how. In addition, most languages – and especially older languages that are maintained with a concern for compatibility – provides a host of "incidental" features that can distract from understanding and complicate use. Here, I will briefly present most of the key concepts of C++. Naturally, my presentation will not be complete in either features offered or their details. That's what textbooks and standards are for. So, with the caveat that there is always much more that could be said, here we go!

C++ is defined by its ISO Standard [1]. A detailed description can be found in [2], a tutorial for beginners in [3], and a list of language and library features added for C++11 in [4].

I assume that you know about traditional naming and lexical scoping so I don't waste time on such topics. Similarly, I assume that you are at least superficially acquainted with C/C++ syntax and linkage conventions.

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 1–25, 2012.

2 Ideals

The aim of C++ is to help in classical systems programming tasks. It supports the use of light-weight abstraction for resource-constrained and often mission-critical infrastructure applications. By "light-weight abstraction," I mean abstractions that do not impose space or time overheads in excess of what would be imposed by careful hand coding of a particular example of the abstraction. The aim is to allow a programmer to work at the highest feasible level of abstraction by providing

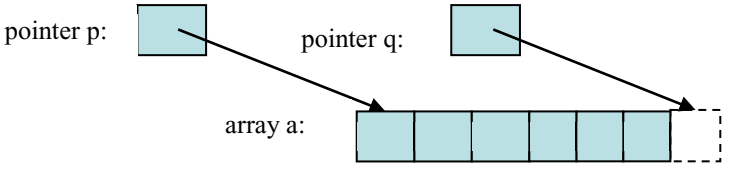
- A simple and direct mapping to hardware
- Zero-overhead abstraction mechanisms

The aim is to support a type-rich style of programming. In particular, C++ supports type-safe programming with a non-trivial set of types.

Naturally, not every application meets these ideals. In particular, a programmer can choose to write a low-level-C style and/or violate every rule of good programming. That is not my topic here.

3 Memory and Objects

C++ maps directly onto hardware. Its basic types (such as, **char**, **int**, and **double**) map directly into memory entities (such as, bytes and words), 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. Memory is seen as sequences of bytes. A typed object is given a location in memory (a sequence of bytes) and values are placed in such objects. Sequences of objects are dealt with as arrays, typically accessed through pointers holding machine addresses. Often, code manipulates sequence of objects defined by a pointer to the beginning of an array and a pointer to one-beyond-the-end of an array:

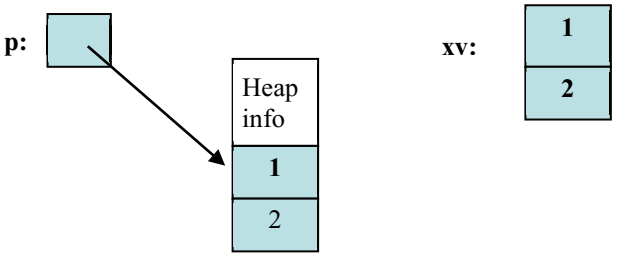


That is, the array **a** can be seen as a half-open sequence of elements [**p**:**q**). The flexibility of forming such addresses by the user and by code generators can be important.

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

Point:

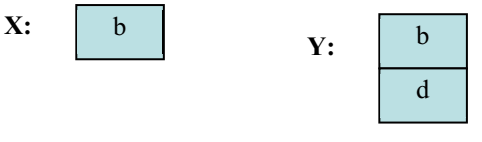
```
class Point { int x; int y; /* … */ };

Point xy {1,2};                          // named and scoped object

Point* p = new Point{1,2};      // free store (dynamic, heap) object
```



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**. Named objects (of any built-in or user-defined type) are allocated statically or on the stack. Only if we explicitly allocate an (unnamed) **Point** on the free store (the heap), as done for the **Point** pointed to by **p**, do we incur memory overhead (and allocation overhead). Such very simple user-defined types are critical to type-rich programming and very common

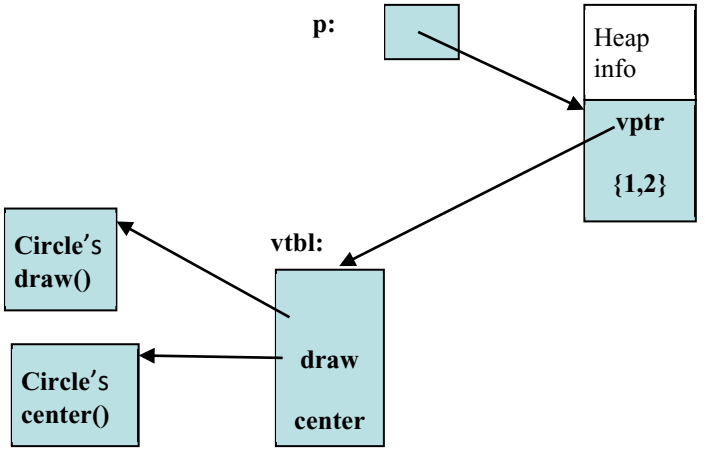
Similarly, basic inheritance simply involves the concatenation of members of the base and derived classes:

```
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:

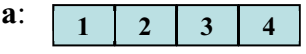
```
class Shape {                          // a base class; an interface

public:
        virtual void draw() = 0;
        virtual Point center() const = 0;
        // …
};

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

Shape* p = new Circle{Point{1,2},3.4};
```



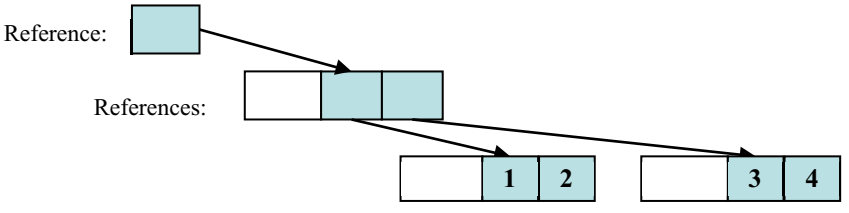
What you see is what you get. For more details see [5]. In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [6]. And further: What you do use, you couldn't hand code any better.

Please note that not every language provides 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} };
```

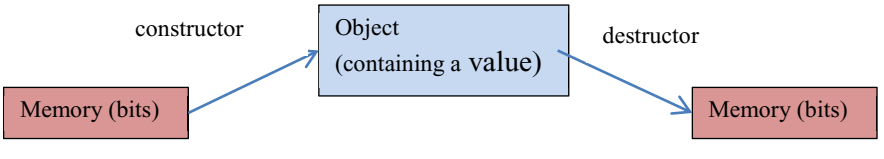


The likely size is **4*sizeof(double)** which is likely to be 8 words (assuming a 32-bit word). Compare this with a more typical layout from a "pure object-oriented language" where each user-defined object is allocated separately on the heap and accessed through a reference:



Here, **3*sizeof(reference)+3*sizeof(heap_overhead)+4*sizeof(double)** is the likely size. Assuming a reference to be one word and the heap overhead to be two words, we get a likely size of 19 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.

Memory is turned into an object containing a value of some type by a constructor [6,7]. This operation is reversed by a destructor: after a destructor is run the object no longer exist and its former location is simply memory again. The meaning of constructors and destructors for built-in types and simple aggregates are language defined. For more complex types, the programmer can define constructors and destructors.



4 Compile-Time Computation

Sometimes, we prefer a computation be done at compile-time. The reasons vary, for example:

- *Efficiency*: To pre-calculate a value (often a size). For simple cases, that is done by an optimizer. Examples include object and array sizes and table values.
- *Type-safety*: To compute a type at compile time.
- *Simplify concurrency*: you can't have a race condition on a constant.

In C++11, we can do type-rich computation at compile time. Consider a simple distance calculation:

```
constexpr double d = dist(NewYork,Boston);
```

Here, I assume that the city names are 2D grid points and that **dist()** computes the distance between them. The **constexpr** keyword is C++'s way of requesting compile-time evaluation. The code doing the calculation might look like this:

```
struct City { double   x, y };
constexpr double csqrt(double) { /* calculate square root */ }
constexpr double square(double d) { return d*d; }
constexpr double dist(City c1, City c2)
        { return csqrt(square(abs(c1.x-c2.x))+square(abs(c1.y-c2.y))); }
```

I had to define my own **csqrt()** because the standard library **sqrt()** isn't designed to work at compile time; **constexpr** is C++'s way of requiring that a function is executable at compile-time. If I wanted to, I could add unit checking [8,9]:

```
constexpr Distance d = dist_in_km(NewYork,Boston);
```

Various forms of user-specified compile-time computation are essential in critical embedded systems applications, much low-level code, and many high-end numerical applications.

5 Error Handling

Errors that cannot be handled locally are reported by throwing an exception. An exception is a value of some type, usually a user-defined type. Code that is interested in handling a type of exception provides a handler (catch-clause) for it. For example:

```
void do_task(int i)
{
        if (i==0) throw std::runtime_error{"do_task() of zero"};
        if (i<0) throw Bad_arg{i};
        // do the task and return normally
}


void task_master(int i)
{
        try {
                do_task(i);
                // …
        }
        catch (Bad_arg a) {
                cout << "do_task() of negative" << a.val << "\n";
        }
}
```

Code that cannot perform its required task throws an exception and that code that requests a task to be done provides a handler for the kinds of errors it is prepared to handle. If an exception that the requestor has not expressed interest in is thrown, the requestor itself fails.

Exceptions can – and often do – carry information. A catch-clause is associated with a try-block. An exception propagated up the call stack until caught. An uncaught exception causes program termination. A thread can transfer a thrown exception (that it is not willing to handle) to another (calling) thread.

For hard-real-time programming (and only for that), this exception-based error handling must be abandoned for a lower-level error-handling style. The reason is that it is hard to provide good real-time guarantees for exception propagation.

6 Containers

How do you store a lot of data? We place it in user-defined containers, such as vectors, lists, and maps. The archetypical C++ container is the vector. Here is a simple first **Vector**:

```
template<typename T>                // T is the element type
class Vector {
public:
    Vector(int   n);                // constructor: initialize to n   elements
    Vector(initializer_list<T>) ;   // constructor: initialize with element list
    ~Vector();                      // destructor: deallocate elements
    int size() ;                    // number of elements
    T& operator[](int i);           // access the ith element
    void push_back(const T& x);     //add x as a new element at the end
T* begin();                         // fist element
    T* end();                       // one-beyond-last element
private:
    int sz;                         // number of elements
    T* elem;                        // pointer to sz elements of type T
};
```

T* means "pointer to **T**" and **T&** means "reference to **T**." Given that declaration, we can allocate and manipulate elements of an arbitrary type, **T**:

```
void f(Vector<string>& vs)
{
        Vector<int> sizes;
        for (auto   x : vs)             // loop through all elements of vs
              sizes.push_back(x.size());
        if (0<vs.size())
              vs[0] = "Whatever!";
           // …
}
```

The range-for loop uses **Vector**'s **begin()** and **end()** members to determine its range. We might call **f()** like this:

```
int main()
{
       f({"Wheeler", "Wilkes", "Radcliffe", "Appleton","Rutherford"});
        Vector<string> places(10);
        places[2] = "Cambridge";
        f(v);
}
```

The declaration of **Vector** separates the class into two parts, the **public** interface and the **private** implementation (so far, just a representation). The implementation of the **Vector** consists of the definitions of the member functions. In particular, the constructors and the destructor manage a **Vector**'s resource, its elements:

```
template<typename T>
Vector<T>::Vector(int n) // make a vector with n elements of default value
    :sz(n)
{
    if (sz<0) throw std::runtime_error{"negative Vector size"};
    elem = new T[sz];       // sz uninitialized memory slots
    std::uninitialized_fill(elem,elem+sz,T{});          // initialize to default
}
```

The standard technique is to throw an exception if a constructor cannot establish its invariant. Here the invariant is that **elem** points to **sz** elements of type **T** allocated on the free store. The **std::** is used to indicate facilities provided by the ISO C++ standard library (so we don't have to do it ourselves). A constructor handling **{}** lists is defined as taking an argument of the standard-library type **initializer_list**:

```
template<typename T>
Vector<T>::Vector(std::initializer_list<T> lst)     // elemens from the list
    :sz{lst.size()}, elem{new T[sz]}
{
    std::uninitialized_copy(lst.begin(), lst.end(), elem);
}
```

The destructor releases resources acquired:

```
template<typename T>
Vector<T>::~Vector()
{
     delete[] elem;
}
```

This **Vector** is pretty basic, but is illustrates several fundamental C++ techniques and their supporting language features. Containers and resource management are not built into the language or into a run-time support system. Instead, only the minimal

facilities for dealing with objects and fixed-sized sequences of objects in memory are "built-in." Everything else is "user-defined" and often provided by libraries written in C++. The standard library **vector**, **map**, **set**, and **list** are examples of containers built using the techniques presented here:

- classes for separating interfaces from implementations,
- constructors for establishing invariants, including acquiring resources,
- destructors for releasing resources,
- templates for parameterizing types and algorithms with types
- mapping of source language features to user-defined code specifying their meaning, e.g. **[]** for subscripting, the **for**-loop, **new/delete** for construction/destruction on the free store, and the **{}** lists.
- use of half-open sequences, e.g. [**begin**():**end**()), to define for-loops and general algorithms.
- Use of standard-library facilities to simplify specification and implementation

Importantly, this abstraction from "memory" to "containers of objects" carries no overheads beyond the code necessarily executed for memory management, initialization, and error checking.

Note that

- There is no data stored in a **Vector** object beyond the two named members
- There is no requirement that the element type should be part of a hierarchy. The only requirements on a template argument are imposed by its use; this is "duck typing."
- The operations on a **Vector** are not required to be dynamically resolved (virtual). Simple operations, such as **size**() and **[]**, are typically inlined.

In other words, these language features and techniques ("abstraction mechanisms") are light weight, aimed for use in demanding systems programming and infrastructure implementation tasks. I could, of course, have built a few key abstractions, such as **vecto**r and **string**, into the language. The reason not to do that is to allow the programmer to define a much larger and varied set of abstractions without losing the flexibility and efficiency needed for the most demanding systems programming tasks.

7 Copy and Move

To complete a class, we have to consider if and how objects can be copied and moved around. As defined above, **Vector** cannot be copied:

```
Vector capitals { "Helsinki", "København", "Riga", "Tallinn" };
Vector c2 = capitals;      // error: no copy defined for Vector
```

By default, you can copy only objects with "simple representations." When I defined a destructor for **Vector**, I implied that I did not consider the representation **Vector** simple: the **elem** pointer represents ownership. Let us define copy:

```
template<typename T>
Vector<T>::Vector(const Vector& v)      // copy constructor
    : sz{v.sz}, elem{new T[v.sz]}
{
    std::uninitialized_copy(v.begin(),v.end(), elem);
}
```

This defines copy initialization. In addition, we can define assignment of one **Vector** by another:

```
template<typename T>
Vector<T> Vector<T>::operator=(Vector<T>& v)    // copy assignment
{
    Vector<T> tmp {v}; // copy v
    delete[] elem;
    elem = tmp.elem;     // "steal" tmp's representation
    tmp.elem = nullptr;
    sz = tmp.elem;
    tmp.sz = 0;
    return *this;
}
```

I chose to have "copy" mean "copy all elements" because that is the intuitive meaning of assignment, fits best with classical mathematical notions, and is what the C++ standard provides for containers.

Copying elements can be costly for large containers, so choosing copy semantics implies a logical or a performance problem. How do we get a **Vector** out of a function? Consider:

```
Vector<int*> find_all(Vector<int>& v, int val)
        // find all occurrences of val in v
{
    Vector<int*> res;
    for (int& x : v)
        if (x==val)
            res.push_back(&x);    // add the address of the element to res
    return res;
}
```

The member function **push_back()** is one of the most useful standard-library container functions. It adds an element to the end of a container, increasing the container's size by one. Here, I have omitted its definition here to avoid getting side tracked. The **find_all** algorithm can be use like this:

```
void   test()
{
    Vector<int> lst { 1,2,3,1,2,3,4,1,2,3,4,5 };
    for (int* p :   find_all(lst,3))
        cout << "address: " << p << ", value: " << *p << "\n";
        // …
}
```

This should work, and it does, but the cost involved in copying the elements out of **find_all()** can be significant. In particular, I might use something like **find_all()** to locate large numbers of elements in **Vector**s of millions of elements. This makes people search for alternatives to returning a container "by value," such as passing a vector to be filled as an argument, returning a pointer to a result stored on the free store, or plugging in a garbage collector. These alternatives all have serious logical or performance problems. Fortunately, there is a much simpler and more general solution: Note that I didn't want to copy anything; I just wanted to transfer (move) the result vector out of **find_all()**. We can define move operations in a way very similar to the way we define copy operations. Move operations "steal" the representation of an object, leaving behind an "empty" object:

```
template<typename T>
Vector<T>::Vector(const Vector&& v)              // move constructor
    : sz{v.sz}, elem{v.elem}                  // grab v's elements
{
    v.elem = nullptr;                 // make v empty
    v.sz = 0;
}
```

```
template<typename T>
Vector<T> Vector<T>::operator=(Vector<T>&& v) // move assignment
{
    delete[] elem;      // delete old elements
    elem = v.elem;      // grab v's elements
    sz = v.sz;
    v.elem = nullptr;   // make v empty
    v.sz = 0;
    return *this;
}
```

The **&&** means "rvalue reference" and the effect is that only rvalues can be used as arguments to move operations. Rvalues are objects that will not be used again, such as a local variable used as the return value.

By using the move constructor rather than the copy constructor to return the value from **find_all**(), that return is efficient even if the returned **Vector** happens to have a million elements.

For this to work, we have to declare the copy and move operations in the definition of **Vector**:

```
template<typename T>  // T is the element type
class Vector {
public:
    // …
    Vector(const Vector&);              // copy constructor
    Vector(Vector&&);                   // move constructor
    Vector& operator=(const Vector&);   // copy assignment
    Vector& operator=(Vector&&);        // move assignment
    // …
};
```

If a class provides both move and copy operations, move is preferred for rvalues and copy for lvalues [10]. All the standard library containers, including **vector** and **string**, have both copy and move operations. This implies that for real program, all the moderately clever and complicated code in the last two sections have already been done for the programmer. What is left is the much simpler use of **vector**, etc.

8 RAII

Systems manipulate resources. We must manage many kinds of resources, such as files, sockets, locks, threads, and database transactions. A resource is anything that a program acquires from another part of the system and must (explicitly or implicitly) release back to its owner after use. An unused and unreleased resource is called a leak. Memory is an important example of a resource. If resources are not properly released, the system's performance will suffer and eventually a long-running system will fail for lack of usable memory. How do we prevent resource leaks?

The constructor/destructor technique used for **Vector** generalizes to any scoped use of a resource and the move technique handles transfers of ownership between scopes. The key idea is that a resource is always owned by a local (scoped) object. Such a local object is sometimes called a resource handle (e.g. file handle), an owner, or simply an interface (e.g., a **Vector** is the interface to its elements). The handle's constructor acquires the resource and the handle's destructor releases it. Consider a standard-library lock used to ensure exclusive access to some shared data:

```
std::mutex m;   // a system resource
int sh;         // shared data

void f()
{
    // …
    std::unique_lock lck(m);      // grab (acquire) the mutex
    sh+=1;                        // manipulate shared data
}                                 // implicitly release the mutex
```

This technique is usually called RAII ("Resource Acquisition Is Initialization") and is widely used in modern C++.

Looking at a simple example, it is tempting to think that a pair of **lock()/unlock()** functions would be as good or better than using an object of the manager type **unique_lock**. In practice, it is not so. A handler has a destructor, so you cannot forget the release operation. But how could anyone forget something as simple as and **unlock()**? Or forget an **fclose()**, a **free()**, or a **delete**? Well, people do, so resource leaks in undisciplined code are common (in any language). There are several reasons, including:

- Often, a resource doesn't look like a resource. For example, a **File*** is just a pointer to the compiler and a casual reader and nothing (except the manual) says that **fclose()** must be called to avoid the leak of a file handle.
- Often, several resources need to be acquired and their patterns of acquisition and release vary.
- Error handling typically requires that a resource is released only if acquired and that related resources are released in some specific order.
- Complex control structures (especially when several functions are involved) obscure acquisition and release patterns.

Consider a simple example:

```
// unsafe, naïve use:
void f(const char* p)
{
    FILE* f = fopen(p,"r");       // acquire
    // use f
    fclose(f);                    // release
}
```

This seems innocent enough, but if the "*use f*" code contains a return statement, a C-style **longjmp()**, or an exception **throw**, we never get to the **fclose()** and we have a leak. This can easily happen if the "*use f*" code is long or complicated. People often try to compensate with code that catches exceptions:

```
// naïve fix:
void f(const char* p)
{
    FILE* f = 0;
    try {
        f = fopen(p, "r");
        // use f
    }
    catch (…) {  // handle every exception
        if (f) fclose(f);
        throw;  // re-throw; let a caller handle this exception
    }
    if (f) fclose(f);
}
```

It is easy to devise a prettier syntax (e.g., Java's **finally**), but the fundamental problem is that any variant of this technique requires the programmer's attention in each place a resource is used. We may open files in dozens of places in a program and in each place the programmer has to remember that **fopen()** acquires a file, be prepared to deal with a failure, and remember to release it. Using return values to handle errors, rather than exceptions, doesn't reduce the complexity.

The solution is to explicitly represent the file handle as a resource:

```
class File_handle {          // belongs in some support library
    FILE* p;
public:
     File_handle(const char* pp, const char* r)
          { p = fopen(pp,r); if (p==0) throw File_error{pp,r}; }
     File_handle(const string& s, const char* r)
          { p = fopen(s.c_str(),r); if (p==0) throw File_error{pp,r}; }

       ~File_handle() { fclose(p); }          // destructor: close file

       // copy and/or move operations
       // access functions
};
```

Now we can simplify the original code to:

```
void f(string s)
{
    File_handle fh {s, "r"};
    // use fh
}
```

The handle class, here **File_handle**, needs only be defined once and put in a library. For example, **unique_lock**, the handle for **mutex**es, is defined in the standard library.

Using such handles, multiple resources are released in the reverse order of acquisition. That is almost always correct. Acquiring a few resources, but failing to acquire all that are needed is handled correctly without programmer intervention. That is crucial for errors in constructors of complex data structures, such as objects from a complex class hierarchy or elements of a container.

Generally, a handle can be moved, but not copied, so move operations need to be provided. For example:

```
class File_handle {          // belongs in some support library
    FILE* p;
public:
     // …

     // move operations:
     File_handle(File_handle&& h) : p{h.p}   { h.p=nullptr; }
     File_handle& operator=(File_handle&& h) { p=h.p;   h.p=nullptr; }

     // access functions
};
```

Given that, we can pass a **File_handle** around (cheaply).

Memory is not the only resource, so a simply adding a garbage collector is not a solution, at least not a complete solution.

9 Class Hierarchies

C++ allows for the definition and use of class hierarchies. The scheme is fairly conventional, but general. It allows for multiple inheritance both of interface classes (abstract classes) and of classes with implementation. The layout of objects is minimal and obvious. The mechanism for virtual function calls is minimal, obvious, and runs in constant time. The resulting compactness, speed, and predictability are essential for many real-time uses. The (classical) **Circle**-and-**Shape** example from "Memory and Objects" is fairly typical. A class can be derived from another (as **Circle** was from **Shape**). The resulting class is called a derived class and – if publicly derived – is a subtype of the other class, called its base.

The protection model is that

- **public** members and bases of a class can be accessed by all
- **protected** members and bases of a class can be accessed only by members of a derived class
- **private** members and bases of a class can be accessed only by members of that class

To avoid confusion and maintenance problems, I recommend using **protected** only for functions and bases.

Abstract classes, like **Shape**, provide the most stable interfaces because they reveal very little about implementation details (such as object sizes), which are supplied in derived classes, such as **Circle**.

C++ does not provide a universal base class. I consider such a class an unnecessary implementation-oriented artifact that imposes avoidable space and time overheads. Also, a universal "Object" base encourages underspecified (overly general) interfaces that let errors that could be detected at compile time through to run time. Typically, C++ uses parameterization where another language might use a common base class and require implicit or explicit type conversion to determine the exact derived class.

10 Algorithms

A function template that implements an algorithm for a variety of types is conventionally called an algorithm. The C++ standard library provides many algorithms, such as **sort**() and **find**(). For example:

```
void f(vector<int>& v, list<string>& lst)
{
    std::sort(v.begin(),v.end());

    // find "Aarhus" in lst:
    auto p = std::find(lst.begin(),lst.end(),"Aarhus");
    if (p!=lst.end()) {        // found: *p=="Aarhus"
                    // …
    }
    else {                     // not found *p!="Aarhus"
                    // …
    }
        // …
}
```

Standard-library algorithms, such as **sort**() and **find**(), take half-open sequences of elements, presented as a pair of iterators, as arguments. An iterator is something that points to an element of a sequence. To get to the next element of a sequence, we use **++** and to access the element pointed to, we use ***.**

Note that I did not name the type of **p**. Instead, I said **auto**, which gives a variable the type of its initializer. This is often a useful shorthand and can be a significant help in generic programming. Here, it saved me from typing **list<string>::iterator**. Interestingly, **auto** is the oldest feature of C++11: I implemented it in 1983, but had to take it out for reasons of C compatibility.

We could implement **find** like this:

```
template<typename Iter, typename Value>
Iter find(Iter first, Iter last, Value val)
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}
```

That is, **find**() compares **val** to each element in the sequence until it finds one that is equal. If you like the terse C-style syntax, you'll find the body of **find**() beautiful. If not, you should still appreciate that **find**() works for a wide variety of data structures and a wide variety of element types with no overhead compared to hand-crafted code for a specific container and value pair. Returning the end of the sequence to indicate "not found" is a standard-library convention.

Algorithms are typically rendered much more useful by parameterizing them with operations. For example, **find()** would be much more useful if instead of simply finding an element of a given value, it found an element that met some user-supplied criterion. The "find" that does that is called **find_if()**. For example:

```
void g(vector< string>& vs)
{
    auto p = std::find_if(vs.begin(),vs.end(),Less_than{"Griffin"});
    if (p!=vs.end()) {        // found: *p<"Griffin"
        // …
    }
    else {                    // not found *p>="Griffin"
        // …
    }
    // …
}
```

Less_than is a function object; that is, an object of type **Less_than** can be called like a function. We can define **find_if()** similarly to **find()**; we just replace the comparison with a call of the predicate:

```
template<typename Iter, typename Value>
Iter find_if(Iter first, Iter last, Predicate p)
{
    while (first!=last && !p(*first))
        ++first;
    return first;
}
```

That is, **find_if** calls the predicate for each element in the sequence. In our example, **p(*first)** means **Less_than{"Griffin"}(*first)** which in turn means ***first<"Griffin"**, assuming that **Less_than** has an obvious definition, such as:

```
struct Less_than {
    String s;
    Less_than(const string& ss) :s{ss} {}    // the value to compare against
    bool operator(const string& v) const { return v<s; } // the comparison
};
```

The general function-object notation can be verbose, but we can let the language write the function object for us by using the lambda notation:

```
auto p = std::find_if(vs.begin(),vs.end(),
                      [](const string& v) { return v<"Griffin"; } );
```

Function objects (incl. lambdas) are very efficient and general because they are easily inlined and can carry information. In particular, simple function objects tend to significantly outperform indirect calls to simple functions. They are the basic parameterization mechanism of the standard library.

If the sequence notation gets too cumbersome, we can define algorithms over containers. For example:

```
namespace MySTL {
        template<class C>
        void sort(C& c) { std::sort(c.begin(),c.end(); }
        // …
}
```

Given that, I can write **sort(v)** for a container **v**, rather than **sort(v.begin(),v.end())**. Notation matters more than we usually like to believe.

Templates are the language-technical basis for generic programming in C++. Similarly, class hierarchies are the language-technical basis for Object-oriented programming in C++. These two programming styles ("paradigms", if you must) are not meant to be disjoint. Rather, they are meant to be used in combination. For example, **vector<Shape*>** is a container of a run-time polymorphic type. Any use will necessarily involve both generic and object-oriented techniques. For example, consider this variant of the classical "draw all shapes example":

```
template<typename Cont>
void draw_all(Cont& c)
{
    for_each(c.begin(),c.end(), [](Shape* p) { p->draw(); }
}
```

Much of the distinction between object-oriented programming and generic programming is an illusion based on a focus on language features and incomplete support for a synthesis of techniques.

11 Type Functions

Templates, as used to parameterize vector with its element type in "Containers," can be seen as generators. A function template generates functions and a class template generates classes. Thus a template can be understood as a function from a set of arguments to a function or a type. For example, **vector<T>** is a function that produces a **vector** of **T**s from the type **T**. The evaluation of such a type function is called template instantiation. Template instantiation is Turing complete [11]. Template arguments are typically types or integers.

This view of templates as type functions gains great practical importance when applied to functions that associate properties to types. For example, elements in a container have a type. We would like to name that type for every data structure we

consider a container, independently of whether its designer planned for that. For starters, we can define a **struct** that defines a name **value_type** for every container that has a member type called **value_type**:

```
template<typename Cont>
struct container_traits {
    using value_type = typename Cont::value_type;
    // …
};


template<typename T>
using Value_type = typename container_traits<T>::value_type;
```

For example, **Value_type<std::vector<int>>** is **int**. Given **container_traits**, we can define **Value_type** for types that do not have a member called **value_type**. For example, for any pointer, **T***, the value type is **T**:

```
template<typename T>
struct container_traits<T*> {
    using value_type = T;
    // …
};
```

Technically, this is a specialization of **container_traits** for pointers. Specialization is the language-technical basis for template metaprogramming [12]. Now, **Value_type<int*>** is **int**. We have provided a type function **Value_type** that provides the type of a contained element for every data structure we consider a container. As a user, the implementation details are immaterial, and we can just write

```
Value_type<X> a;
```

Traits are widely used in the implementation of the standard library.

There is an obvious weakness in my description of **container_traits**: I said "a type that I consider a container" rather than precisely specifying the requirements for being a container. In other words, the arguments to a template are unconstrained and only their instantiations are type checked. This is "Duck tying" ("if it walks like a duck and quacks like a duck, it's a duck") and leads to late (link-time) type checking and appallingly poor error messages.

Designing a system of requirements (called a concept in C++) is still a research topic. A concept design for C++0x [13] failed to meet the needs of C++'s large and diverse user community and concepts is an area of active research [14-17]. The demands of compile-time efficiency (within a few percent of unconstrained templates), run-time efficiency (no slower than templates with unconstrained arguments), ease of use by non-experts, no verbosity, ability to handle type conversion, ability to interoperate with unconstrained templates, and ease of conversion of pre-concept C++ programs makes this a challenging task.

12 Concurrency

C++ must support the forms of concurrency offered by the hardware and operating system on which it runs. Something else may make it a better platform for specific applications, but not supporting "the system's" notion of concurrency would disqualify C++ as a systems programming language. Consequently, ISO standard C++ supports a conventional threads-and-locks model of concurrency. I consider threads-and-locks an unfortunate low-level view, but higher level concurrency models can be efficiently built as libraries on top of what the standard offers. C++ provides support for lock-free programming for cases where you have to get really close to the hardware [18].

What the standard offers differs from earlier C and C++ thread implementations in being type safe. Consider a simple example of a function, **f**, and a function object, **F**, being run on separate threads:

```
void f(vector<double>&);          // function

struct F {                        // function object
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();
};

void code(vector<double>& vec1, vector<double>& vec2)
{
    std::thread t1 {f,vec1};      // f(vec1)
    std::thread t2 {F{vec2}};     // F{vec2}()
    t1.join();
    t2.join();
    // use vec1 and vec2
}
```

For simplicity, I have assumed that **f** and **F** modify their arguments. Note how **t1**'s constructor takes the function to be called followed by its arguments. It will accept any function as long as its arguments type checks using what is called variadic templates. However, here the simplicity of the interface is more important than the implementation technology.

I consider that style of concurrency clumsy, with endless opportunity for confusion and avoidable overheads. However, It does not require regression to type-unsafe the C-style **void**** and macros common in older threads programming and is supported with a variety of synchronization mechanisms (e.g., mutexes, locks, condition variables).

In addition, the standard library supports **future**s to enable a style of concurrency without explicit use of threads and locks. For example:

```
double comp(vector<double>& v)          // spawn many tasks
{
    auto b = v.begin();
    auto sz = v.size();

    auto f0 = std::async(std::accumulate, b, b+sz/4, 0.0);
    auto f1 = std::async(std::accumulate, b+sz/4, b+sz/2, 0.0);
    auto f2 = std::async(std::accumulate, b+sz/2, b+sz*3/4, 0.0);
    auto f3 = std::async(std::accumulate, b+sz*3/4, v.end(), 0.0);

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Here, the "thread launcher" **std::async** launches threads as needed to evaluate **std::accumulate**. Each call of **async** returns a handle, called a **future**, from which the result can be obtained by a call of **get()**. If a task launched by **async** hasn't completed by the call of **get()**, the calling thread waits. This programming model is much cleaner than the more general threads-and-locks model for the independent tasks for which it is intended.

13 Type Safety

C++ is not guaranteed to be statically type safe. A language designed for general and performance critical systems programming with the ability to manipulate hardware cannot be. It provides facilities for manipulating hardware at a low level that can easily be misused to break the type system. Examples are untagged unions, explicit type conversions (casts), arrays without (guaranteed) range checks, and the ability to deallocate a free store (heap) object while holding on to a pointer allowing for post-allocation access. It would be nice to isolate the type violations in a few clearly delimited sections of code, but history precludes that. Don't use these facilities outside the implementation of higher-level facilities (such as **vector**). The ISO C++ standard library contains a rich set of such abstractions (e.g., **string**, **vector**, **map**, **set**, and **thread**), so that you don't have to define them yourself.

14 Challenges

Obviously, C++ is not perfect. For the future, we face several challenges:

- How to make programmers prefer modern C++ styles over low-level (C-style) code, which is far more error-prone and harder to maintain, yet no more efficient.
- How to make C++ a better language given the Draconian constraints of C and C++ compatibility.
- How to improve and complete the techniques and models (incompletely and imperfectly) embodied in C++.

In particular, I would like to:

- Close more type loopholes (in particular, find a way to prevent misuses of **delete** without spoiling RAII)
- Simplify concurrent programming (in particular, provide some higher-level concurrency models as libraries)
- Simplify generic programming (in particular, introduce simple and effective concepts)
- Simplify programming using class hierarchies (in particular, eliminate use of the visitor pattern)
- Provide better support for combinations of object-oriented and generic programming styles.
- Make exceptions usable for hard-real-time projects (that will most likely be a tool rather than a language change)
- Find a good way of using multiple address spaces (as needed for distributed computing); this would most likely involve defining a more general module mechanism that would also address dynamic linking, and more.
- Provide many more domain-specific libraries
- Develop a more precise and formal specification of C++ (e.g. see [19,8,7])

Inside C++ is a smaller, cleaner, and even more powerful language struggling to get out. And no, that language is not C, C#, D, Haskell, Java, ML, Lisp, Scala, Smalltalk, or whatever. Whatever that language is, it must be better than C++ at light-weight abstraction in even the most demanding infrastructure applications.

References

1. ISO/IEC JTC1 SC22 WG21 N3092: Programming Languages — C++
2. Stroustrup, B.: The C++ Programming Language (Special Edition). Addison Wesley, Reading (2000) ISBN 0-201-70073-5
3. Stroustrup, B.: Programming – Principles and Practice Using C++. Addison-Wesley (December 2008) ISBN 978-0321543721
4. Stroustrup, B.: The C++11 FAQ, `http://www.research.att.com/~bs/C++11FAQ.html`

5. Stroustrup, B.: Abstraction and the C++ Machine Model. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 1–13. Springer, Heidelberg (2005)
6. Stroustrup, B.: The Design and Evolution of C++. Addison Wesley (March 1994) ISBN 0-201-54330-3
7. Ramananandro, T., Dos Reis, G., Leroy, X.: A Mechanized Semantics for C++ Object Construction and Destruction with Applications to Resource Management. In: POPL 2012, Philadelphia (Pennsylvania), USA (January 2012)
8. Dos Reis, G., Stroustrup, B.: General Constant Expressions for System Programming Languages. In: The 25th ACM Symposium On Applied Computing, SAC 2010 (March 2010)
9. Stroustrup, B.: Software Development for Infrastructure. IEEE Computer (January 2012)
10. Barron, D.W., et al.: The main features of CPL. The Computer Journal 6(2), 134 (1963)
11. Velthuizen, T.L.: C++ Templates are Turing Complete. University of Indiana Technical Report (2003)
12. Veldhuizen, T.: Using C++ template metaprograms. C++ Report 7(4) (May 1995)
13. Gregor, D., Jarvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: OOPSLA 2006 (October 2006)
14. Dos Reis, G., Stroustrup, B.: Specifying C++ Concepts. In: POPL 2006 (January 2006)
15. Sutton, A., Stroustrup, B.: Design of Concept Libraries for C++. In: Proc. International Conference on Software Language Engineering, SLE 2011 (July 2011)
16. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional, June 19 (2009) ISBN-13: 978-0321635372
17. Stroustrup, B., Sutton, A. (eds.): A Concept Design for the STL. WG21 Technical Report N3351=12-0041 (January 2012)
18. Williams, A.: C++ Concurrency in Action – Practical Multithreading. Manning Publications (2012) ISBN: 1933988770
19. Dos Reis, G., Stroustrup, B.: A formalism for C++. N1885 (October 2005)