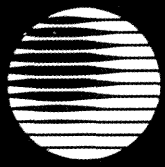

UNIX™ System V
AT&T C++ Translator

Release Notes

Unix System V AT&T Translator Release Notes: Abridged Version

This is an abridged version of the UNIX System V AT&T C++ Translator Release Notes. AT&T's original release notes contain material and many references that are not relevant to your use of ADVANTAGE C++, Lifeboat's implementation of the translator for the MS/PC DOS environment. Therefore, to avoid potential confusion in the use of ADVANTAGE C++, we have deleted those sections for this reprinting of AT&T's Release Notes.

Printed in the U.S.A.

Contents

Introduction

Technical Tips

Language Overview

Complex Arithmetic in C++

INTRODUCTION

The AT&T C++ Translator

The AT&T C++ Translator, Release 1.0, translates C++ source code to C source code. It supports the C++ programming language as described in Bjarne Stroustrup's *The C++ Programming Language*. When you use the type checking and data abstraction features of C++, the translator detects errors that would otherwise go unnoticed until run-time.

After translating a C++ program, you can compile it with most C compilation systems that support long variable names and structure assignment returns. You need only a single command, **CC**, to invoke the translator and the C compilation tools required to produce executable C++ programs. In addition, you can use the options with the **CC** command to alter the standard compilation process, for example, to invoke an optimizer.

The translator includes C++ library functions for complex arithmetic, stream I/O, tasking, and other operations. It also has **#include** files for use with C++ programs.

This highly-portable translator runs on many machines, including the AT&T 3B Computers, running UNIX System V, Release 2.0 or later. Running on UNIX System V, the translator makes many of the system's programming tools and libraries available to you.

Release 1.0 of the translator supercedes Release E, an earlier version made available to universities in December 1984. This new release supports changes, enchancements, and new features added to the C++ language since Release E was introduced.

The C++ Programming Language

C++ is a general purpose programming language designed at AT&T Bell Laboratories. An extension of the C programming language, C++ retains C's efficiency and flexibility. In addition, C++ offers facilities for designing better interfaces among and within program modules. These facilities support stronger type checking than C and extensive data abstraction.

This section briefly describes compatibility with C and type checking and data abstraction features in C++. If you are interested in learning more about these features, you can refer to *The C++ Programming Language*.

Compatibility with C

Retaining compatibility with C served as a major design criterion for C++. The basic syntax and semantics of the two languages are the same. If you are familiar with C, you can program in C++ immediately. Then, as your need for stronger type checking and data abstraction grows, you can begin to apply the C++ facilities that support these programming activities.

C++ preserves C's efficient interface to computer hardware. That is, C++ has the same types, operators, and other facilities defined in C that usually correspond directly to components of computing equipment. You can use these facilities to write code that gets in close to the hardware, to manipulate bits and use register variables, for instance, to make optimal use of the hardware at run-time.

C++ also preserves the C facilities for designing interfaces among program modules: functions, global data, and header files. (In this document, *module* refers to a file containing a variable or function declaration, a function definition, or a number of these or similar items logically grouped together. Several modules may make up one program.) These facilities are essential when you develop an application of any size, but particularly a large or complex one. The differences between C++ and C become evident when you use these facilities, because C++ has enhanced them and added related facilities.

Finally, C++ modules are link-compatible with C modules. You may, therefore, use all C libraries with your C++ programs.

Type Checking Features

You declare functions in C++ just as you do in C, except that you may enhance them to support type checking of arguments and overloading. *Type checking* is an activity a translator or a compiler carries out to ensure that the operations defined in your program are applied to data of the correct type. In C++, type checking is manifested in automatic and explicit type conversions and certain notational conveniences. The emphasis on type also helps the translator detect more errors at compile-time, when they are easy to correct.

Overloading refers to giving functions the same name when they perform the same operations on objects of different types. You may have, for instance, overloaded functions named **print** for printing integers, character strings, and so on.

Functions in C++ support type checking through their arguments, which, unlike those in C, can be typed. Here's an example of a C++ function with argument types:

```
void set_date(int, int, int);
```

This function takes three integer arguments to set the date. Here's the same function declaration that has been overloaded to accept integer and character string arguments:

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

These typed arguments help the translator better manage interfaces among modules; and the arguments help you design either restrictive or permissive interfaces. On the one hand, for instance, the translator checks the argument types and, if necessary, does type conversions to ensure that functions manipulate only the data for which they are designed. This makes the interface between a calling and a called function safer. On the other hand, the translator can use argument types to resolve overloads. By checking the types of arguments passed in a call to **print**, for example, the translator chooses the correct overloaded function. This use of type information makes the interface between a calling and a called function more flexible.

Data Abstraction Features

In addition to providing stronger type checking, C++ gives you extensive facilities for data abstraction. The fundamental idea of *data abstraction* is to separate the representation of a data object from the specifications essential for its correct use. For example, the C++ type **float** is an abstraction for real numbers. You add or assign values to objects of this type without concern for how the numbers are represented by the translator. That's because addition and assignment are specified by the language as correct uses of **float**.

C++ supports data abstraction beyond that defined by the language by letting you define new type called *classes*. You can use classes as conveniently as built-in types. They are a lot like structures, except that they can have function members, as well as data members. The function members usually specify how the data members can be used. That is, function members specify the correct use of class objects.

Here's an example of a class definition. It includes seven functions, some of which serve special purposes. The definition shows some other features of C++ (for example, a friend function), all of which are discussed in the "Language Overview:"

```
class Course {
    int credits, points;
    char *name, *instructor;
    float grade();
public:
    Course();
    ~Course();
    void set_date( int = 0, int = 0, int = 0 );
    void set_date( char* = "NA" );
    int operator=( int );
    friend void report();
};
```

The label **public** separates the class into two parts. The part preceding the label is private; the other part is public. Use of the private members is restricted to the other members (or, more exactly, the friend and member functions). Because **grade** is a private member, only the other members and friends have access to it.

Classes, like functions with argument types, help you design more expressive interfaces. Among other things, they let you hide data, guarantee the initialization of data, and overload operators, as shown in the example. All these facilities can improve a program.

Other Features

Other features of C++ distinguish it from C. For example, you can declare variables almost anywhere in a block, not just at the beginning of the block. As a result, you can locate variables with the statements that use them. C++ also gives you new operators for controlling memory management. These features, and those supporting type checking and data abstraction, can improve productivity in your software development.

TECHNICAL TIPS

This section contains information about changes you may need to make to the C programs and header files that you intend to use with the AT&T C++ Translator. Many of these changes are related to the C++ language's support of type checking. For example, you have to give argument types to all existing function declarations.

This section also offers some advice about using the **CC** command.

New Keywords

The following words are reserved keywords in C++:

class	**inline**	**overload**	**const**
public	**friend**	**new**	**virtual**
delete	**operator**	**this**	

You have to change variables or functions with these names. **signed** and **volatile** are not allowed as identifiers because of their expected use as keywords in the future.

Functions

Argument Types

The most basic change involving functons is adding argument types to function declarations. For instance, the library function **fseek** would be declared in C:

```
int fseek();
```

This function takes a **FILE***, a **long** , and an **int**. So, the C++ declaration is:

```
int fseek( FILE*, long, int);
```

Varying Argument Types

Some functions, like **printf**, take varying numbers of arguments. Their declarations should have an ellipsis (...) where the arguments can vary. Arguments from this place on are not type checked:

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

Function Declarations

In C, undeclared functions are assumed to return an integer. Therefore, many functions are not declared at all. If you want the C++ translator do type checking, you have to declare all functions, regardless of return type.

Overloaded Function Names

The name encoding scheme for generating overloaded function names has changed. The change should not break any of your code. However, you do have to recompile all C++ programs that you compiled under Release E when you move them to the new translator. If you don't recompile these programs and then try to combine object files produced by the new and a previous version of the translator, you may have undefined references at load-time.

Structures

Structure Tags in Declarations

In C++, structure tag names are also type names. Once you define a structure, the structure tag name may be used in a declaration; the keyword **struct** is not needed (but is allowed):

```
struct utmp {
  /*lots of stuff*/
};
utmp * getutid ( utmp *); /* function takes a
                pointer to a utmp, and returns a
                pointer to a utmp*/
```

Conflicts with Structure Tag Names

Because structure tag names are type names, these names may conflict with other names. In cases where they do conflict, the name that the translator sees last hides the name that it sees first, unless the explicit **struct** is provided. For instance, a structure can have the same name as a function:

```
struct stat {
     int i;
};
a = stat( "filename", &statbuf);
```

This code generates an error in C++, because the call to **stat** looks like a constructor call for the structure. To avoid this, make sure that the function is declared after the structure:

```
struct stat{
    int i;
};
int stat ( char*, struct stat*);
```

Note that C code will still work with this declaration, because any use of the structure in C will have the explicit **struct** provided.

A structure name can also conflict with a variable name:

```
struct stat{
    int i;
};
int stat;
```

You should declare the variable name after the structure with the same name.

LANGUAGE OVERVIEW

This section is an overview of the major features of the C++ programming language. Examples and explanations of each feature are included. When you're done, turn to *The C++ Programming Language* for a complete discussion.

Comments

C++ has two notations for comments. Comments may begin with the characters /* and end with */, as they do in C. Or comments may begin with the characters //, and end at the end of the line on which the // occurs. You can comment out (or temporarily remove) a section of code from a program using both comment notations. By using // exclusively for comments in the source, you can then use /* and */ to surround the sections of source code to be commented out. For example, if the following lines appeared in a program, **i** would not be initialized and **j** would not be assigned to:

```
/*
int i = 5;  //initialize i
j = i;  //assign i to j
*/
```

Function Argument Types

C++ functions may specify function argument types. Some properly declared C++ functions with argument types follows:

```
void step(int);
float min(float, float);
extern int strcpy (char* to, const char* from);
```

If **strcmp** were declared as follows:

```
extern int strcmp(to, from);
```

the translator would produce an error message stating that argument types are expected. Other than having argument types, each C++ declaration is similar to a C function declaration. For instance, all C++ declarations include a name and return type. In addition, the declaration of **strcmp** includes argument names and a storage class. When you compile functions with argument types, the translator performs type checking and type conversion. That is, the translator compares the argument types with the parameter types in the function definition each time the function is called. The types must match for the functions to compile.

For example, some calls to the C++ function **min** follow:

```
float min(float, float);

min(7.0, 5.0);
min(7, 5);
min("Seven", 5.0);  //error
```

The translator accepts the first two calls. It performs type conversions for the integer arguments in the second call, and actually passes the floating point numbers 7.0 and 5.0. However, the translator does not accept the third call, because the type of the first argument is a character string. A floating point number is expected and type conversion is not possible, so the translator produces an error message.

Default Arguments

To account for missing arguments in a function call, function declarations may specify default expressions for the arguments. You declare these default expressions by initializing the arguments; the initial values are called *default arguments*. Two default arguments are shown in the following example:

```
void chemical( int = 0, char* = "Missing");
```

When a call to this function is missing an argument, the default argument is substituted in its place. Function definitions may also specify default arguments.

Variable Number of Arguments

In addition to specifying argument types, a C++ function declaration may specify that a variable number of arguments is accepted. You declare a function with variable arguments by adding an ellipsis (...) to the end of the declaration of the function's argument list. The ellipsis instructs the translator to accept any number of arguments of any type after that point in the list in a function call. For example,

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

declares that **fprintf** is a function that returns an integer and may have a variable number of arguments, the first two of which must be of the types **FILE** and **char***, respectively. Functions with unspecified arguments are most useful for specifying an interface to library functions for which you cannot anticipate every call.

Classes

C++ lets you define new types called **classes**, which resemble C structures. For example, developing an application for a publisher or bookstore, you could define a class to represent a book with data members representing title, author, and so on:

```
class Book {
    char *title, *author, *publisher;
    int copyright;
    float price;
};
```

You can declare many variables of the class after defining it:

```
Book novel;
Book play;
```

The class provides a more meaningful definition of a book than any built-in type can by itself. The definition can make your application easier for you to modify and for others to understand.

Member Functions

The class **Book** is incomplete as it is defined above. It specifies only the representation of data objects. In comparison, a built-in type like **int** or **float** also specifies how objects or variables of the type can be used.

To make user-defined types as complete and convenient as built-in types, C++ lets you define a set of functions to manipulate objects of a class like **Book**. For instance, you may define functions to list all books by a particular author or to increase a price. These functions are called *member functions* and are declared as members of the class:

```
class Book {
    char *title, *author, *publisher;
    int copyright;
    float price;
public:
    void search(char*);
    void markup(float);
};
```

The keyword **public** separates the class into two parts. The members in the first part can only be used by those in the second part. In this example, only the two member functions can manipulate **Books**.

Not all classes separate data members and function members as shown in the definition of **Book**. The following class has public data members and a private function member:

```
class Paint {
    int percent_tint;
    short time;
    void tint(int, int, short);
public:
    int color, finish;
    void mix ();
};
```

Notice that the function **tint** can be called only by the function **mix**.

Data Hiding

Using classes, you can hide the representation of data and restrict access to data. As a result, you can use classes in the same way as built-in types.

That is, to declare and use objects of the type **float**, you need not know how the objects are represented in storage. All you need to know is the name of the type and the operations that are allowed. Using floating point objects, you can add or assign values to them without concern for their representation. The representation of the objects is hidden.

Similarly, C++ lets you

- use a class like **Book**, while ignoring the details of how an object of this class is represented. All you have to do is remember that books have authors, prices, and so on, and that authors can be searched for and prices can be marked up.

- design large or complex applications with many pieces that use objects of a class, whose representation needs only to be defined in one place.

- change the representation of a class without affecting the rest of a program.

Restricting access to the representation of data objects through the member functions offers you other advantages:

- When debugging a program, you can readily trace an error involving the private members of a class to the member functions or friend functions.
- You can protect the integrity of the objects by limiting how other users manipulate them.
- Other users can examine the definitions of the member and friend functions to learn how to use the class.

Constructors and Destructors

C++ lets you guarantee that class objects are properly initialized through the use of *constructors*. Constructors are member functions designed explicitly to initialize objects. A constructor sets up and assigns a value in storage when a class object is declared. More often than not classes in C++ have constructors.

Many classes also have *destructors*. A destructor ensures that storage is released, counters are reset, and other maintenance takes place when class objects are destroyed (for example, when a variable goes out of scope).

A constructor has the same name as its class: a destructor for class **X** is named **~X**. For example, the following class **Date** has a constructor and destructor:

```
class Date {
    int day, month, year;
    void set_date( int, int, int );
public:
    Date( int = 0, int = 0, int = 0);   // constructor
    ~Date();                            // destructor
};
```

As shown, the constructor **Date** takes three integer arguments, all of which have default values. The destructor takes no arguments. You can declare objects of the class as follows:

```
Date today = Date( 27, 1, 1958 );
Date halloween( 31, 10);
```

Each time you define an object of a class with a constructor, a constructor is im-

plicitly called. Because initialization occurs when the object is created, constructors eliminate the need to declare the object first and later initialize it. Constructors also save you from forgetting to initialize objects or from doing so twice.

Friend Functions

Functions that are not members of a class may have access to class objects, if the functions are declared as friend functions. A function becomes a friend to a class when you declare the function with the keyword **friend** in a class definition:

```
class Book {
    char *title, *author, *publisher;
    int copyright;
    float price;
    friend void search(char*);
    friend void markup(float);
};
```

A function may be friend to more than one class. The function needs simply to be declared in the private or the public part of the class definitions. Other than their ability to access private class members, friend functions are just like other C++ functions.

Derived Classes

Base and Derived Classes

C++ lets you derive a class from another class. Using the class **Book** as a *base* class, you can define derived classes, as follows:

```
class Cook:Book { /* Cook's unique members*/};
class Fict: Book { /* Fict's unique members */};
```

Derived classes inherit all members of the base class. Deriving classes helps you in two ways. First, you can define details common to many potential derived classes in a base class. The base class can be written and compiled just once, stored in a header file, and then used to derive new classes with additional data or functions. Class **Cook** might have an additional **char*** to specify the type of cooking described in a book. Second, a derived class supplies a specialized interface to a base class.

Virtual Functions

C++ also lets you declare functions in a base class that can be redefined in each class derived from it. These functions are called *virtual functions*. **Book**, in the following example, has a virtual function for printing an index. It is defined as part of the class and is called by the member function **print**, which prints the entire contents of a book:

```
class Book {
    char *title, *author;
    //...
public:
    void print();
    virtual void print_index();
};

void Book::print(){
    //...       //print the title page,dedication,etc.
    print_index();
}
```

Cook and **Fict** can then define their own versions of the function **print_index**. Given an object **japanese** of the class **Cook**, the translator will make sure the **print_index** function applied to the object is the **Cook** version. If **Cook** does not have its own version of the function, the version for **Book** is used.

Overloaded Operators

Classes can have functions that assign special user-defined meanings to the standard C++ operators when they are applied to class objects. (Operators cannot be redefined for built-in types.) These functions are called *overloaded operator* functions. Designing an application using complex numbers, you could overload the **operator+** to handle complex additon; and designing an application for recipes, you could define the same operator to add one cup.

The name of an operator function is the keyword **operator** followed by the operator itself, such as **operator+**. You can declare and call an operator function in the same way you call any other function; and an operator function may be a member function.

Conversion Operators

Sometimes you want to convert a value from one type given the value of another. In addition to using casts like those in C, you can use conversion operators in C++. Conversion operators are member functions with the same name as their destination types. Every conversion operator is declared like an overloaded operator function with the keyword **operator**:

```
class Count {
  int n;
public:
  Count(int x) { n = x: }  //constructor
  operator int() { return n; } //conversion operator
};
```

In this example, the class **Count** has a conversion operator that defines a conversion from **Count** to **int**.

The type conversion possible with C++ conversion operators surpasses that provided by casts. Notice that the conversion from **Count** to **int** is from a user-defined type to a built-in type. Conversion operators also make conversions from new types to old types possible without modifying the declarations of the old types.

Overloaded Functions

Consider again this definition of the class **Date**:

```
class Date {
  int day, month, year;
  void set_date( int, int, int );
public:
  Date( int = 0, int = 0, int = 0);  // constructor
  ~Date();                 // destructor
};
```

Users of this class would probably like several ways of initializing an object. You can redefine the class to include several constructors to give users this flexibility:

```
class Date {
    //...
public:
    Date( int, int, int );
    Date( char* );  //date in string representation
    Date( char*,int); //month as a string, day as
                              an integer
    ~Date();
};
```

Notice that each of the constructors has the same name.

Giving functions the same name can be appropriate, when the functions perform the same operations on objects of different types. This use of a name is called *function name overloading*. The translator knows which constructor to select based on the argument types that are passed. For example, the translator selects the above constructors in the order shown for the following function calls:

```
Date due(10, 26, 85);
Date test( "January 1, 1986" );
Date birthday("January", 27);
```

Constructors are a special case of overloaded functions. To overload other functions in C++, you must use the keyword **overload**, in the function declaration:

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

or

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

When **print** is called, the translator

- checks the arguments passed to determine their type
- searches for a match of the argument list among the **print** functions
- selects the exact match (or uses built-in or user-defined type conversion to match).

Overloaded function names give you more than a notational convenience.

Because of the matching rules, you can also ensure that the simplest algorithm (function) is used where the efficiency or precision of computations differs significantly for the types involved. With an overloaded function for calculating square roots, for example, you can ensure efficiency with integers and precision with complex numbers.

The New and Delete Operators

You can declare a named object in C++ to be static or automatic. A static object exists throughout program execution; an automatic object is local to a function (or block of a program) and exists only during the execution of the function.

You may sometimes want to make use of an object created by a function after leaving the function. To make this control possible, C++ provides the operator **new** to create objects and the operator **delete** to destroy them. Using these operators, you can control the lifetime of an object and allocate storage just for the time the object is needed. Because they are built-in to the language, **new** and **delete** are easier to use than UNIX system tools **malloc** and **free**.

Inline Functions

Small functions frequently called can increase a program's run-time. C++ lets you declare these functions to be *inline expanded.* This means that the translator will try to generate the code for the function at the place it is called. When used with small functions, **inline** can reduce the run-time.

You can make a function inline by declaring it with the scope **inline:**

```
inline int max(int a, int b)
    { return a > b ? a : b; }
```

Member functions become inline when they are defined within the definition of their classes.

Inline functions behave the same way as other functions, so declaring a function inline does not change the meaning of a program. In fact, the compiler may choose not to expand a function inline (for instance, if the function is recursive). Unlike C preprocessor macros, you can use inline functions for multiple statements and to avoid double argument evaluation.

Summary

This language overview highlights the major features of C++. Classes, member and friend functions, constructors and destructors, overloaded operators, derived classes, and virtual functions account for the language's support of data abstraction. Most of these features also play a role in the strong type checking provided by C++. Of course, function argument types are one of the most visible and powerful examples of type checking support in C++; and overloaded functions show how to make good use of the argument types.

The operators **new** and **delete** and inline functions give a flavor of the language features that are not directly related to data abstraction or type checking. There are others. You should turn to *The C++ Programming Language* now that you have read this overview.

All of the C++ language features were designed to help you express concepts clearly in programs and to design programs made up of manageable pieces. When used well, C++ can improve productivity in software development for both systems programmers and applications programmers.

COMPLEX ARITHMETIC IN C++

ABSTRACT

This abstract 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 it provides. For example:

```
#include <complex.h>

main () {
    complex xx;
    complex yy = complex(1,2.718);
    xx = log(yy/3);
    cout << 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 date abstraction facilities in 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.

This document is based on an AT&T Bell Laboratories Computer Science technical report by Leonie V. Rose and Bjarne Stroustrup.

Introduction

The C++ language does not have a built-in data type for complex numbers, but it does provide language facilities for defining new data types. 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 a language. First, the type **complex** could be directly supported by the 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 expressions involving only built-in data types. A third approach was used to implement type **complex**; it was specified as a user-defined type. 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 operators **>>** (get from) and **<<** (put to). The initialization functions and **>>** accept a Cartesian representation of a **complex**. The functions **real ()** and **imag()** return the real and imaginary part of a **complex**, respectively, and **<<** prints a **complex** as **(real,imaginary)**. The internal representation of a **complex** is, 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.

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

*Such a function is called a constructor. A constructor for a type always has the same name as the type itself.

initializing **complex** variables, and will convert the arguments for 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:

```
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 operators **>>** (get from) and **<<** (put to). They are declared like this using the facility for overloading function operators:

```
ostream& operator<<(ostream&, complex);
istream& operator>>(istream&, complex&);
```

When **zz** is a complex variable **cin>>zz** reads a pair of numbers from the standard input stream **cin** into **zz**. The first number of the pair in interpreted as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The expression **cout<<zz** writes **zz** to the standard ouput stream **cout**. For example:

```
void copy(istream& from, ostream& to)
{
    complex zz;
    while (from>>zz) to <<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 **operator >>.**

A user who does not like the standard implementation of **<<** and **>>** 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);
cout << real(cc) << "+" << imag(cc) << "*i";
```

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)
doube angl = arg(cc);                   // angl = PI/4
cout << "(m=" << magn << ", a=" << 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 product 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 product a result for every possible argument. If it is not possible to product 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 B.

```
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 intended 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**:

```
cout << 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 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 functions call overhead for the simplest operations, for example, **conj()**, **+**, **+=**, and the constructors (See Appendix A).

Acknowledgments

Phil Gillis supplied 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 rewrite.

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 standard header file **<stream.h>** is included to allow declaration of the stream I/O operators **<<** and **>>** for complex numbers.

```
#include <stream.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);
};
```

```
ostream& operator<<(ostream&, complex);
istream& operator>>(istream&, complex&);
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: Errors and Error Handling

These are the declarations used by the error handling:

```
int errno;
int complex_error(c_exception &);
```

complex_error is invoked by functions in the complex arithmetic package when errors are detected. Users may define their own procedures for handling errors, by including a function named **complex_error** in their programs. **complex_error** must be of the form described above. When an error occurs, a pointer to the exception structure will be passed to the user-supplied **complex_error** function. This structure, which is defined in the **<complex.h>** header file is as follows:

```
struct c_exception {
    int type;
    char *name;
    complex arg1, arg2, retval;
};
```

The element type is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

SING	**argument singularity**
OVERFLOW	**overflow range error**
UNDERFLOW	**underflow range error**

The element name points to a string containing the name of the function that incurred the error. The variables **arg1** and **arg2** are the arguments with which the function was invoked. **retval** is set to the default value that will be returned by the function unless the user's **complex_error** sets it to a different value.

If the user's **complex_error** function returns non-zero, no error message will be printed, and **errno** will not be set.

If **complex_error is** not supplied by the user, a default **complex_error** will be supplied that will set the return value according to the table on the following page. In every case, **errno** is set to **EDOM** or **ERANGE** and the program continues.

DEFAULT ERROR HANDLING PROCEDURES			
Types of Errors			
Type	SING	OVERFLOW	UNDERFLOW
errno	EDOM	ERANGE	ERANGE
EXP:			
re overflow/underflow	-	±H.±H	0.0
|im| too large	-	0.0	-
LOG:			
arg = (0.0)	M.H.0	-	-
SINH:			
|re| too large	-	±H.±H	-
|im| too large	-	0.0	-
COSH:			
|re| too large	-	±H.±H	-
|im| too large	-	0.0	-