Data Abstraction in C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ is a superset of the C programming language; it is fully implemented and has been used for non-trivial projects. The facilities for data abstraction provided in C++ are described. These include Simula-like classes providing (optional) data hiding, (optional) guaranteed initialization of data structures, (optional) implicit type conversion for user defined types, and (optional) dynamic typing; mechanisms for overloading function names and operators; and mechanisms for user-controlled memory management. It is shown how a new data type, like complex numbers, can be implemented, and how an "object-based" graphics package can be structured. A program using these data abstraction facilities is at least as efficient as an equivalent program not using them, and the compiler is faster than older C compilers.

Introduction

The aim of this paper is to show how to write C++ programs using "data abstraction" as described below†. This paper presents some general discussion of each new language feature to help the reader to understand where that feature fit in the overall design of the language, which programming techniques it is intended to support, and what kind of errors and costs it is intended to help the programmer to avoid. However, the paper is not a reference manual, so it does give not complete details of the language primitives; these can be found in reference 9.

C++ evolved from C[5] through some intermediate stages, collectively known as "C with classes"[8,9]. The primary influence on the design of the abstraction facilities was the Simula67 class concept[1,2]. The intent was to create data abstraction facilities which are both expressive enough to be of significant help in structuring large systems, and at the same time useful in areas where C's terseness and ability to express low level detail are great assets. Consequently, while C++ classes provide general and flexible structuring mechanisms, great care has been taken to ensure that their use does not cause run time or storage overhead which could have been avoided in C.

Except for details like introduction of new keywords, C++ is a superset of C; see section "Implementation and Compatibility" below. The language is fully implemented and in use. Tens of thousands of lines of code have been written and tested by dozens of programmers.

The paper falls into three main sections:

[1] A brief presentation of the idea of data abstraction.

[2] The bulk of the paper describes the facilities provided for the support of that idea through the presentation of small examples. This in itself falls into three sections:

[a] Basic techniques for data hiding, access to data, allocation, and initialization. Classes, class member functions, constructors, and function name overloading are presented. (Starts with section "Restriction of Access to Data").

[b] Mechanisms and techniques for creating new types with associated operators. Operator overloading, user defined type conversion, references, and free store operators are presented. (Starts with section "Operator Overloading and Type Conversion").

[c] Mechanisms for creating abstraction hierarchies, for dynamic typing of objects, and for creating polymorphic classes and functions. Derived classes and virtual functions are presented. (Starts with section "Derived Classes").

Sections [b] and [c] do not depend directly on each other.

[3] Finally some general observations on programming techniques, on language implementation, on efficiency, on compatibility with C, and on other languages are offered. (Starts with section "Input and Output").

A few sections are marked as "digressions"; they contain information that, while important to a programmer, and hopefully of interest to the general reader, does not directly relate to data abstraction.

Data Abstraction

"Data abstraction" is a popular, but generally ill-defined, technique for programming. The fundamental idea is to separate the incidental details of the implementation of a sub-program from the properties essential to the correct use of it. Such a separation can be expressed by channeling all use of the sub-program through a specific "interface". Typically the interface is the set of functions that may access the data structures which provide the representation of the "abstraction". One reason for the lack of a generally accepted definition is that any language facility supporting it will emphasize some aspects of the fundamental idea at the expense of others. For example:

† Note on the name C++: ++ is the C increment operator; when applied to a variable (typically a vector index or a pointer) it increments the variable so that it denotes the succeeding element. The name C++ was coined by Rich Mascitti. The slightly shorter name C+ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C, but the latter is not an acceptable name. The language is not called D, since it is an extension of C and does not attempt to remedy problems inherent in the basic structure of C. The name C++ signifies the evolutionary nature of the changes from old C. For yet another interpretation of the name C++ see the appendix of reference 7.

[1] Data hiding
Facilities for specifying interfaces that prevent corruption of data and relieve a user from the need to know about implementation details.

[2] Interface tailoring
Facilities for specifying interfaces that support and enforce particular conventions for the use of abstractions. Examples include operator overloading and dynamic typing.

[3] Instantiation
Facilities for creating and initializing of one or more "instances" (variables, objects, copies, versions) of an abstraction.

[4] Locality
Facilities for simplifying the implementation of an abstraction by taking advantage of the fact that all access is channeled through its interface. Examples include simplified scope rules and calling conventions within an implementation.

[5] Programming Environment
Facilities for supporting the construction of programs using abstractions. Examples include loaders which understand abstractions, libraries of abstractions, and debuggers that allow the programmer to work in terms of abstractions.

[6] Efficiency
To be useful, a language facility must be "efficient enough". The intended range of applications is therefore a major factor in determining which facilities can be provided in a language. Conversely, the efficiency of the facilities determine how freely they can be used in a given program. Efficiency must be considered in three separate contexts: compile time, link time, and run time.

The emphasis in the design of the C++ data abstraction facility was on 2, 3, and 6, that is, on facilities enabling a programmer to provide elegant and efficient interfaces to abstractions. In C++, data abstraction is supported by enabling the programmer to define new types, called "classes". The members of a class cannot be accessed, except in an explicitly declared set of functions. Simple data hiding can be achieved like this:

```
class data_type {
        /* data declarations */
        /* list of functions that may use the data declarations (``friends'') */
};
```

where only the "friends" can access the representation of variables of class data_type as defined by the data declarations. Alternatively, and often more elegantly, one can define a data type where the set of functions that may access the representation is an integral part of the type itself:

```
class object_type {
        /* declarations used to implement object_type */
public:
        /* declarations specifying the interface to object_type */
};
```

One obvious, but non-trivial, aim of many modern language designs is to enable programmers to define "abstract data types" with properties similar to the properties of the fundamental data types of the languages. Below it will be shown how to add a data type complex to the C++ language, so that the usual arithmetic operators can be applied to complex variables. For example:

```
complex a, x, y, z;
a = x/y + 3*z;
```

The idea of treating an object as a black box is further supported by a mechanism for hierarchically constructing classes out of other classes. For example:

```
class shape { ... };
class circle : shape { ... };
```

The class circle can be used as a simple shape in addition to being used as a circle. Class circle is said to be a derived class with class shape as its base class. It is possible to leave the resolution of the type of objects sharing common base classes to run time. This allows objects of different types to be manipulated in a uniform manner.

Restriction of Access to Data

Consider a simple old C fragment†, outlining an implementation of the concept of a date:

```
struct date { int day, month, year; };
struct date today;
extern void set_date(), next_date(), next_today(), print_date();
```

There are no explicit connections between the functions and the data type, and no indication that these functions should be the only ones to access the members of the structure `date`. It ought to be possible to state such an intent.

A simple way of doing this is to declare a data type that can only be manipulated by a specific set of functions. For example:

```
class date {
        int day, month, year;
        friend void set_date(date*, int, int, int),
                    next_date(date*),
                    next_today(),
                    print_date(date*);
};
```

The keyword `class` indicates that only functions mentioned as "friends" in the declaration can use the class member names `day`, `month`, and `year`; otherwise a `class` behaves like a traditional C `struct`. That is, the class declaration itself defines a new type of which variables can be declared. For example:

```
date my_birthday, today;

set_date(&my_birthday,30,12,1950);
set_date(&today,23,6,1983);
print_date(&today);
next_date(&today);
```

Friend functions are defined in the usual manner. For example:

```
void next_date(date* d)
{
        if ( ++d->day > 28 ) {
                /* do the hard part */
        }
}
```

This solution to the problem of data hiding is simple, and often quite effective. It is not perfectly flexible because it allows access by the "friends" to all variables of a type. For example, it is not possible to have a different set of friends for the dates `my_birthday` and `today`. A function can, however, be the friend of more than one class. The importance of this will be demonstrated below. There is no requirement that a friend should only manipulate variables passed to it as arguments. For example, the name of a global variable may be built into a function:

```
void next_today()
{
        if ( ++today.day > 28 ) {
                /* do the hard part */
        }
}
```

The protection of the data from functions that are not friends relies on restriction on the use of the

† The keyword `void` specifies that a function does not return a value. It was introduced into C about 1980.

class member names. It can therefore be circumvented by address manipulation and explicit type conversion.

There are several benefits to be obtained from restricting access to a data structure to an explicitly declared list of functions. Any error causing an illegal state of a `date` must be caused by code in the friend functions, so the first stage of debugging, localization, is completed before the program is even run. This is a special case of the general observation that any change to the behavior of the type `date` can and must be effected by changes to its friends. Another advantage is that a potential user of such a type need only examine the definition of the friends to learn to use it. Experience has amply demonstrated this.

Digression: Argument Types

The argument types of the functions above were declared. This could not have been done in old C; neither would the matching function definition syntax used for `next_date` have been accepted. In C++ the semantics of argument passing are identical to those of initialization. In particular, the usual arithmetic conversions are performed. A function declaration that does not specify an argument type, for example `next_today()`, specifies that the function does not accept any arguments†. The argument types of all declarations and the definition of a function must match exactly.

It is still possible to have functions which take an unspecified and possibly variable number of arguments of unspecified types, but such relaxation of the type checking must be explicitly declared. For example

```
int wild(...);
int fprintf(FILE*, char* ...);
```

The ellipsis specifies that any arguments (or none) will be accepted without any checking or conversion exactly as in old C. For example:

```
wild(); wild("asdf",10); wild(1.3,"ghjk",wild);
fprintf(stdout,"x=%d",10);
fprintf(stderr,"file %s line %d\n", file_name, line_no);
```

Note that the first two arguments of `fprintf` must be present and will be checked.

As ever, undeclared functions may be used and will be assumed to return integers. They must, however, be used consistently. For example:

```
undef1(1, "asdf");   undef1(2, "ghjk");   /* fine */
undef2(1, "asdf");   undef2("ghjk", 2);   /* error */
```

The inconsistent use of `undef2` is detected by the compiler.

Objects

The structure of a program using the `class/friend` mechanism to restrict access to the representation of a data type is exactly the same as the structure of a program not using it. This implies that no advantage has been taken of the new facility to make the functions implementing the operations on the type easier to write. For many types, a more elegant solution can be obtained by incorporating such functions into the new type itself. For example:

```
class date {
        int day, month, year;
public:
        void set(int, int, int);
        void next();
        void print();
};
```

Functions declared this way are called member functions and can be invoked only for a specific

† This is different from old C; see section "Implementation and Compatibility" below.

variable of the appropriate type using the standard C structure member syntax. Since the function names no longer are global they can be shorter:

```
my_birthday.print();
today.next();
```

On the other hand, to define a member function one must specify both the name of the function and the name of its class:

```
void date.next()
{
        if ( ++day > 28 ) {
                /* do the hard part */
        }
}
```

Variables of such types are often referred to as objects. The object for which the function is invoked constitutes a hidden argument to the function. In a member function, class member names can be used without explicit reference to a class object. In that case, like the use of `day` above, the name refers to that member of the object for which the function was invoked. A member function sometimes needs to refer explicitly to this object, for example to return a pointer to it. This is achieved by having the keyword `this` denote that object in every class function. Thus, in a member function `this->day` is equivalent to `day` for every member of the class `date`.

The `public` label separates the class body into two parts. The names in the first, "private", part can only be used by member functions (and friends). The second, "public", part constitutes the interface to objects of the class. A class function may access both public and private members of every object of its class, not just members of the one for which it was invoked.

The relative merits of friends and member functions will be discussed in section "Friends vs Members" after a larger body of examples has been presented. For now, it is sufficient to notice that a friend is not affected by the `public/private` mechanism and operates on objects in a standard and explicit manner. A member, on the other hand, must be invoked for an object and treats that object differently from all others.

Static Members

A class is a type, not a data object, and each object of the class has its own copy of the data members of the class. However, there are concepts (abstractions) which are best supported if the different objects of the class share some data. For example, to manage tasks in an operating system or a simulation a list of all tasks is often useful:

```
class task {
        ...
        task* next;
        static task* task_chain;
        void schedule(int);
        void wait(event);
        ...
};
```

Declaring the member `task_chain` as `static` ensures that there will only be one copy of it, not one copy per task object. It is still in the scope of class `task`, however, and can only be accessed from "the outside" if it was declared public. In that case its name must be qualified by its class name:

```
task::task_chain
```

In a member function it can be referred to as plain `task_chain`. The use of `static` class members can reduce the need for global variables considerably.

The operator `::` (colon colon) is used to specify the scope of a name in expressions. As a unary operator it denotes external (global) names. For example, if the task function `wait` in a simulator needs to call a non-member function `wait` it can be done like this:

```
void task.wait(event e)
{
        ...
        ::wait(e);
}
```

Constructors and Overloaded Functions

The use of functions like `set_date()` to provide initialization for class objects is inelegant and error prone. Since it is nowhere stated that an object must be initialized, a programmer can forget to do so or, often with equally disastrous results, do so twice. A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type it is called a constructor. A constructor is recognized by having the same name as the class itself. For example:

```
class date {
        ...
        date(int, int, int);
};
```

When a class has a constructor all objects of that class must be initialized:

```
date today = date(23, 6, 1983);
date xmas(25, 12, 0);           /* legal abbreviated form */
date july4 = today;
date my_birthday;               /* illegal, initializer missing */
```

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```
class date {
        ...
        date(int, int, int);    /* day month year */
        date(char*);            /* date in string representation */
        date(int);              /* day, assume current month and year */
        date();                 /* default date: today */
};
```

As long as the constructor functions differ in their argument types the compiler can select the correct one for each use:

```
date today(4);
date july4("July 4, 1983");
date guy("5 Nov");
date now;                        /* default initialized */
```

Constructors are not restricted to initialization, but can be used where ever it is meaningful to have a class object:

```
date us_date(int month, int day, int year)
{
        return date(day, month, year);
}
...
some_function( us_date(12,24,1983) );
some_function(    date(24,12,1983) );
```

When several functions are declared with the same name, that name is said to be overloaded. The use of overloaded function names is not restricted to constructors. However, for non-member functions the function declarations must be preceded by a declaration specifying that the name is to be overloaded. For example:

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

or possibly abbreviated like this:

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

As far as the compiler is concerned, the only thing common for a set of a set of functions of the same name is that name. Presumably they are in some sense similar, but the language does not constrain or aid the programmer. Thus, overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names like `sqrt`, `print`, and `open`. Where a name is semantically significant, as in the case of constructors, this convenience becomes essential. For example, consider writing a single constructor for class `date` above.

For arguments to functions with overloaded names the C++ type conversion rules do not apply fully. The conversions that may destroy information are not performed, leaving only *char->short->int->long*, *float->double*, and *int->double*. It is, however, possible to provide different functions for integral and floating types. For example:

```
overload print(int), print(double);
```

The list of functions for an overloaded name will be searched in order of appearance for a match, so that `print(1)` will invoke the integer print function, and `print(1.0)` the floating-point print function. Had the order of declaration been reversed both calls would have invoked the floating-point print function with the `double` representation of 1.

Operator Overloading and Type Conversion

Some languages provide a `complex` data type, so that programmers can use the mathematical notion of complex numbers directly. Since C++ does not, it is an obvious test of an abstraction facility to see to what extent the conventional notion of complex numbers can be supported†. The aim of the exercise is to be able to write code like this:

```
complex x;
complex a = complex(1, 1.23);
complex b = 1;
complex c = PI;

if (x!=a) x = a+log(b*c)/2;
```

That is, the standard arithmetic and comparison operators must be defined for complex numbers and for mixtures of complex and scalar constants and variables.

Here is a declaration of a very simple class `complex`:

```
class complex {
        double re, im;

        friend complex operator+  (complex, complex);
        friend complex operator*  (complex, complex);
        friend int     operator!= (complex, complex);
public:
        complex()                      { re=im=0; }
        complex(double r)              { re=r; im=0; }
        complex(double r, double i) { re=r; im=i; }
};
```

An operator is recognized as a function name when it is preceded by the keyword `operator`.

† Note, however, that `complex` is an unusual data type in that it has an extremely simple representation and there are very strong traditions for its proper use. It is therefore primarily a test of the abstraction facility's power to imitate conventional notation. In most other cases the designer's attention will be directed towards finding a good representation of the abstraction and towards finding a suitable way of presenting the abstraction to its users.

When an operator is used for a class type the compiler will generate a call to the appropriate function, if declared. For example, for complex variables xx and yy the addition xx+yy will be interpreted as operator+(xx,yy) given the declaration of class complex above. The complex add function could be defined like this:

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

Naturally, all names of the form operator@ are overloaded. To ensure that the language is only extendible and not mutable, an operator function must take at least one class object argument. By declaring operator functions the programmer can assign meaning to the standard C++ operators applied to objects of user specified data types. These operators retain their usual places in the C++ syntax, and it is not possible to add new operators. It is therefore not possible to introduce a unary plus operator, to change the precedence of an operator, or to introduce a new operator (for example, ** for exponentiation). This restriction keeps the analysis of C++ expressions simple.

Declarations of functions for unary and binary operators are distinguished by their number of arguments. For example:

```
class complex {
        ...
        friend complex operator-(complex);              /* unary  */
        friend complex operator-(complex, complex);     /* binary */
};
```

There are three ways the designer of class complex could decide to handle mixed-mode arithmetic, like xx+1, where xx is a complex variable. It can simply be considered illegal, so that the user has to write the conversion from double to complex explicitly: xx+complex(1). Alternatively, several complex add functions may be specified:

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
```

so that the compiler will choose the appropriate function for each call. Finally, if a class has constructors that take a single argument then they will be taken to define conversions from their argument type to the type they construct values for. Thus, with the declaration of class complex above xx+1 would automatically be interpreted as operator+(xx,complex(1)).

This last alternative violates many people's idea of strong typing. However, using the second solution will nearly triple the number of functions needed and the first provides little notational convenience to the user of class complex. Note that complex numbers are typical with respect to the desirability of mixed-mode arithmetic. A typical data type does not exist in a vacuum. Furthermore, for many types there exists a trivial mapping from the C++ numeric and/or string constants into a subset of the values of the type (similar to the mapping of the C++ numeric constants into the complex values on the real axis).

The friend approach was chosen in favor of using member functions for the operator functions. The inherent asymmetry in the notion of objects does not match the traditional mathematical view of complex numbers.

Digression: Default Arguments and Inline Functions

Class complex had three constructors, two of which simply provided the default value zero for notational convenience of the programmer. This use of overloading is typical for constructors, and has also been found to be quite common for other functions. However, overloading is a quite elaborate and indirect way of providing default argument values and, in particular for more complicated constructors, quite verbose. Consequently, an facility for expressing default arguments directly is provided. For example:

```
class complex {
        ...
public:
        complex(double r = 0, double i = 0) { re=r; im=i; }
};
```

When a trailing argument is missing the default constant expression can be used. For example:

```
complex a(1,2);
complex b(1);       /* b = complex(1,0) */
complex c;          /* c = complex(0,0) */
```

When a member function, like `complex` above, is not only declared, but also defined (that is, its body is presented) in a class declaration it may be inline substituted when called, thus eliminating the usual function call overhead. An inline substituted function is not a macro; its semantics are identical to other functions. Any function can be declared inline by preceding its definition by the keyword `inline`. Inline functions can make class declarations quite untidy, they will only improve run-time efficiency if used judicially, and will always increase the time and space needed to compile a program. They should therefore be used only when a significant improvement of run-time is expected. They are included in C++ because of experience with C macros. Macros are sometimes essential for an application (and it is not possible to have a class member macro), but more often they create chaos by appearing to be functions without obeying the syntax, scope, and argument passing rules of functions.

Storage Management

There are three storage classes in C++: static, automatic (stack), and free (dynamic). Free store is managed by the programmer through the operators `new` and `delete`. No standard garbage collector is provided†.

Constructors are handy for hiding details of free store management. For example:

```
class string {
        char* rep;
        string(char*);
        ~string()       { delete rep; }
        ...
};

string.string(char* p)
{
        rep = new char[strlen(p)+1];
        strcpy(rep,p);
}
```

Here the use of free store is encapsulated in the constructor `string()` and its inverse, the destructor `~string()`. Destructors are implicitly called when an object goes out of scope. They are also called when an object is explicitly deleted by `delete`, but never for static objects. The `new` operator takes a type as its argument and returns a pointer to an object of that type; `delete` takes such a pointer as argument. A `string` may itself be allocated on the free store. For example:

```
string* p = new string("asdf");
delete p;
p = new string("qwerty");
```

It is furthermore possible for a class to take over the free store management for its objects. For

† It is, however, not that difficult to write a garbage collecting implementation of the `new` operator, as has been done for the C free store allocator function `malloc()`. It is not in general possible to distinguish pointers from other data items when looking at the memory of a running C++ program, so a garbage collector must be conservative in its choice of what to delete, and it must examine unappealingly large amounts of data. They have been found useful for some applications, though.

example:

```
class node {
        int type;
        node* l;
        node* r;
        node()  { if (this==0) this = new_node(); }
        ~node() { free_node(this); this = 0; }
        ...
};
```

For an object created by new, the this pointer will be zero when a constructor is entered. If the constructor does not assign to this the standard allocator function is used. The standard deallocator function will be used at the end of a destructor if and only if this is non-zero. An allocator provided by the programmer for a specific class or set of classes can be much simpler and at least an order of magnitude faster than the standard allocator.

Using constructors and destructors the designer may specify data types, like string above, where the size of the representation of an object can vary, even though the size of every static and automatic variable must be known at load time and compile time, respectively. The class object itself is of fixed size, but its class maintains a variable sized secondary data structure.

Hiding Storage Management

Constructors and destructors cannot completely hide storage management details from the user of a class. When an object is copied, either by explicit assignment or by passing it as a function argument, the pointers to secondary data structures are copied too. This is sometimes undesirable. Consider the problem of providing value semantics for a simple data type string. A user sees a string as a single object, but the implementation consists of two parts as outlined above. After the assignment s1=s2 both strings refer to the same representation, and the store used for the old representation of s1 is unreferenced. To avoid this the assignment operator can be overloaded.

```
class string {
        char* rep;
        void operator=(string);
        ...
};

void string.operator=(string source)
{
        if (rep != source.rep) {
                delete rep;
                rep = new char[ strlen(source.rep)+1 ];
                strcpy(rep,source.rep);
        }
}
```

Since the function needs to modify the target string it is best written as a member function taking the source string as argument. The assignment s1=s2 will now be interpreted as s1.operator=(s2).

This leaves the problem of what to do with initializers and function arguments. Consider

```
string s1 = "asdf";
string s2 = s1;
do_something(s2);
```

This leaves the strings s1, s2, and the argument of do_something with the same rep. The standard bitwise copy clearly does not preserve the desired value semantics for strings.

The semantics of argument passing and initialization are identical; both involve copying an object into an uninitialized variable. They differ from the semantics of assignment (only) in that an object assigned to is assumed to contain a value, and an object being initialized is not. In particular, a constructors are used in argument passing exactly as in initialization. Consequently, the undesirable bitwise copy can be avoided if we can specify a constructor to perform the proper copy operation. Unfortunately, using the obvious constructor

```
class string {
        ...
        string(string);
}
```

leads to infinite recursion. It is therefore illegal. To solve this problem a new type "reference" is introduced. It is syntactically identified by the declarator & which is used in the same way as the pointer declarator *. When a variable is declared to be a T&, that is a reference to T, it can be initialized either by a pointer to type T or an object of type T. In the latter case the address-of operator & is implicitly applied. For example

```
int x;
int& r1 = &x;
int& r2 = x;
```

assigns the address of x to both r1 and r2. When used a reference is implicitly dereferenced, so for example:

```
r1 = r2
```

means copy the object pointed to by r2 into the object pointed to by r1. Note that initialization of a reference is quite different from assignment to it.

Using references class string can now be declared like this:

```
class string {
        char* rep;
        string(char*);
        string(string&);
        ~string();
        void operator=(string&);
        ...
};

string(string& source)
{
        rep= new char[ strlen(source.rep)+1 ];
        strcpy(rep,source.rep);
}
```

Initialization of one string with another (and passing a string as an argument) will now involve a call of the constructor string(string&) that will correctly duplicate the representation. The string assignment operator was redeclared to take advantage of references. For example:

```
void string.operator=(string& source)
{
        if (this != &source) {
                delete rep;
                rep = new char[ strlen(source.rep)+1 ];
                strcpy(rep,source.rep);
        }
}
```

This type string will not be efficient enough for many applications. It is, however, not difficult to modify it so that the representation is only copied when necessary and shared otherwise.

Further Notational Convenience

It is curious that references, a facility with great similarity to the "call by reference" rules for argument passing in many languages, are introduced primarily to enable a programmer to specify "call by value" semantics for argument passing. They have several other uses as well, however, including of course "by reference" argument passing. In particular, references provide a way of having non-trivial expressions on the left-hand side of assignments. Consider a `string` type with a substring operator:

```
class string {
        ...
        void    operator=(string&);
        void    operator=(char*);
        string& operator()(int, int);  /* substring: (pos,length) */
};
```

where `operator()` denotes function application.

```
string s1 = "asdf";
string s2 = "ghjkl";
s1(1,2) = "xyz";        /* s1 == "axyzf" */
s2 = s1(0,3);           /* s2 == "axy"   */
```

The two assignments will be interpreted as:

```
( s1.operator()(1,2) )->operator=("xyz");
s2.operator=( s1.operator()(0,3) );
```

The `operator()` function need not know whether it is invoked on the left-hand or the right-hand side of the assignment. The `operator=` function can take care of that.

Vector element selection can be similarly overloaded by defining `operator[]`.

Digression: References and Type Conversion

Conversions defined for a class are applied even when references are involved. Consider a class `string` where assignment of simple character strings is not defined, but the construction of a string from such a character string is:

```
class string {
        ...
             string(char*);
        void operator=(string&);
};

string s = "asdf";
```

The assignment

```
s = "ghjk";
```

is legal, and will produce the desired effect. It is interpreted as

```
s.operator=( (temp.string("ghjk"),&temp) )
```

where `temp` is a temporary variable of type `string`. Applying constructors before taking the address as required by the reference semantics ensures that the expressive power provided by constructors is not lost for variables of reference type. In other words, the set of values accepted by a function expecting an argument of type `T` is the same as that accepted by a function expecting a `T&` (reference to `T`).

Derived Classes

Consider writing a system for managing geometric shapes on a terminal screen. An attractive approach is to treat each shape as an object that can be requested to perform certain actions like "rotate" and "change color". Each object will interpret such requests in accordance with its type. For example, the algorithm for rotation is likely to be different (simpler) for a circle than for a triangle. What is needed is a single interface to a variety of co-existing implementations. The different kind of shapes cannot be assumed to have similar representations. They may differ widely in complexity, and it would be a pity to be unable to utilize the inherent simplicity of basic shapes like circle and triangle because of the need to support complex shapes like "mouse" and "British

Isles".

The general approach is to provide a class `shape` defining the common properties of shapes, in particular a "standard interface". For example:

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

The functions that cannot be implemented without knowledge of the specific `shape` are declared `virtual`. A `virtual` function is expected to be defined later. At this stage only its type is known; this, however, is sufficient to check calls to it.

A class defining a particular shape may be defined like this:

```
class circle : public shape {
        float radius;
public:
        void  rotate(int angle) {}
        void  draw();
        ...
};
```

This specifies a `circle` to be a `shape`, and as such it has all the members of class `shape` in addition to its own members. The class `circle` is said to be derived from its "base class" `shape`. Circles can now be declared and used:

```
circle c1;
shape* sh;
point p(100,30);

c1.draw();
c1.move(p);
sh = &c1;
sh->draw();
```

Naturally the function called by `c1.draw()` is `circle::draw()`, and since `circle` did not define its own `move()`, the function called by `c1.move(p)` is `shape::move()`, which class `circle` inherited from class `shape`. However, the function called by `sh->draw()` is also `circle::draw()` despite the fact that no reference to class `circle` is found in the declaration of class `shape`. A virtual function is defined (or redefined) when a class is derived from its class. Each object of a class with virtual functions contains a type indicator. This enables the compiler to find the proper `virtual` function for a call even when the type of the object is not known at compile time. Calling a virtual function is the only way of using the hidden type indicator in a class (a class without virtual functions does not have such an indicator).

A shape may also provide facilities which cannot be used unless the programmer knows its particular type. For example:

```
class clock_face : public circle {
        line   hour_hand, minute_hand;
public:
        void   draw();
        void   rotate(int);
        void   set(int, int);
        void   advance(int);
        ...
}
```

The time displayed by the clock can be `set()` to a particular time, and one can `advance()` the displayed time a number of minutes. The `draw()` in `clock_face` hides `circle::draw()`, so that the latter can only be called by its full name. For example:

```
void clock_face.draw() {
        circle::draw();
        hour_hand.draw();
        minute_hand_draw();
}
```

Note that a virtual function must be a member. It cannot be a friend, and there is no equivalent in the `class/friend` style of programming to the use of dynamic typing presented here and in the following section.

Digression: Structures and Unions

The old C constructs `struct` and `union` are legal, but conceptually absorbed into classes. A `struct` is a class with all members public, that is

```
struct s { ... };
```

is equivalent to

```
class s { public: ... };
```

A `union` is a `struct` that can hold exactly one data member at a time.

These definitions imply that `struct` or a `union` can have function members. In particular they can have constructors. For example:

```
union uu {
        int   i;
        char* p;
        uu(int ii)    { i=ii; }
        uu(char* pp)  { p=pp; }
};
```

This takes care of most problems concerning initialization of unions. For example:

```
uu u1 = 1;
uu u2 = "asdf";
```

Polymorphic Functions

By using derived classes one can design interfaces providing uniform access to objects of unknown and/or different classes. This can be used to write polymorphic functions, that is functions where the algorithm is specified so that it will apply to a set of different argument types. For example:

```
void sort(object* v[], int size)
{
        /* sort the vector of objects ``v[size]'' */
}
```

The `sort` function need only be able to compare `objects` to perform its task. So, if class object has a virtual function cmpr(), sort() will be able to sort vectors of objects of any class derived from class object for which cmpr() is defined. For example:

```
class object {
        ...
        virtual int cmpr(object*);
};

class apple : public object {
        ...
        int key;
        int cmpr(object* arg)
        {       /* assume that arg is also an apple */
                int k = ((apple*)arg)->key;
                return (key==k) ? 0 : (key<k) ? -1 : 1;
        }

};

class orange : public object {
        ...
        int cmpr(object*);
};
```

The cmpr() function was preferred to the superficially more attractive approach of overloading the ``<'' operator because my favorite sort algorithm uses a three-way compare. To write a sort() to operate on a vector of objects, rather than on a vector of pointers to objects, a virtual "size" function would be needed.

Should it be desirable to compare an apple with an orange, some way for the comparison function to find its sort-key would be needed. Class object could, for example, contain a virtual sort-key extraction function.

Polymorphic Classes

Polymorphic classes can be constructed in the same way as polymorphic functions. For example:

```
class set : public object {
        class set_mem {
                set_mem* next;
                object*  mem;
                set_mem(object* m, set_mem* n) { mem=m; next=n; }
        } *tail;
public:
        int insert(object*);
        int remove(object*);
        int is_member(object*);
            set()  { tail = 0; }
            ~set() { if (tail) error(0,"non-empty set deleted"); }
};
```

That is, a set is implemented as a linked list of set_mem objects, each of which points to an object. Pointers to objects (not objects) are inserted. For completeness a set is itself an object so that you can create a set of sets. Since class set is implemented without relying on data in the member objects, an object can be member of two or more sets. This model is quite general and can be (and indeed has been) used to create "abstractions" like set, vector, linked_list, and table. The most distinctive feature of this model for "container classes" is that in general the container cannot rely on data stored in the contained objects nor can the contained objects rely on data identifying their container (or containers). This is often an important structural advantage; classes can be designed and used without concerns about what kind of data structures programs using them may need. Its most obvious disadvantage is that there is a minimum overhead of one pointer per member (two pointers in the linked list implementation of class `set` above)†. Another advantage is that such container classes are capable of holding heterogeneous collections of members. Where this is undesirable, it is trivial to derive a class which will accept only members of one particular class. For example:

```
class apple_set : public set {
public:
        int insert(apple* a)     { return set::insert(a); }
        int remove(apple* a)     { return set::remove(a); }
        int is_member(apple* a) { return set::is_member(a); }
};
```

Note that since the functions of class `apple_set` do not perform any actions in addition to those performed by the base class `set`, they will be optimized away. They serve only to provide compile time type checking.

Input and Output

C does not have special facilities for handling input and output. Traditionally the programmer relies on library functions like `printf()` and `scanf()`. For example, to print a data structure representing a complex number one might write:

```
printf("(%g,%g)\n", zz.real, zz.imag);
```

Unfortunately, since the C standard input/output functions know only the standard types it is necessary to print a structure member by member. This is often tedious and can only be done where the members are accessible. The paradigm cannot be cleanly and generally extended to handle user-defined types and input/output formats.

The approach taken in C++ is to provide (in a "standard" library, NOT in the language itself) the operator << ("put to") for a data type `ostream` and each basic and user-defined type. Given an output stream `cout` one can write

```
cout<<zz;
```

The implementor of class complex defines << for a complex number. For example:

```
ostream& operator<<(ostream& s, complex& c)
{
        return s <<"(" <<c.real <<"," <<c.imag <<")\n";
}
```

The << operator was chosen in preference to a function name to avoid the tedium of having to write a separate call for each argument. For example:

```
put(cout,"(");     /* intolerably verbose */
put(cout,c.real);
put(cout,",");
put(cout,c.imag);
put(cout,"\n");
```

There is a loss of control over the formatting of output when using << compared with using `printf`. Where such finer control is necessary, one can use "formatting functions". For example:

```
cout <<"hexadecimal x = " <<hex(x) <<"octal x = " <<oct(x);
```

where `hex()` and `oct()` return a string representation of their first argument.

Input is handled by providing the operator >> ("get from") for a data type `istream` and each basic and user-defined type. If an input operation fails the stream is put into an error state that will cause subsequent operations on it to fail. For a variable `zz` of any type one can write code like this

† plus another pointer for the implementation of the virtual function mechanism. See section "Efficiency" below.

```
while ( cin>>zz ) cout<<zz;
```

Surprisingly enough, the input operations are typically trivial to write, since there invariably is a constructor to do the non-trivial part of the job, and the arguments to the constructor(s) give a good first approximation of the input format. For example:

```
istream& operator>> (istream& s, complex& zz)
{
        if (!s) return s;
        double re = 0, im = 0;
        char c1 = 0, c2 = 0, c3 = 0;
        s >>c1 >>re >>c2 >>im >>c3;
        if (c1!='(' || c2!=',' || c3!=')') s.state = _bad;
        if (s) zz = complex(re,im);
        return s;
}
```

The convention for functions implementing the input and output operators is to return the argument stream and indicate success or failure in its state. This example is a bit too simple for real use, but it will change the value of its argument `zz` and return the stream in a non-error state iff a complex number of the form `(double,double)` was found. The interpretation of a test on a stream as a test on its state is handled by providing a conversion from `ostream` which examines `s.state`.

Note that there is no loss of type information when using << and >>, so, compared with the C `printf/scanf` paradigm, a large class of errors has been eliminated. Furthermore, << and >> can be defined for a new (user-defined) type without affecting the "standard" classes `istream` and `ostream` in any way, and without any knowledge of the internals of these classes. An `ostream` can be bound to a real output device (buffered or unbuffered) or simply to an in-core buffer. So can an `istream`. This extends the range of uses considerably and eliminates the need for the old C functions `sscanf` and `sprintf`.

Character level operations `put()` and `get()` are also available for i/o streams.

Friends vs Members

When a new operation are to be added to a class there are typically two ways it can be implemented, as a friend or as a member. Why are two alternatives provided, and for what kind of operations should each alternative be preferred?

A friend function is a perfectly ordinary function, distinguished only by its permission to use private member names. Programming using friends is essentially programming as if there were no data hiding. The friend approach cleanly implements the traditional mathematical view of values that can be used in computation, assigned to variables, but never really modified. This paradigm is then compromised by using pointer arguments.

A member function, on the other hand, is tied to a single class and invoked for one particular object. The member approach cleanly implements the idea of operations that change the state of an object, for example assignment. Because a single object is distinguished the language can take advantage of local knowledge to provide notational convenience, efficient implementation, and let the meaning of the operation depend on the value of that object. Note that it is not possible to have a virtual friend. Constructors, too, must be members.

As the first approximation, use a member to implement an operation if it might conceivably modify the state of an object. Note that type conversion, if declared, is performed on arguments, but not on the object for which a member is invoked. Consequently, the member implementation should also be chosen for operations where type conversion is undesirable.

A friend function can be the friend of two or more classes while a member function is a member of a single class. This makes it convenient to implement operations on two or more classes as friends. For example:

```
class matrix {
        friend matrix operator*(matrix, vector);
        ...
};

class vector {
        friend matrix operator*(matrix, vector);
        ...
};
```

It would take two members `matrix::operator*()` and `vector::operator*()` to achieve what the friend `operator*()` does.

The name of a friend is global while the scope of a member name is restricted to its class. When structuring a large program one tries to minimize the amount of global information, therefore friends should be avoided in the same way as global data is. Ideally, at this level, all data is encapsulated in classes and operated on using member functions. However, at a more detailed level of programming this becomes tedious and often inefficient; here friends come into their own.

Finally, if there is no obvious reason for preferring one implementation of an operation over another make that operation a member.

Separate Compilation

For separate compilation the traditional C approach has been retained. Type specifications are shared by textually including them in separately compiled source files. There is no automatic mechanism that ensures that the header files contain complete type specifications and that they are used consistently. Such checks must be specifically requested and performed separately from the compilation process. The names of external variables and functions from the resulting object files are matched up by a loader which has no concept of data type. A loader that could check types would be of great help, and would not be difficult to provide.

A class declaration specifies a type so it can be included in several source files without any ill effects. It must be included in every file using the class. Typically, member functions do not reside in the same file as the class declaration. The language does not have any expectations of where member functions are stored. In particular, it is not required that all member functions for a class should be in one file, or that they should be separated from other declarations.

Since the private and the public parts of a class are not physically separated, the private part is not really "hidden" from a user of a class, as it would be in the ideal data abstraction facility. Worse, any change to the class declaration may necessitate recompilation of all files using it. Obviously, if the change was to the private part, only the files containing member functions or friends have to be recompiled†. A facility that could determine the set of functions (or the set of source files) that needs to be re-compiled after a change to a class declaration would be extremely useful. It is unfortunately non-trivial to provide one that does not slow down the compiler significantly.

Efficiency

Run time efficiency of the generated code was considered of primary importance in the design of the abstraction mechanisms. The general assumption was that if a program can be made to run faster by not using classes, many programmers will prefer speed. Similarly, if a program can be made to use less store by not using classes, many programmers will prefer compact representation. It is demonstrated below that classes can be used without any loss of run time efficiency or data representation compactness compared to "old C" programs.

This insistence on efficiency led to the rejection of facilities requiring garbage collection. To compensate, the overloading facility was designed to allow complete encapsulation of storage

† The addition of a new member function will in most cases not create a need for any re-compilation. The addition may, however, hide an extern function used in some other member function, thus changing the meaning of the program. Unfortunately, this rare event is quite hard to detect.

management issues in a class. Furthermore, it has been made easy for a programmer to provide special purpose free store managers. As described above, constructors and destructors can be used to handle allocation and deallocation of class objects. In addition, the functions `operator new()` and `operator delete()` can be declared to redefine the meaning of the `new` and `delete` operators.

A class which does not use `virtual` functions uses exactly as much space as a C `struct` with the same data members. There is no hidden per object store overhead. There is no per class store overhead either. A member function does not differ from other functions in its store requirements. If a class uses `virtual` functions there is an overhead of one pointer per object plus one pointer per `virtual` function.

When a (non-virtual) member function is called, for example `ob.f(x)`, the address of the object is passed as a hidden argument: `f(&ob,x)`. Thus call of a member function is as least as efficient as a call of a non-member function. The call of a `virtual` function `p->f(x)` is roughly equivalent to an indirect call `(*(p->virtual[5]))(p,x)`. Typically this causes three memory references more than a call of an equivalent non-virtual function.

If the function call overhead is unacceptable for an operation on a class object the operation can be implemented as an inline function, thus achieving the same run-time efficiency as if the object had been directly accessed.

Implementation and Compatibility

The C++ compiler front end, `cfront`, consists of a YACC parser[11] and a C++ program. Classes are used extensively. It is about same size as the equivalent part of the PCC compiler for C (13500 lines including comments etc.). It runs a bit faster, but uses more store. The amount of store used depends on the number of external variables and the size of the largest function. It will never run on machines with a 128K byte address space (like a DEC PDP11/70); three times that amount of store appears to be more reasonable. A completely type checked internal representation is produced. This can then be transformed into suitable input for a range of new and old code generators. In particular, an "old C" version of any C++ program can be produced. This makes it trivial to transfer `cfront` to any system with a C compiler.

With few exceptions the C++ complier accepts old C. The runtime environment, the linkage conventions, and the method for specifying separate compilation remain unchanged. The major incompatibility is that a function declaration, for example

```
int f();
```

in old C declares a function with an unknown number of arguments of unknown types. In C++, that declaration specifies that `f` takes no arguments. A C++ version of the declarations for the standard libraries exists. Another difference is that in C++ a non-local name can only be used in the file in which it occurs, unless it is explicitly declared to be `extern`; in old C a non-local name is common to all files in a multi-file program, unless it is explicitly declared to be `static`. Name clashes with the new key words `class`, `const`, `delete`, `friend`, `inline`, `new`, `operator`, `overload`, `public`, `this`, and `virtual` may cause minor irritations.

It is often claimed that one of C's major virtues is that it is so small that every programmer understands every construct in the language. In contrast, languages like PL/1 and Ada are presented as if every programmer writes in his own subset of the language and can understand programs written by others only with great difficulty. It follows from this view that extension of C is bad. This argument against "big languages" ignores the simple fact that the dependencies between data structures and the functions using them exist in a program independently of whether or not they have been recorded in a class declaration. Programs using classes tend to be marginally shorter than their unstructured counterparts†. Furthermore, C is already large enough for subcultures using subsets of the language to exist, and the macro facilities are often used to create arbitrarily incomprehensible variations of the language.

† 1% to 10% shorter is typical; 50% shorter has been seen; the author has yet to see a program that grew without functionality being added.

The C++ manual is only 14% longer than the C manual so the effort of learning the new language facilities should not be prohibitively large. In particular, it should be a small effort compared with learning a new language containing data abstraction features. However, when classes are used to create new data types, a new dialect of the language is in fact created. This will lead to different incompatible "dialects". This is not that much different from the current state of affairs, and hopefully "standard" classes providing basic facilities like input/output, sets, tables, strings, graphics, etc. will win wide acceptance.

Comparison with Other Languages

To compare two languages takes a whole paper, if not a book. Consequently, this single page can provide only a few personal opinions and pointers to the main areas of difference between the languages. For completeness C++ itself is criticized in the same way as the other languages.

The C++ class facility is modeled on the original Simula67 classes[1,2]. Simula relies on garbage collection both for class objects and procedure activation records, and does not provide facilities for function name or operator overloading. It is, however, a most beautiful and expressive language, and C++ classes owe more to it than to any other language.

Smalltalk[3] is another language with the same kind of facilities for creating class hierarchies. There, however, all functions are virtual and all type checking done at run time. This means that where a C++ base class provides a fixed type-checked interface to a set of derived classes, a Smalltalk superclass provides a minimal untyped set of facilities that can be arbitrarily modified. Smalltalk relies on garbage collection and on dynamic resolution of member function names. It does not provide operator overloading in the usual sense, but an operator may be the name of a member function. Smalltalk provides an extremely nice integrated environment for program construction. The resulting programs are very demanding of resources, however.

Modula-2[12] provides a rudimentary abstraction facility called a `module`. A module is not a type but a single object containing data and access functions. It is somewhat similar to a class with all data members `static`. There is no facility equivalent to derived classes. It does not allow overloading of function names or operators. No garbage collection is provided.

Mesa's[6] modules are distinguished by a clean and flexible separation of the interface of a module from its implementation. This enables and requires sophisticated facilities for separate compilation and linking. A module can import and export both procedure and type names. The rules for instantiation of modules (object creation and initialization) are so general as to make them inelegant. Some space and time overheads are incurred by using modules. There are no facilities for constructing module hierarchies and no facilities for operator overloading. Mesa relies on garbage collection both for data objects and procedure activation records. Consequently, it will run efficiently only where hardware support for garbage collection is available.

Ada's[4] data abstraction facility, the `package`, is essentially similar to the `class/friend` facility in C++. There is no equivalent to member functions or constructors; this leads to verbosity. Nor is there an equivalent to derived classes, so the `shape` example above does not appear to have an elegant solution in Ada. Operators and function names can be overloaded; assignment can not. Packages can be generic. That is, a package can be defined with types as arguments. The standard example is a stack of elements where the type of an element is an argument. The facility is far less flexible than C++ "polymorphic classes", but more space efficient for simple abstractions. Ada does not provide garbage collection.

C++ provides no integrated environment for editing, debugging, control of separate compilation, and source code control. The Unix/C environment[11,5] provides a tool kit of such services, but it leaves much to be desired. No garbage collection is provided. C++ classes distinguish themselves by combining facilities for creating class hierarchies with efficient implementation. The facilities for object creation and initialization are notable. The facilities for overloading assignment and argument passing are unique to C++.

Conclusion

The addition of classes represents a quantum jump for the C language, the least extension that provides facilities for data abstraction for systems programming. The experience of three years with intermediate versions ("C with classes") demonstrated both the usefulness of classes and the need for the more general facilities presented here. The efficiency of both the compiled code and the compiler itself compares favorably with old C.

Acknowledgements

The concepts presented here would never have matured without the constant help and constructive criticism from my colleagues and users; notably, Tom Cargill, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro.

References

[1] Dahl, O-J. and Hoare, C.A.R.
Hierarchical Program Structures
Structured Programming pp175-220
Academic Press (1972)

[2] Dahl, O-J., Myrhaug, B., and Nygaard, K.
SIMULA Common Base Language
Norwegian Computing Center, S-22 (1970)

[3] Goldberg A. and Robson D.
Smalltalk-80 The Language and its Implementation
Addison Wesley (1983)

[4] Ichbiah J.D. et.al.
Rationale for the Design of the ADA Programming language
SIGPLAN Notices, Vol 14 no 6, June 1979

[5] Kernighan B.W. and Ritchie, D.M.
The C Programming Language
Prentice Hall (1978)

[6] Mitchell J.G. et.al.
Mesa Reference Manual
Xerox PARC CSL-79-3 (1979)

[7] Orwell, G.
1984
Harcourt Brace Jovanovich, Inc. 1949

[8] Stroustrup, B.
Classes: An Abstract Data Type Facility for the C Language
SIGPLAN Notices, Vol 17 no 1. pp42-15, January 1982

[9] Stroustrup, B.
Adding Classes to C: An Exercise in Language Evolution
Software Practice and Experience, VOL 13, pp139-161 (1983)

[10] Stroustrup, B.
C++ Reference Manual
In this volume.

[11] Unix Programmer's Manual
Bell Laboratories (1979)

[12] Wirth N.
Programming in modula-2
Springer-verlag 1982