AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 109

Data Abstraction in C

Bjarne Stroustrup

January 1, 1984

Data Abstraction in C

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ is a superset of old C; 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 old 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 old C.

Except for details like introduction of new keywords, C++ is a superset of old 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 a dozen 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 old 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.

† 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. Consider ++ a surname, to be used only on formal occasions or to avoid ambiguity. Among friends C++ is referred to as C, and the C language described in the C book [5] is "old C". 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.

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:

[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:

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

```
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 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:

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

```
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 */
```

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 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*. 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);
}
```

† 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.

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 old 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 example:

† It is, however, not that difficult to write a garbage collecting implementation of the *new* operator, as has been done for the old 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.

```
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 often 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 without the programmer knowing its particular type. For example:

```
class clock_face : public shape {
        shape* face;
        line   h_hand, m_hand, s_hand;
public:
        void   draw();
        void   rotate(int);
        void   set(int, int, int);
        void   move(int);
        ...
}
```

The time displayed by the clock can be *set()* to a particular time or one can *move()* the displayed time a number of seconds ahead. Note that *clock_face::move()* hides *shape::move(),* so to change a *clock_face's* location on the screen one must either use the qualified name *shape::move* or use a *shape* pointer.

```
clock_face cf;
shape* sh = &cf;
cl.move(1);
cl.shape::move( point(100,200) );
sh->move( point(200,100) );
```

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.

A class *object* with a "matching" set of polymorphic classes and functions is being designed. The intention is to provide it as a standard library.

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("real=%g imaginary=%g", zz.real, zz.imag);
```

Unfortunately, since the standard input/output functions know only the standard types it is

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

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 is to write a function *put()* and a function *get()* for each basic and user defined type, so that one can write code like this

```
while ( get(zz) ) put(zz);
```

Conventionally, a *get* function takes a reference to an object which is to have a value read into it and returns zero if the read succeeded and non-zero otherwise. For example, for class *complex* one could declare:

```
int        get(char*, complex&);  /* read from string */
int        get(FILE*, complex&);  /* read from file */
inline int get(complex& a) { return get(stdin,a); }
int        put(char*, complex);
int        put(FILE*, complex);
inline int put(complex a) { return put(stdout,a); }
```

Such input/output functions can be defined using previously defined *put* and *get* functions until only basic types need to be handled. Then *printf()* and *scanf()* can be used. For large objects *put()* can also take a reference. Surprisingly enough, it appears that the *get* functions are the easiest to write, since there invariably is a constructor to do the non-trivial part of the job.

There is a loss of control over the formatting of output when using *put* compared with using *printf* directly. Where such finer control is necessary, one must revert to the old style. Macros like

```
#define putlab(x) ( put("x= "), put(x), put("\n") )
```

are handy to overcome the lack of an output function taking a variable number of arguments of varying types. A "%v" option to *printf()* and *scanf()* for handling pointers to objects of class *object* is being considered, but not yet implemented.

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.

† 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.

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 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 *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 old C (12000 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 an old 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, and a program producing the "missing declarations" for a set of source files is being written. 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 sub-

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

cultures using subsets of the language to exist, and the macro facilities are often used to create arbitrarily incomprehensible variations of the language.

The *cfront* manual is only 14% longer than the "old 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 classes providing basic facilities like sets, tables, strings and graphics 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.

---

functionality being added.

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
AT&T Bell Laboratories CSTR-108. January 1, 1984.

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

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

Operator Overloading in C

Bjarne Stroustrup

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the mechanism for operator overloading provided in C++[1]. A programmer can define a meaning for the standard C operators when applied to objects of a specific class[2]. Most operators can be defined for objects of a class; in addition to arithmetic, logical, and relational operators, call "*( )*" and element selection "*[ ]*" can be defined, and both assignment and initialization can be redefined. It is not possible to change the meaning of operators applied to non-class objects, nor to change the syntax or precedence rules for operators. Constructors can be used both to provide constants for a user-defined type and to specify conversions between types.

Examples of how to define data types like *complex, matrix,* and *string* are presented. In these examples it is shown how to handle expressions involving objects of mixed type, large objects, and objects of varying size.

The overloading facilities are discussed in some detail to demonstrate the run-time efficiency that can be achieved when using them and the simplicity of their implementation in the compiler.

[1] Bjarne Stroustrup: "The C++ Reference Manual" AT&T Bell Laboratories CSTR-108. January 1, 1984.
[2] Bjarne Stroustrup: "Data Abstraction in C" AT&T Bell Laboratories CSTR-109. January 1, 1984.

Introduction

Programs often manipulate objects that are concrete representations of abstract concepts. For example, the C data type *int* together with the operators + − * / etc., provides a (restricted) implementation of the mathematical concept of integers. Such concepts typically include a set of operators representing basic operations on objects in a terse, convenient, and conventional way. Unfortunately, only very few such concepts are or can be directly supported by a programming language. For example, ideas like complex arithmetic, matrix algebra, logic signals, and strings receive no direct support in C. Classes provide a facility for specifying a representation of non-primitive objects in C together with a set of operations that can be performed on such objects. Defining operators to operate on class objects sometimes allows a programmer to provide a more conventional and convenient notation for manipulating class objects than could be achieved using only the basic functional notation. For example:

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

defines a simple implementation of the concept of complex numbers, where a number is represented by a pair of double precision floating point numbers manipulated (exclusively) by the operators + and *. The programmer provides a meaning for + and * by defining functions named *operator+* and *operator**. For example *a=b+c*a* means (by definition) *a=operator+(b,operator*(c,a))*. It is now possible to approximate the conventional interpretation of complex expressions. For example:

```
complex a = complex(1, 3.1);
complex b = complex(1.2, 2);
complex c = b;

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

The usual C precedence rules hold so the second statement really does assign *b+(c*a)* to *b*.

Overview

First the aims of the design of the overloading mechanism are briefly stated.

Then the facilities for defining class *complex* are described. Complex arithmetic was chosen as the first example of a user-defined data type because a complex number has a trivial representation, because the set of basic operations is well known, and because there exists a conventional notation for these operations. This allows the discussion of complex numbers to focus on the problem of providing that conventional notation. This will involve discussion of several language facilities that do not directly concern operator overloading. These facilities ("friend functions", "constructors", and "overloaded function names") can be used to make a user-defined data type convenient to use. It will be shown how to specify rules for initializing variables, how to provide constants for a type, how to specify type conversion rules, and how to accommodate the need to use conventional names like *abs* and + for operations on different types.

Once these techniques have been presented, it is possible to consider data types where the representation is non-trivial. Class *matrix* is an example of a data type where it is prohibitively expensive to copy an object each time an operation is performed on it. A new data type "reference" is introduced to cope with this. Class *string* shows how a data type whose implementation involves free store management and sharing of objects can be implemented.

Finally, the techniques used to implement the overloading facilities are discussed.

Aims

The main aims for the design of the overloading mechanism described here are:

[1] It must be possible to define data types like *complex*, *matrix*, and *string* with a user interface as elegant as one would expect from a built in data type.

[2] It must be possible to provide efficient implementations of the basic operations on such user defined data types.

[3] The base language must remain immutable, that is, it should not be possible to redefine the meaning of operators applied to objects of a non-class type.

[4] Programs using the overloading facilities must be relatively easy to compile.

In particular, the first aim is considered to imply

[a] that all storage management issues in the implementation of a new type can be hidden from a user.

[b] that "mixed mode" operations between user defined types and between user defined and basic types can be defined.

[c] that constants can be defined for a user defined type.

[d] that variables of a user defined type can be initialized like variables of basic data types.

It is not the intention to provide facilities for the user to change the syntax of the language in any way. Furthermore, C++ is largely compatible with "old C"; see reference 1 for details†.

It is not the intention to provide any "default semantics" for operators, nor is it the intention to restrict definitions of operators to some preconceived idea of what is reasonable. For example, it is quite possible to define = to mean plus and + to mean assignment. The only protection provided against idiotic use is the guarantee that the base language is immutable.

Friends and Members

Two kinds of functions can be defined to manipulate the representation of a user-defined data type: member functions and friend functions. They differ in both scope and calling syntax. Consider a simple class *complex* with a conjugation function of each type:

```
class complex {
        float re, im;
public:
        complex member_conj();
        friend complex friend_conj(complex arg);
};
```

The member function has the more elegant definition since it can refer to the representation directly where the friend function must explicitly refer to it through its argument:

† The major incompatibility is that *int f()* declares a function taking no arguments and returning an *int*.

```
complex complex.member_conj()
{
        complex con;
        con.re = re;
        con.im = -im;
        return con;
}

complex friend_conj(complex arg)
{
        arg.im = -arg.im;
        return arg;
}
```

However, only the friend can be called using conventional mathematical notation; the member must use *object.member* notation:

```
complex a;
a.member_conj();
friend_conj(a);
```

For a conjugation operation the calling syntax for the friend function is obviously preferable. The calls of a member function *conj()* are simply too ugly to live with. Consider, for example, *a+b.conj()* and *(a+b).conj()*.

Furthermore, consider a binary operator:

```
class complex {
        ...
        complex operator+(complex);
        complex operator+(double);
};
```

This will allow one to write *aa+aa* and *aa+2* but not *2+aa* for a complex variable *aa!* Worse, there is no way of adding a function to that class *complex* that would cope with the example *2+aa*. Using friends a better declaration of *complex* can be written:

```
class complex {
        ...
        friend complex operator+(complex, complex);
        friend complex operator+(complex, double);
        friend complex operator+(double, complex);
};
```

Friends are ideal for implementing traditional arithmetic operations. However, if it is necessary to modify the contents of one operand, as in a function defining an assignment operator, a member function must be used. Furthermore, where there is no other reason to prefer the one implementation over the other an operation should be a member. A definition of a member is typically shorter than the definition of the equivalent friend, the implicit passing of the pointer identifying the object for which a member is invoked is potentially more efficient than argument passing, and the name of a member is restricted to its proper scope rather than global. Finally, constructors and destructors (see below) must be members.

Constructors

A class may have a function member with the name of the class itself, like *complex()* in class *complex* above. Such a function is called a constructor†. A constructor is a prescription for initializing a class object (construct a value of the class type). If a class has a constructor then the

† In earlier implementations of classes value constructors were called *new()* functions, and their use was limited to class object initialization.

constructor will be called to initialize objects of the class before their use. If the constructor requires arguments they must be provided. The following examples are equivalent, that is, the three variables will be initialized to the same value:

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

Initialization with an object of the same type is a legal alternative to calling the constructor.

A constructor can also be used in expressions, often replacing the use of a temporary variable. For example, *operator+()* can be defined without use of an explicit temporary variable:

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

Operator Functions

Functions defining meanings for the following operators can be declared for a class:

```
+    -    *    /    %    ^    &    |    ~    !    =    <    >
+=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --   []   ()
```

The last two are subscript and function call (see below).

The name of an operator function is the keyword *operator* followed by the operator itself. Such names are syntactically legal only in declarations, so an operator function cannot be called directly, only through the use of its associated operator.

A binary operator can be defined either by a member function taking one argument or by a friend function taking two arguments. Thus, for any binary operator @, *aa@bb* can be interpreted as either *aa.operator@(bb)* or *operator@(aa,bb)*. If both *operator@* functions are defined the former interpretation is used. A unary operator, whether prefix or postfix, can be defined by either by a member function taking no arguments or a friend function taking one argument. Thus, for any unary operator @, *both aa@* and *@aa* can be interpreted as either *aa.operator@()* or *operator@(aa)*. If both *operator@* functions are defined the former interpretation is used. For example, *operator&()* defines the operator usually called "address of", and not the binary operator "and". An operator which can be used both as a unary and as a binary *(*−, *, and *&)* can be used only as declared. When the operators ++ and −− are overloaded, it is not possible to distinguish prefix application from postfix application.

The meanings of some C operators are defined to be equivalent to some combination of other operators on the same arguments. For example: *++a* means *a+=1* which in turn means *a=a+1*. Such relations do not hold for overloaded operators unless the user defines them that way. For example, the definition of *operator+=* cannot be deduced from the definitions of *operator+* and *operator=*. No assumptions are made about the meaning of overloaded operators. In particular, since an overloaded "assignment operator" is not assumed to implement assignment to its first argument no test is made to ensure that that argument is an lvalue.

Because of historical accident the operators = and *&* have pre-defined meanings when applied to class objects. There is no elegant way of "undefining" these two operators. They can, however, be disabled for a class *X*. For example, by declaring *X.operator&()* but not providing a definition for that function one can ensure that no program taking the address of an *X* will run. It is not possible to define a meaning for the binary *&* ("and") and retain the pre-defined meaning of the unary *&* ("address of").

An operator function must either be a member or take at least one class object argument. This rule ensures that a user cannot change the meaning of any expression not involving a user-defined data type. In particular, it is not possible to define an operator function which operates exclusively on pointers. It is not possible to define a new operator or to change the precedence of an existing operator. The following operators can not be defined or re-defined:

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

Overloaded Function Names

The implementation of complex numbers presented in the introduction is too restrictive to please anyone, so it must be extended. This is mostly a trivial repetition of the techniques presented above. However, two desirable features cannot be handled that easily: unary minus and mixed operations on complex and real numbers. Both require two different interpretations for a single operator symbol. For example:

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

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

When several (different) function declarations are specified for a single name, that name is said to be overloaded. When that name is called, the correct function to execute is selected by comparing the types of the actual arguments with the argument types in the function declarations. Of the "usual arithmetic conversions" defined in §6.6 of the C++ reference manual[1] only the conversions *char−>short−>int, int−>double, int−>long,* and *float−>double* are performed.

With this declaration of *complex* we can now write:

```
complex a(1,1), b(2,2), c(3,3), d(4,4), e(5,5);
a = -b-c;
b = c*2.0*c;
c = (d+e)*a;
```

Any function name can be overloaded, not just *operators*. The names of non-class-member functions must explicitly be declared overloaded. For example:

```
overload abs;
int      abs(int);
double   abs(double);
complex  abs(complex);
```

When an overloaded name is called, the list of functions is scanned in order to find one which can be invoked. For example *abs(12)* will invoke *abs(int)* and *abs(12.0)* will invoke *abs(double)*. Had the order of declarations been reversed, both calls would have invoked *abs(double)*.

All operators are by definition overloaded.

Conversion

Writing a function for each combination of *complex* and *double*, as for *operator*()* above, is unbearably tedious. For example, a realistic facility for complex arithmetic provides more than 10 functions taking two complex or real arguments [3]. An alternative is to declare a constructor that

[3] Leonie V. Rose and Bjarne Stroustrup: "Complex Arithmetic in C" AT&T Bell Laboratories CSTR-109. January 1, 1984.

creates a *complex* given a *double*.For

```
                class complex {
                        ...
                        complex(double r) { re=r; im=0; }
                };
```

A constructor requiring a single argument need not be called explicitly. For example

```
                        complex z1 = complex(23);
                        complex z2 = 23;
```

are both legal, and *z1* and *z2* will both be initialized calling by *complex(23)*.

A constructor is a prescription for making a value of a given type. Where a value of a type is expected, and where such a value can be created by a constructor, given the value to be assigned, the constructor will be used. For example, class *complex* could be declared like this:

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

                friend complex operator+(complex, complex);
                friend complex operator*(complex, complex);
        };
```

and operations involving *complex* variables and integer constants would be legal. An integer constant will be interpreted as *complex* with the imaginary part zero. For example: *a=b*2* generates code like *a=operator*(b,complex(double(2)))*.

An assignment to an object of class *X* is therefore legal if either the assigned value is an *X*, or if *X* has a constructor accepting a single argument of the type of the assigned value.

In some cases a value of the desired type can be constructed by repeated use of constructors. This must be handled by explicit use of constructors; only one level of implicit construction is legal. Note that the standard conversions described above are always performed and do not count as "implicit construction". In some cases a value of the desired type can be constructed in more than one way. Such cases are illegal. For example:

```
    class x { ... x(int); x(char*); };
    class y { ... y(int); };
    class z { ... z(x); };

    overload f;
    x f(x);
    y f(y);

    z g(z);

    f(1);          /* illegal: ambiguous f(x(1)) or f(y(1))  */
    f(x(1));
    f(y(1));
    g("asdf");     /* illegal: g(z(x("asdf"))) not tried */
    g(z("asdf"));
```

Where an overloaded function takes an argument of a type for which a constructor exists more than one interpretation may appear legal. For example:

```
class x { ... x(int); };
overload h(double), h(x);
h(1);
```

The call could be interpreted either as *h(double(1))* or as *h(x(1))* and would appear to be ambiguous and therefore illegal according to the rule above. However, the first interpretation is an "exact match" and will be chosen; the rule against ambiguities applies only to user-defined conversions.

These rules for conversion are neither the simplest to implement, the simplest to document, nor the most general which could be devised. Consider the requirement that a conversion must be unique to be legal. A simpler approach would allow the compiler to use any conversion it could find; thus it would not be necessary to consider all possible conversions before declaring an expression legal. Unfortunately, this would mean that the meaning of a program depended on which conversion was found. In effect, the meaning of a program would in some way depend on the order of the declaration of the conversions. Since these will often reside in different source files (written by different programmers), the meaning of a program would depend on the order in which its parts were merged together. Alternatively, implicit conversions could be disallowed. Nothing could be simpler, but this rule leads to either inelegant user interfaces or an explosion of overloaded functions as seen in the class *complex* in the previous section.

The most general approach would take all available information into account. For example, using the declarations above, it would handle *aa=f(1)* because the type of *aa* will determine a unique interpretation. If *aa* is an *x*, *f(x(1))* is the only one yielding the *x* needed in the assignment. The most general approach would also cope with *g("asdf")* because *g(z(x("asdf")))* is a unique interpretation. The problem with this approach is that it requires extensive analysis of a complete expression to determine the interpretation of each operator and function call. This leads to slow compilation and also to surprising interpretations and error messages, as the compiler considers conversions defined in libraries etc. It simply takes more information into account than the programmer writing the code can be expected to know!

Constants

It is not possible to define constants of a class type in the sense that *1.2* and *12e3* are constants of type *double*. However, constants of the basic types can often be used instead if class member functions are used to provide an interpretation for them. Constructors taking a single argument provides a general mechanism for this. Where constructors are "simple" and inline substituted, as in class *complex* above, it is quite reasonable to think of constructor invocations as constants. For example *zz1*3+zz2*complex(1,2)* will cause three function calls and not five.

References

For each use of a *complex* binary operator a copy of the second operand is passed as an argument to the function implementing the operator. The overhead of copying two *doubles* is noticeable but probably quite acceptable. Unfortunately, not all classes have a conveniently small representation. To cope with this, one could try to declare operator functions to take pointer arguments. For example:

```
class matrix {
        double m[4][4];      /* 128 bytes on a VAX */
public:
        matrix();
        friend matrix operator+(matrix*, matrix*);
        friend matrix operator*(matrix*, matrix*);
};
```

Unfortunately, this leads to rather strange looking expressions.

```
matrix a, b, c;
a = &b + &c;
```

This is clearly not acceptable. Furthermore, the declarations of the *matrix* operator functions above are illegal, since they do not take any class object arguments (pointers to class objects do not count). The alternative is to define matrix operations to take "references" as arguments, rather than pointers.

A reference, like a pointer, embodies the idea of an address of an object. The notation *X&* means reference to *X*. A reference differs from a pointer in that

[1] Use: when a reference is used the dereference operator * will be implicitly applied.

[2] Initialization: when a variable is declared to be a *X&*, it can be initialized by an *X*. The address operator *&* will be implicitly applied. It is also legal to initialize a *X&* by a *X**.

For example:

```
matrix m;
matrix& r = m;    /* means  r = &m                                    */
matrix m2 = r;    /* means m2 = *r                                    */
r = m;            /* means *r =  m                                    */
m = r;            /* means  m = *r                                    */
```

Note that initialization of a reference is treated very differently from assignment to it. This is reasonable, since dereferencing of an uninitialized variable is known to be meaningless. For example:

```
int a;
int& r1 = a;       /* means  r1 = &a                                  */
int& r2; r2 = a;   /* error: uninitialized reference                  */
```

Argument passing and function value return are considered to be initializations, so:

```
matrix& f(int);
matrix operator+(matrix&, matrix&);
m = f(1);          /* means m = *f(1);                                  */
f(2) = m2;         /* means *f(2) = m2;                                 */
m+m2;              /* means +(&m,&m2)                                   */
r+m;               /* means *r+m that is +(&*r,&m) that is +(r,&m)      */
```

Despite appearances, no operator operates on a reference. For example,

```
int ii = 0;
int& rr = ii;
rr++;
```

is legal, but *rr++* does not increment the reference *rr;* rather, the interpretation is *(*rr)++*. That is, ++ is applied to an *int* which happens to be *ii*.

The value of a reference cannot be changed after initialization. To get a pointer *pp* to denote the same object as a reference *rr* one can write *pp=&rr*. This will be interpreted as *pp=&*rr*.

Consider the declaration

```
double& dr = 1;
```

Here, the initializer is not an lvalue. In fact, it is not even of the right type. In such cases

[1] first the conversion rules are applied,

[2] then the resulting value is placed in a temporary variable,

[3] finally the address of this is used as the value of the initializer.

Thus, the interpretation of the example above is:

```
                                        double* dr;
                                        double temp;
                                        temp = double(1);
                                        dr = &temp;
```

References allow the use of conventional expressions involving the usual arithmetic operators for "large objects" without requiring an implementation that causes excessive copying. The plus operator could be defined like this:

```
    matrix operator+(matrix& arg1, matrix& arg2)
    {
            matrix sum;
            int i, j;

            for (i=0; i<4; i++)
                    for (j=0; j<4; j++)
                            sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
            return sum;
    }
```

The dot is used for member selection, rather than −>, because *arg1* and *arg2* are implicitly dereferenced; *arg1.mem[i][j]* means *(*arg1).mem[i][j]*.

Returning a Class Object

In the example above *operator+(matrix&,matrix&)* operates on the operands to + through references, but returns an object value rather than a reference. Returning a reference would appear to be more efficient. For example:

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

This would be legal, and the matrix expression *a*b+c* would be interpreted as *operator+(operator*(&a,&b),&c)*. However, the user would have to implement some kind of temporary variable to hold the result of *operator()*. Since a reference to it will be passed out of the function as the return value, it cannot be a local variable. It would typically be allocated on the free store, and later freed for re-use. Copying the return value will often be cheaper (in execution time, code space, and data space) and simpler to program.

For medium sized objects the style of declaring an *operator* function to take reference arguments and return an object value is probably the best approach. The implementation of functions returning medium and large objects relies on passing a pointer to an object where the result is deposited. If a function, like the original *operator+(matrix&,matrix&)* above, composes its return value in a local variable and then returns it, the compiler will ensure that the result value is composed right in the object where it would eventually have been copied to (this will be explained in greater detail below). This means that the code generated for an operator taking reference arguments and returning an object value operates on three objects through pointers: the two reference arguments and the result through a compiler provided target pointer.

Overloading Function Call: *operator()*

Function call, that is, the notation *expression(expression-list)*, can be interpreted as a binary operation and the call operator *()* can be overloaded in the same way as other operators. For example:

```
class string {
        char*   s;              /* representation */
        int     length;
        string* sub_of;         /* substring of   */
public:
        string(char*);
        string();
        string  operator=(string&);             /* assignment         */
        string  operator()(int pos, int len);   /* substring selector */
};
```

With suitable definitions of *operator=()* and *operator()()* one can define assignment and subscripting to be used like this:

```
                    string a("foo"), b("bar"), c;
                    c = b;
                    c = b(1,2);
                    a(0,1) = b;
```

The basic idea is to let the substring function return an object containing a pointer to the original string which can be used to manipulate that string. For example:

```
          string string.operator()(int pos, int lgt) {
          {
                  string sub;
                  sub.sub_of = this;
                  sub.length = lgt;
                  sub.s = s+pos;
                  return sub;
          }

          void string.operator=(string& from)
          {
                  if (sub_of) {   /* assign to sub_of->s, etc. */
                  }
                  else {          /* assign to s, etc. */
                  }
          }
```

This technique for assigning to a part of an object through an object returned by a selector function is useful in many contexts.

An argument list for an *operator()* function is evaluated and checked according to the usual argument passing rules.

Overloading Subscripts: *operator[]*

An *operator[]* function can be used to give subscripts a meaning for class objects. For example:

```
class string {
        char* s;
        ...
        char& operator[](int i) { return (0<=i && i<length) ? s[i] : 0; }
};
```

Note that because the value returned by *string.operator[]()* is a *char&* that function can be used on either side of an assignment. For example:

```
string ss = "asdf";
ss[1] = ss[3];
```

The assignment is interpreted as

```
*(ss.operator[](1)) = *(ss.operator[](3))
```

that is, the new value of *ss* is *"afdf"*

The second argument (the subscript) of an *operator[]* function may be of any type. This makes it possible to define associative arrays, etc.

Destructors

A destructor† is a function that will be called (implicitly) when a class object is destroyed. A class object is destroyed if it goes out of scope or if it is explicitly deleted using the *delete* operator. The name of the destructor for class *X* is ~*X*, and it takes no argument. Destructors are often useful to clean up secondary data structures. For example:

```
class vector {
        int* v;
        int size;
        ...
        vector(int sz) { v = new int[size=sz]; }
        ~vector()      { delete v; }
};
```

Note that a destructor is not invoked when a value is destroyed by assignment. For example,

```
vector v1(10), v2(20);
v1 = v2;
```

will not cause the destructor *~vector()* to be invoked for *v1*. Otherwise, assignment of an object to itself, for example *v1=v1*, would lead to chaos. To ensure that *v1's* vector of integers is not lost *vector.operator=()* can be defined.

Constructors Revisited

When *operator=()* is declared it will be called for all uses of the assignment operator "=". However, an object can also be copied as an argument to a function, as a return value, or as an initializer for a new class object. Each of these three cases involves the construction of a value in an uninitialized object. They are treated identically and are collectively referred to as initialization.

Consider the problem of implementing "call by value" for objects of class *string*. A user sees a *string* as a single object, but its implementation consists of two parts: the class object itself and the string representation pointed to by it. When a copy is made of a class *string* object both copies will denote the same representation and further operations on them may corrupt it. To handle explicit assignments *operator=()* can be declared (for simplicity substrings and assignments like *s=s* will be ignored here):

```
void string.operator=(string& from) {
        delete s;
        length = from->length;
        s = new char[length+1];
        strcpy(s,from->s);
}
```

To cope with initialization for a class *X*, a constructor is needed since the default initialization method, bitwise copy, may cause chaos. Intuitively, a constructor *X(X)* would implement

† In earlier implementations destructors were called *delete()* functions.

construction of one *X* from another. However, to avoid infinite regression the constructor invoked to implement passing of arguments of type *X* cannot itself take an *X* as an argument. Consequently, it is illegal to declare a constructor *X(X),* and a constructor *X(X&)* is used instead.

The constructor *X(X&)* will typically be simpler than the corresponding *X.operator=()* for the same class. For example:

```
string.string(string& from)
{
        length = from->length;
        s = new char[length+1];
        strcpy(s,from->s);
}
```

One can now use strings without ending up with shared representations.

```
string a = "asdf";
string b = a;      /* a's representation will be copied ( b.string(&a) ) */
f(a);              /* a's representation will be copied (see below)      */
```

Function Return Revisited

In the following pseudo-C will be used to describe generated code. Syntactically illegal names will be used freely. Consider the function

```
X f()
{
        X a(1);
        X b(2);
        b->compute();
        ...
        return b;
}
```

where the constructor *X(X&)* has been declared. The (unoptimized) code generated will be something like this:

```
void f(X* tp) {         /* pass result location */
        X a, b;
        a.X(1);         /* construct value */
        b.X(2);
        b.compute();
        ...
        tp->X(&b);      /* return result */
        a.~X();         /* destroy value */
        b.~X();
}
```

Assuming that the constructor *X()* and the destructor *~X()* have also been declared, they must be called as the scope of the function is entered and left, respectively.

By using the result variable, **tp,* as the local variable *b* the code can be optimized to something like this:

```
void f(X* tp) {
        X a;
        a.X(1);
        tp->X(2);           /* b.X(2);     */
        tp->compute();      /* b.compute(); */
        ...
        a.~X();
}
```

The object denoted by *tp* now needs to be initialized because it is manipulated under the name *b*. However, *b* is not deleted because it does not belong to the scope of this function.

Now consider a call of *f()*:

```
X x(0);
x = f();
```

Assuming *X.operator=()* is declared the code generated is

```
X x;
X temp;
x.X(0);
f(&temp);
x.operator=(&temp);
temp.~X()
```

where *temp* is used nowhere else. Note that this code is independent of whether the return operation is optimized.

An obvious optimization appears to have been missed here. Why wasn't *f(&x)* generated so that the temporary could be omitted and no copying at all would occur? This further optimization can only be done where it is known that the value of *x* is not used during the execution of *f()*. Unfortunately, for most functions this cannot be deduced at compile time. However, where this is known, it is considered legal to optimize away temporary variables of type *X* even when *X(X&)*, *X.operator=()*, or *~X()* is defined by the user.

Argument passing Revisited

Consider a function *f()* that takes an argument of class *X* for which the constructor *X(X&)* has been declared:

```
void f(X a) { ... }
```

The code generated will be something like this:

```
void f(X a) {
        ...
        a.~X();
}
```

The code generated for calls

```
f(x);
f(g());         /* g() returns a class X object */
```

is something like this

```
X temp;         /* uninitialized */
temp.X(&x)
f(temp);
g(&temp);       /* g() returns its value in temp */
f(temp);
```

Caveat

Like most programming language features, operator overloading can be both used and misused. In particular, the ability to define new meanings for old operators can be used to write programs that are well nigh incomprehensible to anyone. Imagine, for example, the problems facing a reader of a program where the operator + has been made to denote subtraction, or a program where all common operations are invoked using the arithmetic operators even though the data types used have no conventional association with those operators.

The mechanism presented here should protect the programmer/reader from the worst excesses of overloading by not enabling a programmer to change the meaning of operators for basic data types like *int*, and by preserving the syntax of expressions and the precedence of operators.

Acknowledgements

Stu Feldman, Doug McIlroy, and Jonathan Shopiro contributed greatly to the design of these mechanisms.

Appendix A: class string

To demonstrate the overloading techniques in a larger example class *string* is outlined. It implements a version of character strings with associated operators where the user need not worry about the store management necessary for handling variable length strings. Many implementation details are left as comments. Error handling is ignored. The standard C library string functions are used for the low level character handling. It differs from the simpler string classes used above.

The major ideas behind class *string* are

[1] to provide all operations on strings through a class *string* which in effect is a pointer to the string representation.

[2] to let class *string* take care of all storage management for string representations.

[3] to present operations on strings as operators, and let the compiler take care of the resulting temporaries.

[4] to provide value, rather than pointer semantics, for strings. That is, if a string is assigned or passed to a function it will be copied.

[5] to delay such copying until it is neccesary by maintaining a shared representation for several strings until they really differ.

[6] to use *operator()* to denote both positional and contextually defined substrings. A substring can be used whereever a string can.

[7] to use C character string constants as constants for the class *string* operators.

A *string* simply contains a pointer to a string representation, *srep*. Only *string* functions can access *srep* members. An *srep* contains a *use_count* which is increased when "its" string is copied and decreased when one of those copies goes out of scope. A function wanting to change the representation of a string must examine the *use_count* and make its own copy of the *srep* if it is larger than 1.

```
class srep {
friend string;
        srep* sub_of;     /* this is a substring of "sub_of" */
        int   use_count;
        char* s;          /* points to string representation */
        int   length;     /* length of string */
        char* a_s;        /* points to allocated space */
        int   a_length;   /* length of allocated space */

        srep(char*);
        ~srep();
};
```

Since the size of every string representation is known it is easy and probably worth while to manage the free store for objects of class *srep* separately.

```
srep.srep(char* ss) {
        if (ss) {
                a_length = length = strlen(ss);
                a_s = s = myalloc(a_length+1);
                strcpy(s,ss);
        }
        use_count = 1;
}

srep.~srep() {
        if (a_length) myfree(a_s,a_length+1);
}
```

A substring is a *string* with an *srep* where *sub_of!=0.*

The string operators take reference arguments for efficiency and simple implementation of the shared representation idea. A user is expected to pass strings as values, not pointers, so that the constructor *string()*, the destructor *~string()*, and *operator=()* can ensure proper free store management of the strings and consistent use of substrings.

```
class string {
        srep*   rep;

        string(char*);
        string();
        string(string&);                           /* initialization        */
        string(srep*);
        ~string();

        char*   join(char*);                       /* char* concatenation   */
        char*   join(string&);

public:

        string operator()(int pos, int lgt);       /* substring extraction */
        string operator()(string&);
        char&   operator[](int pos);               /* character extraction */
        friend string operator+(string, string);  /* concatenation         */
        void    operator= (string);
        void    operator+=(string);                /* add to end of string */
                                                                ...
};
```

The functions *string()*, *~string()*, and *operator=()* cooperate to implement value semantics for strings. Together they ensure that no free store is ever "lost", that is becomes unreferenced without being deleted, even though each user function need only consider the use of its own immediate arguments.

```
string.string()
{
        rep = new srep("");
}

string.string(char* ss)
{
        rep = new srep(ss);
}

string.string(srep* p)
{
        rep = p;
}

string.~string()
{
        if (rep->use_count-- == 1) delete rep;
}

string.string(string& from) {                /* assign to uninitialized */
        srep* fr = from.rep;

        if (fr->sub_of) {                    /* substring, make new representation */
                int lgth = fr->length;
                char* ss = new char[lgth+1];
                strncpy(ss,fr->s,lgth);
                ss[lgth] = 0;
                rep = new srep(ss);
        }
        else {                               /* string, share representation */
                rep = fr;
                rep->use_count++;
        }
}

void string.operator=(string& from) {
        srep* fr = from.rep;

        if (sub_of) {
                if (fr->sub_of) {    /* substring = substring */
                }
                else {               /* substring = string */
                }
        }
        else {
                if (fr->sub_of) {    /* string = substring */
                }
                else {              /* string = string */
                        fr->use_count++;
                        if (rep->use_count-- == 1) delete rep;
                        rep = fr;
                }
        }
}
```

The result of applying *operator()()* to a *string* is a *string* that denotes a substring of that string. This substring can then be used either to read from the original string or to write into the original string. A substring is used at most once, since it can only be assigned or passed to a function. It will therefore be interpreted and deleted by either *string.operator=()* or *string(string&)*.

```
string string.operator()(int pos, int lgt)
{
        srep* p = new srep(0);
        p->a_s = (char*)this;
        p->a_length = 0;
        p->s = &s[pos];
        p->length = lgt;
        return string(p);
}
```

The rest of the class *string* functions are comparatively simple since they need not be concerned with substrings or character strings. The "basic operations" on a string could have been implemented using reference arguments so that they did not affect the use counts and did not incur the overhead involved in updating them. Unfortunately, to do this implies that they would have to know about substrings.

```
string operator+(string s1, string s2)
{
        char* j = s1.join(s2);
        srep* p = new srep(j);
        return string(p);
}

void string.operator+=(string arg)
{
        char* j = join(arg);
        delete rep;
        rep = new srep(j);
}
```

Class *string* can be used like this:

```
void print(string a) {
        char ch;
        int i;
        while (ch = a[i++]) putc(ch);
        putc("\n");
}

main() {
        string a,
               b = "foo",
               c = b+" bar";

        a = b+c;
        print(a);
        a(1,2) = b(0,4);
        print("a="+a+", b="+b);
}
```

Complex Arithmetic in C

Leonie V. Rose†

Bjarne Stroustrup

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This memo describes a data type *complex* providing the basic facilities for using complex arithmetic in C. The usual arithmetic operators can be used on complex numbers and a library of standard complex mathematical functions is provided. For example:

```
#include <complex.h>

main(){
        complex xx;
        complex yy = complex(1,2.718);
        xx = log(yy/3);
        put(1+xx);
}
```

initializes *yy* as a complex number of the form *(real+imag*i)*, evaluates the expressions and prints the result: *(0.706107,1.10715)*.

The data type *complex* is implemented as a class using the data abstraction facilities in C (that is C++ [1], not old C). The arithmetic operators + − * /, the assignment operators = += −= *= /=, and the comparison operators == != are provided for complex numbers. So are the trigonometric and mathematical functions: *sin(), cos(), cosh(), sinh(), sqrt(), log(), exp(), conj(), arg(), abs(), norm(), pow()*. Expressions such as *(xx+1)*log(yy*log(3.2))* that involves a mixture of real and complex numbers are handled correctly. The simplest complex operations, for example + and +=, are implemented without function call overhead.

† SRP student from Lehman College, City University of New York

[1] Bjarne Stroustrup: "The C++ Reference Manual" AT&T Bell Laboratories CSTR-108. January 1, 1984.

Introduction

The C language does not have a built-in data type for complex numbers, but it (that is C++[1], not old C) does provide language facilities for defining new data types (see also references 2 and 3). The type *complex* was designed as a useful demonstration of the power of these facilities.

There are three plausible ways to support complex numbers in C. First, the type *complex* could be directly supported by the C compiler in the same way as the types *int* and *float* are. Alternatively, a preprocessor could be written to translate all use of complex numbers into standard C (for example, see reference 4 for a description of the *cxp* pre-processor). A third approach was used to implement type *complex;* it was specified as a user-defined type using the facilities of C. This demonstrates that one can achieve the elegance and most of the efficiency of a built in data type without modifying the compiler. It is even much easier to implement than the preprocessor approach, which is likely to provide an inferior user interface.

This facility for complex arithmetic provides the arithmetic operators + / * −, the assignment operators = += −= *= /=, and the comparison operators == != for complex numbers. Input and output can be done using the functions *get()* and *put()*. The initialization functions and *get()* accept a Cartesian representation of a *complex*. The functions *real()* and *imag()* return the real and imaginary part of a *complex*, respectively, and *put()* prints a *complex* as *(real,imaginary)*. The internal representation of a *complex*, is, however, inaccessible and in principle unknown to a user. Polar coordinates can also be used. The function *polar()* creates a *complex* given its polar representation, and *abs()* and *arg()* return the polar magnitude and angle, respectively, of a *complex*. The function *norm()* returns the square of the magnitude of a *complex*. The following complex functions are also provided: *sqrt(), exp(), log(), sin(), cos(), sinh(), cosh(), pow(), conj()*. The declaration of *complex* and the declarations of the complex functions can be found in Appendix A. A complete program using complex numbers can be found in Appendix B.

Complex Variables and Data Initialization

A program using complex arithmetic will contain declarations of *complex* variables. For example:

```
complex zz = complex(3,-5);
```

will declare *zz* to be complex and initialize it with a pair of values. The first value of the pair is taken as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The function *complex()* constructs a complex value given suitable arguments†. It is responsible for initializing *complex* variables, and will convert the arguments to the proper type *(double)*. Such initializations may be written more compactly. For example:

```
complex zz(3,-5);
complex c_name(-3.9,7);
complex rpr(SQRT_2,root3);
```

A complex variable can be initialized to a real value by using the constructor with only one argument. For example:

```
complex ra = complex(1);
```

will set up *ra* as a complex variable initialized to *(1,0)*. Alternatively the initialization to a real value can also be written without explicit use of the constructor:

[2] Bjarne Stroustrup: "Data Abstraction in C" AT&T Bell Laboratories CSTR-109. January 1, 1984.
[3] Bjarne Stroustrup: "Operator Overloading in C" AT&T Bell Laboratories CSTR-109. January 1, 1984.
[4] J. P. Brezin: "cxp: A Preprocessor for Complex Arithmetic Expressions in C Language Programs, Version 2". Bell Laboratories Internal Memorandum. July 1982.
† Such a function is called a constructor. A constructor for a type always has the same name as the type itself.

```
complex rb = 123;
```

The integer value will be converted to the equivalent complex value exactly as if the constructor *complex(123)* had been used explicitly. However, no conversion of a *complex* into a *double* is defined, so

```
double dd = complex(1,0);
```

is illegal and will cause a compile time error.

If there is no initialization in the declaration of a complex variable, then the variable is initialized to *(0,0)*. For example:

```
complex orig;
```

is equivalent to the declaration:

```
complex orig = complex(0,0);
```

Naturally a complex variable can also be initialized by a complex expression. For example:

```
complex cx(-0.5000000e+02,0.8660254e+02);
complex cy = cx+log(cx);
```

It is also possible to declare arrays of complex numbers. For example:

```
complex carray[30];
```

sets up an array of 30 complex numbers, all initialized to *(0,0)*. Using the above declarations:

```
complex carr[] = { cx, cy, carray[2], complex(1.1,2.2) };
```

sets up a complex array *carr[]* of four *complex* elements and initializes it with the members of the list. However, a struct style initialization cannot be used. For example:

```
complex cwrong[] = {1.5, 3.3, 4.2, 4};
```

is illegal, because it makes unwarranted assumptions about the representation of complex numbers.

Input and Output

Simple input and output can be done using the functions *get* and *put*. They are declared like this using the facility for overloading function names†:

```
int put(FILE*, complex);
int put(complex z) { return put(stdout,z); }

int get(FILE*, complex&);
int get(complex& z) { return get(stdin,z); }
```

The integer returned is in all cases zero if the operation was performed, non-zero otherwise. When *zz* is a complex variable the call *get(zz)* reads a pair of numbers from *stdin* into *zz*. The first number of the pair is interpreted as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The function *put()* writes a complex argument to *stdout*. For example:

† In C, that is in C++, a name can be used to denote several functions. For each call the proper function to execute will be chosen based on the argument type. See the section on "efficiency" below for more detail.

```
void copy()
{
        complex zz;
        while ( get(zz)==0 ) put(zz);
}
```

reads a stream of complex numbers like *(3.400000,5.000000)* and writes them like *(3.4,5)*. The parentheses and comma are mandatory delimiters for input, while white space is optional. A single real number, for example *10e-7* or *(123)*, will be interpreted as a *complex* with *0* as the imaginary part by *get()*.

A user who does not like the standard *put()* and *get()* functions can provide alternate versions.

Cartesian and Polar Coordinates

The functions *real()* and *imag()* return the real and imaginary parts of a complex number, respectively. This can, for example, be used to create differently formatted output of a *complex:*

```
complex cc = complex(3.4,5);
printf(" %g+%g*i", real(cc), imag(cc));
```

will print *3.4+5*i.*

The function *polar()* creates a *complex* given a pair of polar coordinates (magnitude, angle). The functions *arg()* and *abs()* both take a *complex* argument and return the angle and magnitude (modulus), respectively. For example:

```
complex cc = polar(SQRT_2,PI/4);    /* also known as complex(1,1) */
double magn = abs(cc);              /* magn = sqrt(2) */
double angl = arg(cc);              /* angl = PI/4 */
printf("(m=%g, a=%g)",magn,angl);
```

If input and output functions for the polar representation of complex numbers are needed they can easily be written by the user.

Arithmetic operators

The basic arithmetic operators + − (unary and binary) / *, the assignment operators = += −= *= /=, as well as the equality operators == != can be used for complex numbers. The operators have their conventional precedences. For example: *a=b*c+d* for complex variables *a, b, c,* and *d* is equivalent to *a=(b*c)+d*. There are no operators for exponentiation and conjugation; instead the functions *pow()* and *conj()* are provided. The operators += −= *= /= do not produce a value that can be used in an expression; thus the following examples will cause compile time errors:

```
complex a, b;
...
if ( (a+=2)==0 ) { ... }
b = a *= b;
```

Mixed Mode Arithmetic

Mixed mode expressions are handled correctly. Real values will be converted to complex where necessary. For example:

```
complex xx(3.5,4.0);
complex yy = log(yy) + log(3.2);
```

This expression involves a mixture of real values: *log(3.2)*, and complex values: *log(yy)* and the sum. Another example of mixing real and complex, *xx=1* is equivalent to *xx=complex(1)* which in turn is equivalent to *xx=complex(1,0)*

The interpretation of the expression *(xx+1)*yy*3.2* is *(((xx+complex(1))*yy)*complex(3.2))*

Mathematical Functions

A library of complex mathematical functions is provided. A complex function typically has a counterpart of the same name in the standard mathematical library. In this case the function name will be overloaded. That is, when called, the function to be invoked will be chosen based on the argument type. For example, *log(1)* will invoke the real *log()*, and *log(complex(1))* will invoke the complex *log()*. In each case the integer *1* is converted to the real value *1.0*.

These functions will produce a result for every possible argument. If it is not possible to produce a mathematically acceptable result, the function *complex_error()* will be called and some suitable value returned. In particular, the functions try to avoid actual overflow, calling *complex_error()* with an overflow message instead. The user can supply *complex_error()*. Otherwise a function that simply sets the integer *errno* is used. See appendix C for details.

```
complex conj(complex);
```

Conj(zz) returns the complex conjugate of *zz*.

```
double norm(complex);
```

Norm(zz) returns the square of the magnitude of *zz*. It is faster than *abs(zz)*, but more likely to cause an overflow error. It is intented for comparisons of magnitudes.

```
overload pow;
double   pow(double, double);
complex  pow(double, complex);
complex  pow(complex, int);
complex  pow(complex, double);
complex  pow(complex, complex);
```

Pow(aa,bb) raises *aa* to the power of *bb*. For example, to calculate *(1−i)**4:*

```
put( pow( complex(1,-1), 4) );
```

The output is *(−4,0)*.

```
overload log;
double   log(double);
complex  log(complex);
```

Log(zz) computes the natural logarithm of *zz*. *Log(0)* causes an error, and a huge value is returned.

```
overload exp;
double   exp(double);
complex  exp(complex);
```

Exp(zz) computes *e**zz*, *e* being 2.718281828...

```
overload sqrt;
double   sqrt(double);
complex  sqrt(complex);
```

Sqrt(zz) calculates the square root of *zz*.

The trigonometric functions available are:

```
overload sin;
double   sin(double);
complex  sin(complex);

overload cos;
double   cos(double);
complex  cos(complex);
```

Hyperbolic functions are also available:

```
overload sinh;
double   sinh(double);
complex  sinh(complex);

overload cosh;
double   cosh(double);
complex  cosh(complex);
```

Other trigonometric and hyperbolic functions, for example *tan()* and *tanh()*, can be written by the user using overloaded function names.

Efficiency

C's facility for overloading function names allows *complex* to handle overloaded function calls in an efficient manner. If a function name is declared to be overloaded, and that name is invoked in a function call, then the declaration list for that function is scanned in order, and the first occurrence of the appropriate function with matching arguments will be invoked. For further detail see reference 4. For example, consider the exponential function:

```
overload exp;
double   exp(double);
complex  exp(complex);
```

When called with a *double* argument the first, and in this case most efficient, *exp()* will be invoked. If a *complex* result is needed, the *double* result is then implicitly converted using the appropriate constructor. For example:

```
complex foo = exp(3.5);
```

is evaluated as

```
complex foo = complex( exp(3.5) );
```

and not

```
complex foo = exp( complex(3.5) );
```

Constructors can also be used explicitly. For example:

```
complex add(complex a1, complex a2) /* silly way of doing a1+a2 */
{
        return complex( real(a1)+real(a2), imag(a1)+imag(a2) );
}
```

Inline functions are used to avoid function call overhead for the simplest operations, for example, *conj()*, +, +=, and the constructors (See appendix A).

Acknowledgments

Phil Gillis supplied us with the complex functions used for the *cxp* package. Most of the functions presented here are modified versions of those. Stu Feldman provided us with valuable advice and some functions. Doug McIlroy's constructive comments led to a major re-write. Eric Grosse suggested the FFT function in Appendix B as an example.

Appendix A: Type *complex*

This is the definition of type *complex*. It can be included as <*complex.h*>. A *friend* declaration specifies that a function may access the internal representation of a *complex*. The names *put* and *get* need not be declared *overload* since that is done in *stdio.h*.

```
#include <stdio.h>
#include <errno.h>

overload cos;
overload cosh;
overload exp;
overload log;
overload pow;
overload sin;
overload sinh;
overload sqrt;
overload abs;

#include <math.h>
```

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

        friend  double  abs(complex);
        friend  double  norm(complex);
        friend  double  arg(complex);
        friend  complex conj(complex);
        friend  complex cos(complex);
        friend  complex cosh(complex);
        friend  complex exp(complex);
        friend  double  imag(complex);
        friend  complex log(complex);
        friend  complex pow(double, complex);
        friend  complex pow(complex, int);
        friend  complex pow(complex, double);
        friend  complex pow(complex, complex);
        friend  complex polar(double, double = 0);
        friend  double  real(complex);
        friend  complex sin(complex);
        friend  complex sinh(complex);
        friend  complex sqrt(complex);

        friend  complex operator+(complex, complex);
        friend  complex operator-(complex);
        friend  complex operator-(complex, complex);
        friend  complex operator*(complex, complex);
        friend  complex operator/(complex, complex);
        friend  int     operator==(complex, complex);
        friend  int     operator!=(complex, complex);

        void operator+=(complex);
        void operator-=(complex);
        void operator*=(complex);
        void operator/=(complex);
};
```

```
int put(FILE*, complex);
int get(FILE*, complex&);
inline int put(complex z)  { return put(stdout,z); }
inline int get(complex& z) { return get(stdin,z); }

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

inline complex operator-(complex a1,complex a2)
{
        return complex(a1.re-a2.re, a1.im-a2.im);
}

inline complex operator-(complex a)
{
        return complex(-a.re, a.im);
}

inline complex conj(complex a)
{
        return complex(a.re, -a.im);
}

inline int operator==(complex a, complex b)
{
        return (a.re==b.re && a.im==b.im);
}

inline int operator!=(complex a, complex b)
{
        return (a.re!=b.re || a.im!=b.im);
}

inline void complex.operator+=(complex a)
{
        re += a.re;
        im += a.im;
}

inline void complex.operator-=(complex a)
{
        re -= a.re;
        im -= a.im;
}
```

Appendix B: A FFT Function

Transcribed from Fortran as presented in "FFT as Nested Multiplication, with a Twist" by Carl de Boor in SIAM Sci. Stat. Comput. Vol 1 No 1, March 1980.

```
#include <complex.h>

void fftstp(complex*, int, int, int, complex*);

const NEXTMX = 12;
int prime[NEXTMX] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };

complex* fft(complex *z1, complex *z2, int n, int inzee)
/*
        Construct the discrete Fourier transform of z1 (or z2) in the
        Cooley-Tukey way, but with a twist.

        z1[before], z2[before].
        inzee==1 means input in z1; inzee==2 means input in z2
*/
{
        int before = n;
        int after = 1;
        int next = 0;
        int now;

        do {
                int np = prime[next];
                if ( (before/np)*np < before ) {
                        if (++next < NEXTMX) continue;
                        now = before;
                        before = 1;
                }
                else {
                        now = np;
                        before /= np;
                }
                if (inzee == 1)
                        fftstp(z1, after, now, before, z2);
                else
                        fftstp(z2, after, now, before, z1);
                inzee = 3 - inzee;
                after *= now;
        } while (1 < before)

        return (inzee==1) ? z1 : z2;
}
```

```
void fftstp(complex* zin, int after, int now, int before, complex* zout)
/*
        zin(after,before,now)
        zout(after,now,before)

        there are ample scope for optimization
*/
{
    double        angle = PI2/(now*after);
    complex omega = complex(cos(angle), -sin(angle));
    complex arg = 1;
    int     j;
    for (j=0; j<now; j++) {
        int ia;
        for (ia=0; ia<after; ia++) {
            int ib;
            for (ib=0; ib<before; ib++) {
                int in;
                     /* value = zin(ia,ib,now) */
                complex value = zin[ia + ib*after + (now-1)*before*after];

                for (in=now-2; 0<=in; in--) {
                 /* value = value*arg + zin(ia,ib,in) */
                    value *= arg;
                    value += zin[ia + ib*after + in*before*after];
                }
             /* zout(ia,j,ib) = value */
                zout[ia + j*after + ib*now*after] = value;
            }
            arg *= omega;
        }
    }
}
```

The main program below calls *fft()* with a sine curve as argument. The complete unedited output is presented on the next page. All but two of the numbers ought to have been zero. The very small numbers shows the roundoff errors. Since C floating-point arithmetic is done in double-precision these errors are smaller than the equivalent errors obtained using the published Fortran version.

```
#include <complex.h>

main()
/*
        test fft() with a sine curve
*/
{
        int i, n=26;
        complex *z1;
        complex *z2;
        complex *zout;
        extern complex* fft(complex*, complex*, int, int);

        z1 = new complex[n];
        z2 = new complex[n];

        printf("input: \n");
        for (i = 0; i < n ;i++) {
                z1[i] = sin(i*PI2/n);
                put(z1[i]);
                printf("\n");
        }

        errno = 0;
        zout = fft(z1, z2, n, 1);
        if (errno) printf("Cerror %d occurred\n",errno);

        printf("output: \n");
        for (i = 0; i < n ;i++) {
                put(zout[i]);
                printf("\n");
        }
}
```

```
input:
(0, 0)
(0.239316, 0)
(0.464723, 0)
(0.663123, 0)
(0.822984, 0)
(0.935016, 0)
(0.992709, 0)
(0.992709, 0)
(0.935016, 0)
(0.822984, 0)
(0.663123, 0)
(0.464723, 0)
(0.239316, 0)
(4.35984e-17, 0)
(-0.239316, 0)
(-0.464723, 0)
(-0.663123, 0)
(-0.822984, 0)
(-0.935016, 0)
(-0.992709, 0)
(-0.992709, 0)
(-0.935016, 0)
(-0.822984, 0)
(-0.663123, 0)
(-0.464723, 0)
(-0.239316, 0)
output:
(9.56401e-17, 0)
(-3.76665e-16, -13)
(9.39828e-17, 1.11261e-17)
(6.42219e-16, -4.20613e-17)
(7.37279e-17, 2.33319e-16)
(2.85084e-16, 2.87918e-16)
(4.03134e-17, 5.1789e-17)
(2.60865e-16, 6.78794e-17)
(-5.71667e-17, -3.86348e-17)
(2.76315e-16, 2.36902e-17)
(-6.43755e-17, -3.80255e-17)
(1.95031e-16, 9.77858e-17)
(1.49087e-16, -7.57345e-17)
(3.17224e-16, 1.64294e-17)
(1.49087e-16, 7.57345e-17)
(2.7218e-16, -4.03777e-17)
(-6.43755e-17, 3.80255e-17)
(4.93805e-16, 3.36874e-17)
(-5.71667e-17, 3.86348e-17)
(7.86047e-16, -4.11068e-18)
(4.03134e-17, -5.1789e-17)
(1.60788e-15, -1.06841e-16)
(7.37279e-17, -2.33319e-16)
(5.45186e-15, 2.42719e-16)
(9.39828e-17, -1.11261e-17)
(-1.12013e-14, 13)
```

Appendix C: Errors and Error Handling

These are the declarations used by the error handling:

```
int errno;
int complex_error(int, double);
```

The user can supply *complex_error()*. Otherwise a function that simply sets *errno* is used. The exceptions generated are:

cosh(zz):	
C_COSH_RE	|zz.re| too large. Value with correct angle and huge magnitude returned.
C_COSH_IM	|zz.im| too large. Complex(0,0) returned.
exp(zz):	
C_EXP_RE_POS	zz.im too small. Value with correct angle and huge magnitude returned.
C_EXP_RE_NEG	zz.re too small. Complex(0,0) returned.
C_EXP_IM	|zz.im| too large. Complex(0,0) returned.
log(zz):	
C_LOG_0	zz==0. Value with a large real part and zero imaginary part returned.
sinh(zz):	
C_SINH_RE	|zz.re| too large. Value with correct angle and huge magnitude returned.
C_SINH_IM	|zz.im| too large. Complex(0,0) returned.