An Overview of C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1 Introduction

C++ is a general purpose programming language[3] designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C language[2]. C++ was designed to

[1] be a better C.
[2] support data abstraction.
[3] support object-oriented programming.

This paper describes the features added to C to achieve this. In addition to C, the main influences of the design of C++ were Simula67[1] and Algol68[4].

C++ has been in use for about four years and has been applied to most branches of systems programming including compiler construction, data base management, graphics, image processing, music synthesis, networking, numerical software, programming environments, robotics, simulation, and switching. It has a highly portable implementation and there are now at least 1500 installations including AT&T 3B, DEC VAX, Intel 80286, Motorola 68000, and Amdahl machines running UNIX† and other operating systems*.

2 What is Good about C?

C is clearly not the cleanest language ever designed nor the easiest to use so why do so many people use it?

[1] C is flexible: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.
[2] C is efficient: The semantics of C are "low level"; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.
[3] C is available: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
[4] C is portable: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.

Compared with these "first order" advantages, the "second order" drawbacks like the curious C

† UNIX is a Trademark of AT&T Bell Laboratories.
* C++ is available from AT&T, Software Sales and Marketing, PO Box Box 25000, Greensboro, NC 27420, USA (telephone 800-828-UNIX) or from your local sales organization for the UNIX system.

declarator syntax and the lack of safety of some language constructs become less important. Designing "a better C" implies compensating for the major problems involved in writing, debugging, and maintaining C programs *without compromising the advantages of C*. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language "from scratch" does not ensure perfection and the C++ compilers compare favorably in run-time, have better error detection and reporting, and equal the C compilers in code quality.

3 A Better C

The first aim of C++ is to be "a better C" by providing better support for the styles of programming for which C is most commonly used. This primarily involves providing features that make the most common errors unlikely (since C++ is a superset of C such errors cannot simply be made impossible).

Argument Type Checking and Coercion

The most common error in C programs is a mismatch between the type of a function argument and the type of the argument expected by the called function. For example:

```
double sqrt(a) double a;
{
    /* ... */
}

double sq2 = sqrt(2);
```

Since C does not check the type of the argument 2, the call `sqrt(2)` will typically cause a run time error or give a wrong result when the square root function tries to use the integer 2 as a double precision floating point number. In C++, this program will cause no problem since 2 will be converted to a floating point number at the point of the call. That is, `sqrt(2)` is equivalent to `sqrt((double)2)`.

Where an argument type does not match the argument type specified in the function declaration and no type conversion is defined the compiler issues an error message. For example, in C++ `sqrt("Hello")` causes a compile time error.

Naturally, the C++ syntax also allows the type of arguments to be specified in function declarations:

```
double sqrt(double);
```

and a matching function definition syntax is also introduced:

```
double sqrt(double d)
{
    // ...
}
```

Inline Functions

Most C programs rely on macros to avoid function call overhead for small frequently-called operations. Unfortunately the semantics of macros are very different from the semantics of functions so the use of macros has many pitfalls. For example:

```
#define mul(a,b) a*b
int z = mul(x*3+2,y/4);
```

Here `z` will be wrong since the macro will expand to `x*3+2*y/4`. Furthermore, C macro definitions do not follow the syntactic rules of C declarations, nor do macro names follow the usual C scope rules. C++ circumvents such problems by allowing the programmer to declare `inline` functions:

```
inline int mul(int a, int b) { return a*b; }
```

An inline function has the same semantics as a "normal" function but the compiler can typically inline expand it so that the code-space and run-time efficiency of macros are achieved.

Scoped and Typed Constants

Since C does not have a concept of a symbolic constant macros are used. For example:

```
#define TBLMAX (TBLSIZE-1)
```

Such "constant macros" are neither scoped nor typed and can (if not properly parenthesized) cause problems similar to those of other macros. Furthermore, they must be evaluated each time they are used and their names are "lost" in the macro expansion phase of the compilation and consequently are not known to symbolic debuggers and other tools. In C++ constants of any type can be declared:

```
const int TBLMAX = TBLSIZE-1;
```

Varying Numbers of Arguments

Functions taking varying numbers of arguments and functions accepting arguments of different types are common in C. They are a notable source of both convenience and errors.

C functions where the type of arguments or the number of arguments (but not both) can vary, can be handled in a simple and type-secure manner in C++. For example, a function taking one, two, or three arguments of known type can be handled by supplying default argument values which the compiler uses when the programmer leaves out arguments. For example:

```
void print(char*, char* = "-", char* = "-");

print("one", "two", "three");
print("one", "two");  // that is, print("one", "two", "-");
print("one");        // that is, print("one", "-", "-");
```

Some C functions take arguments of varying types to provide a common name for functions performing similar operations on objects of different types. This can be handled in C++ by overloading a function name. That is, the same name can be used for two functions provided the argument types are sufficiently different to enable the compiler to "pick the right one" for each call. For example:

```
overload print;
void print(int);
void print(char*);

print(1);      // integer print function
print("two"); // string print function
```

The most general examples of C functions with varying arguments cannot be handled in a type-secure manner. Consider the standard output function `printf`, which takes a format string followed by an arbitrary collection of arguments supposedly matching the format string†:

```
printf("a string");
printf("x = %d\n",x);
printf("name: %s\n size: %d\n", obj.name, obj.size);
```

However, in C++ one can specify the type of initial arguments and leave the number and type of the remaining arguments unspecified. For example, `printf` and its variants can be declared like this:

† A C++ I/O system that avoids the type insecurity of the `printf` approach is described in reference 3.

```
int printf(const char* ...);
int fprintf(FILE*, const char* ...);
int sprintf(char*, const char* ...);
```

These declarations allows the compiler to catch errors such as

```
printf(stderr,"x = %d\n",x);    // error: printf does not take a FILE*
fprintf("x = %d\n",x);          // error: fprintf needs a FILE*
```

Declarations as Statements

Uninitialized variables are another common source of errors. One cause of this class of errors is the requirement of the C syntax that declarations can occur only at the beginning of a block (before the first statement). In C++, a declaration is considered a kind of statement and can consequently be placed anywhere. It is often convenient to place the declaration where it is first needed so that it can be initialized immediately. For example:

```
void some_function(char* p)
{
    if (p==0) error("p==0 in some_function");
    int length = strlen(p);
    // ...
}
```

4 Support for Data Abstraction

C++ provides support for data abstraction: the programmer can define types that can be used as conveniently as built-in types and in a similar manner. Arithmetic types such as rational and complex numbers are common examples:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r) { re=r; im=0; }   // float->complex conversion

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);   // binary minus
    friend complex operator-(complex);            // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
}
```

The declaration of class (that is, user-defined type) `complex` specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, `re` and `im` are accessible only to the functions defined in the declaration of class `complex`. Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re, a1.im+a2.im);
}
```

and used like this:

```
main()
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b+complex(1,2.3);
    // ...
}
```

Functions declared in a class declaration using the keyword `friend` are called *friend functions*. They do not differ from ordinary functions except that they may use private members of classes that name them friends. A function can be declared as a friend of more than one class. Other functions declared in a class declaration are called *member functions*. A member function is in the scope of the class and must be invoked for a specific object of that class.

Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation) instead of two separate operations. Such an initialization function is called a constructor. In cases where construction of objects of a type is non-trivial one often needs a complementary operation to clean up objects after their last use. In C++ such a cleanup function is called a destructor. Consider a vector type:

```
class vector {
    int  sz;          // number of elements
    int* v;           // pointer to integers
public:
    vector(int);      // constructor
    ~vector();        // destructor
    // ...
};
```

The `vector` constructor can be defined to allocate a suitable amount of space like this:

```
vector::vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];   // allocate an array of "s" integers
}
```

The cleanup done by the `vector` destructor consists of freeing the storage used to store the vector elements for re-use by the free store manager:

```
vector::~vector()
{
    delete v;         // deallocate the memory pointed to by v
}
```

Clearly C++ does not support garbage collection. This is, however, compensated for by enabling a type to maintain its own storage management without requiring intervention from a user.

Free Store Operators

The operators `new` and `delete` were introduced to provide a standard notation for free store allocation and deallocation. A user can provide alternatives to their default implementations by defining functions called `operator new` and `operator delete`. For built-in types the `new` and `delete` operators provides only a notational convenience (compared with the standard C functions `malloc()` and `free()`). For user-defined types such as `vector` the free store operators ensure

that constructors and destructors are called properly:

```
vector* fct1(int n)
{
    vector v(n);                // allocate a vector on the stack
                                // the constructor is called
    vector* p = new vector(n); // allocate a vector on the free store
                                // the constructor is called
    // ...
    return p;
    // the destructor is implicitly called for "v" here
}

void fct2()
{
    vector* pv = fct1(10);
    // ...
    delete pv; // call the destructor and free the store
}
```

References

C provides (only) "call by value" semantics for function argument passing; "call by reference" can be simulated by explicit use of pointers. This is sufficient, and often preferable to using "pass by value" for the built-in types of C. However, it can be inconvenient for larger objects† and can get seriously in the way of defining conventional notation for user-defined types in C++. Consequently, the concept of a reference is introduced. A reference acts as a name for an object; `T&` means reference to `T`. A reference must be initialized and becomes an alternative name for the object it is initialized with. For example:

```
int a = 1;   // "a" is an integer initialized to "1"
int& r = a;  // "r" is a reference initialized to "a"
```

The reference `r` and the integer `a` can now be used in the same way and with the same meaning. For example:

```
int b = r;   // "b" is initialized to the value of "r", that is, "1"
r = 2;       // the value of "r", that is, the value of "a" becomes "2"
```

References enable variables of types with "large representations" to be manipulated efficiently without explicit use of pointers. Constant references are particularly useful:

```
matrix operator+(const matrix&, const matrix&)
{
    // code here cannot modify the value of "a" or "b"
}

matrix a = b+c;
```

In such cases the "call by value" semantics are preserved while achieving the efficiency of "call by reference".

Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many, but not all, types. It can also be necessary to control all copy operations. Consider:

† as indicated by an inconsistency in the C semantics: arrays are always passed by reference.

```
vector v1(100);       // make v1 a vector of 100 elements
vector v2 = v1;       // make v2 a copy of v1
v1 = v2;              // assign v1 to v2 (that is, copy the elements)
```

Declaring a function with the name `operator=` in the declaration of class `vector` specifies that vector assignment is to be implemented by that function:

```
class vector {
    int* v;
    int  sz;
public:
    // ...
    void operator=(vector&);  // assignment
};
```

Assignment might be defined like this:

```
vector::operator=(vector& a) // check size and copy elements
{
    if (sz != a.sz) error("bad vector size for =");
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Since the assignment operation relies on the "old value" of the vector assigned to, it cannot be used to implement initialization of one vector with another. What is needed is a constructor that takes a vector argument:

```
class vector {
    // ...
    vector(int);       // create vector
    vector(vector&);   // create vector and copy elements
};

vector::vector(vector& a)    // initialize a vector from another vector
{
    sz = a.sz;               // same size
    v = new int[sz];         // allocate element array
    for (int i = 0; i<sz; i++) v[i] = a.v[i];    // same values
}
```

A constructor like this (of the form `X(X&)`) is used to handle all initialization. This includes arguments passed "by value" and function return values:

```
vector v2 = v1; // use vector(vector&) constructor to initialize

void f(vector);
f(v2);          // use vector(vector&) constructor to pass a copy of v2

vector g(int sz)
{
    vector v(sz);
    return v;   // use vector(vector&) constructor to return a copy of v
}
```

Operator Overloading

As demonstrated above, standard operators like `+`, `-`, `*`, `/` can be defined for user-defined types, as can assignment and initialization in its various guises. In general, all the standard operators with the exception of

```
->   .   ,   ?:
```

can be overloaded. The subscripting operator `[]` and the function application operator `()` have

proven particularly useful. The C "operator assignment" operators, such as += and *=, have also found many uses.

It is not possible to redefine an operator when applied to built-in data types, to define new operators, or to redefine the precedence of operators.

Coercions

User-defined coercions, like the one from floating point numbers to complex numbers implied by the constructor `complex(double)`, have proven unexpectedly useful in C++. Such coercions can be applied explicitly or the programmer can rely on the compiler adding them implicitly where necessary and unambiguous:

```
complex a = complex(1);
complex b = 1;              // implicit: 1 -> complex(1)
a = b+complex(2);
a = b+2;                    // implicit: 2 -> complex(2)
a = 2+b;                    // implicit: 2 -> complex(2)
```

Coercions were introduced into C++ because mixed mode arithmetic is the norm in languages used for numerical work and because most user-defined types used for "calculation" (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

Great care is taken (by the compiler) to apply user-defined conversions only where a unique conversion exists. Ambiguities caused by conversions are compile time errors.

It is also possible to define a conversion to a type without modifying the declaration of that type. For example:

```
class point {
    float dist;
    float angle;
public:
    // ...
    operator complex()  // convert point to complex number
    {
        return polar(dist,angle);
    }
    operator double()   // convert point to real number
    {
        if (angle) error("cannot convert point to real: angle!=0");
        return dist;
    }
};
```

These conversions could be used like this:

```
void some_function(point a)
{
    complex z = a;      // z = a.operator complex()
    double d = a;       // d = a.operator double()
    complex z3 = a+3;   // z3 = a.operator complex() + complex(3);
    // ...
}
```

This is particularly useful for defining conversions to built-in types since there is no declaration for a built-in type for the programmer to modify. It is also essential for defining conversions to "standard" types where a change may have (unintentionally) wide ranging ramifications and where the average programmer has no access to the declaration.

5 Support for Object-Oriented Programming

C++ provides support for object-oriented programming: the programmer can define class hierarchies and a call of a member function can depend on the actual type of an object (even where the actual type is unknown at compile time). That is, the mechanism that handles member function calls handles the case where it is known at compile time that an object belongs to *some* class in a hierarchy, but exactly *which* class can only be determined at run time. See examples below.

Derived Classes

C++ provides a mechanism for expressing commonality among different types by explicitly defining a class to be part of another. This allows re-use of classes without modification of existing classes and without replication of code. For example, given a class `vector`:

```
class vector {
    // ...
public:
    // ...
    vector(int);
    int& operator[](int); // overload the subscripting operator: []
}
```

one might define vector for which a user can define the index bounds:

```
class vec : public vector {
    int low, high;
public:
    vec(int, int);
    int& operator[](int);
};
```

Defining `vec` as

```
: public vector
```

means that first of all a `vec` is a `vector`. That is, type `vec` has ("inherits") all the properties of type `vector` in addition to the ones declared specifically for it. Class `vector` is said to be the *base* class for `vec`, and conversely `vec` is said to be *derived* from `vector`.

Class `vec` modifies class `vector` by providing a different constructor, requiring the user to specify the two index bounds rather than the size, and by providing its own access function `operator[]()`. A `vec`'s `operator[]()` is easily expressed in terms of `vector`'s `operator[]()`:

```
int& vec::operator[](int i)
{
    return vector::operator[](i-low);
}
```

The scope resolution operator `::` is used to avoid getting caught in an infinite recursion by calling `vec::operator[]()` from itself. Note that `vec::operator[]()` *had* to use a function like `vector::operator[]()` to access elements. It could not just use `vector`'s members `v` and `sz` directly since they were declared *private* and therefore accessible only to `vector`'s member functions.

The constructor for `vec` can be written like this:

```
vec::vec(int lb, int hb) : (hb-lb+1)
{
    if (hb-lb<0) hb = lb;
    low = lb;
    high = hb;
}
```

The construct `:(hb-lb+1)` is used to specify the argument list needed for the base class constructor `vector()`.

Class vec can be used like this:

```
void some_function(int l, int h)
{
    vec v1(l,h);
    const int sz = h-l+1;
    vector v2(sz);
    // ...
    for (int i = 0; i<sz; i++) v2[i] = v1[l+i]; // copy elements explicitly
    v2 = v1; // copy elements by using vector::operator=()
}
```

Virtual Functions

Class derivation (often called subclassing) is a powerful tool in its own right but a facility for run-time type resolution is needed to support object-oriented programming.

Consider defining a type shape for use in a graphics system. The system has to support circles, triangles, squares, and many other shapes. First specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked *virtual* (the Simula67 and C++ term for "to be defined later in a class derived from this one"). Given this definition one can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

For each shape v[i], the proper rotate function for the actual type of the object will be called. That "actual type" is not known at compile time.

To define a particular shape we must say that it is a shape (that is, derive it from class shape) and specify its particular properties (including the virtual functions):

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

In many contexts it is important that the C++ virtual function mechanism is very nearly as efficient as a "normal" function call. The additional run-time overhead is about 4 memory references (dependent on the machine architecture and the compiler) and the memory overhead is one word per object plus one word per virtual function per class.

Visibility Control

The basic scheme for separating the (public) user interface from the (private) implementation details has worked out very well for data abstraction uses of C++. It matches the idea that a type is a black box. It has proven to be less than ideal for object-oriented uses.

The problem is that a class defined to be part of a class hierarchy is not simply a black box. It is often primarily a building block for the design of other classes. In this case the simple binary choice *public/private* can be constraining. A third alternative is needed: a member should be private as far as functions outside the class hierarchy are concerned but accessible to member functions of a derived class in the same way that it is accessible to members of its own class. Such a member is said to be *protected*.

For example, consider a class `node` for some kind of tree:

```
class node {
    // private stuff
protected:
    node* left;
    node* right;
    // more protected stuff
public:
    virtual void print();
    // more public stuff
};
```

The pointers `left` and `right` are inaccessible to the general user but any member function of a class derived from class `node` can manipulate the tree without overhead or inconvenience.

The protection/hiding mechanism applies to names independently of whether a name refers to a function or a data member. This implies that one can have `private` and `protected` function members. Usually it is good policy to keep data `private` and present the `public` and `protected` interfaces as sets of functions. This policy minimizes the effect of changes to a class on its users and consequently maximizes its implementor's freedom to make changes.

Another refinement of the basic inheritance scheme is that it is possible to inherit public members of a base class in such a way that they do not become public members of the derived class. This can be used to provide restricted interfaces to standard classes. For example:

```
class dequeue {
    // ...
    void insert(elem*);
    void append(elem*);
    elem* remove();
};
```

Given a dequeue a `stack` can be defined as a derived class where only the `insert()` and `remove()` operations are defined:

```
class stack : dequeue { // note: just ``:'' not ``: public''
                        // members of dequeue are private members of stack
public:
    dequeue::insert;    // make "insert" a public member of stack
    dequeue::remove;    // make "remove" a public member of stack
};
```

Alternatively, inline functions can be defined to give these operations the conventional names:

```
class stack : dequeue {
public:
    void push(elem* ee) { dequeue::insert(ee); }
    elem* pop() { return dequeue::remove(); }
};
```

6 What is Missing?

C++ was designed under severe constraints of compatibility, internal consistency, and efficiency: no feature was included that

[1] would cause a serious incompatibility with C at the source or linker levels.
[2] would cause run-time or space overheads for a program that did not use it.
[3] would increase run-time or space requirements for a C program.
[4] would significantly increase the compile time compared with C.
[5] could only be implemented by making requirements of the programming environment (linker, loader, etc.) that could not be simply and efficiently implemented in a traditional C programming environment.

Features that might have been provided but weren't because of these criteria include garbage collection, parameterized classes, exceptions, multiple inheritance, support for concurrency, and integration of the language with a programming environment. Not all of these possible extensions would actually be appropriate for C++ and unless great constraint is exercised when selecting and designing features for a language a large, unwieldy, and inefficient mess will result. The severe constraints on the design of C++ have probably been beneficial and will continue to guide the evolution of C++.

7 Conclusions

C++ has succeeded in providing greatly improved support for traditional C-style programming without added overhead. In addition, C++ provides sufficient language support for data abstraction and object-oriented programming in demanding (both in terms of machine utilization and application complexity) real-life applications. C++ continues to evolve to meet demands of new application areas. There still appears to be ample scope for improvement even given the (self imposed) Draconian criteria for compatibility, consistency, and efficiency. However, currently the most active areas of development are not the language itself but libraries and support tools in the programming environment.

8 Acknowledgements

C++ could never have matured without the constant help and constructive criticism of my colleagues and users; notably Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro. Brian Kernighan and Andy Koenig made many helpful comments on drafts of this paper.

9 References

[1] Birtwistle, Graham et.al.
SIMULA BEGIN
Studentlitteratur, Lund, Sweden. 1973.

[2] Kernighan, B.W. and Ritchie, D.M.
The C Programming Language
Prentice-Hall, 1978.

[3] Stroustrup, Bjarne
The C++ Programming Language
Addison-Wesley, 1986.

[4] Woodward, P.M. and Bond, S.G.
Algol 68-R Users Guide
Her Majesty's Stationery Office, London. 1974.