VMS Software

VSI OpenVMS

C++ Class Library Reference Manual

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI C++ Version 7.4-6 for OpenVMS I64
VSI C++ Version 7.4-8 for OpenVMS Alpha

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

C++ Class Library Reference Manual

VMS Software

Legal Notice

Table of Contents

Preface

This manual describes the library of classes supplied with VSI C++ for OpenVMS systems. It contains detailed information on members of these classes (including member functions) and information on other associated functions, variables, and macros.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for experienced programmers who have a basic understanding of the VSI C++ language, and who are using VSI C++ with the OpenVMS operating system in either a single or multiple platform environment. Some familiarity with the operating system is assumed.

3. Document Structure

This manual consists of an introductory chapter and 10 chapters describing each of the packages of predefined classes supplied with the VSI C++ compiler.

4. Related Documents

The following documents contain information related to this manual:

- *The C++ Programming Language, 3nd Edition* provides an exhaustive introduction to the C++ programming language, and includes the text but not the annotation of *The Annotated C++ Reference Manual*.
- *C++ Installation Guide for OpenVMS* describes how to install VSI C++ on your system.
- *C++ Command Reference Pages* provides references for C++ commands and libraries.
- *Musser and Saini, STL Tutorial and Reference Guide, Addison-Wesley, 1995* describes how to use the Standard Templates Library (STL).
- *The Annotated C++ Reference Manual* contains the definitive language description of C++.
- *C Language Reference Manual* provides a complete technical description of the C language, as specified by the ANSI X3J11 committee. This manual also fully describes all extensions to this standard implemented in C.
- *Ladebug Debugger Manual* describes how to use DIGITAL's Ladebug debugger.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have VSI OpenVMS support contracts through VSI can contact `<support@vmssoftware.com>` for help with this product.

7. Conventions

The conventions found in the following table are used in this document.

Convention	Meaning

`class complex {`
`.`
`.`
`.`
`};`
	A vertical ellipsis indicates that some intervening program code or output is not shown. Only the more pertinent material is shown in the example.

`, ...`	A horizontal ellipsis in a syntax description indicates that you can enter additional parameters, options, or values. A comma preceding the ellipsis indicates that successive items must be separated by commas.
`The complex class ...` `The get() function ...`	Monospaced type denotes the names of VSI C++ language elements, and also the names of classes, members, and nonmembers. Monospaced type is also used in text to reference code elements displayed in examples.
italic	Italic type denotes the names of variables that appear as parameters or in arguments to functions.
boldface	Boldface type in text indicates the first instance of terms defined in text.
UPPERCASE, lowercase	UNIX operating system differentiates between uppercase and lowercase characters. Literal strings that appear in examples, syntax descriptions, and function definitions must be typed exactly as shown.

Chapter 1. Overview

The VSI C++ Class Library is a set of headers and other files implementing a collection of basic VSI C++ classes. In the library, these classes are arranged in functionally related groups called **packages**.

The VSI C++ Class Library makes use of other run-time libraries.

Note

Identifiers beginning with `cxxl` or `CXXL` are reserved for the VSI C++ Class Library and should not be used by customer programs except as specified in this manual.

1.1. Thread Safe Programming

Developers of multithreaded applications should note the following:

- Internal class library data is thread safe; multiple threads can access the VSI C++ Class Library simultaneously without compromising the integrity of the internal data.
- The predefined stream objects, `cerr`, `cin`, `clog`, and `cout` are thread safe. However, you need to provide synchronization around sequences of operations on these objects. For more information on synchronizing access to the predefined stream objects, see Chapter 4.
- User-defined objects are not thread safe; users must provide synchronization for such objects if they are shared between threads. For more information on synchronizing access to user-defined objects, see Chapter 6.
- The `ios` class member function `sync_with_stdio()` is not thread safe; if your application calls this function, the call must come before any threads use the predefined stream objects: `cerr`, `cin`, `clog`, or `cout`.
- Generation of error messages within the vector package is not thread safe; the package uses static data members to handle the current error message and there is no synchronization between threads. VSI recommends that you define a single Mutex object to synchronize all use of the vector package.
- The task package is not thread safe; only one task can execute at a time.

1.2. Using RMS Attributes with iostreams

The Class Library class `fstream` constructors and `open()` member function do not support different RMS attributes, for example, creating a stream-lf file.

To work around this restriction, use the C library `creat()` or `open()` call, which returns a file descriptor, and then use the `fstream` constructor, which accepts a file descriptor as its argument. For example:

```
#include <fstream.hxx>

int main()
{
  int fp;
```

```
  // use either creat or open
  //if ( !(fp= creat("output_file.test", 0, "rfm=stmlf")) )

  if ( !(fp= open("output_file.test", O_WRONLY | O_CREAT | O_TRUNC , 0,
"rfm=stmlf")) )
    perror("open");

  ofstream output_file(fp); // use special constructor which takes
                            // a file descriptor as argument
  // ...
}
```

1.3. Class Library Restrictions

The following are restrictions in the VSI C++ Class Library:

- No Class Library support for 128-bit long doubles

  The Class Library does not include support for 128-bit long doubles.

- Conflict with redefinition of `clear()`

  If your program includes both `<curses.h>` and `<iostream.hxx>`, VSI C++ might fail to compile your program because `clear()` is defined by both header files. In `<curses.h>`, `clear()` is defined as a macro whereas in `<iostream.hxx>` `clear()` is defined as a member function.

  Workarounds:

  If your program does not use either `clear()` or uses the `clear()`, include the `<iostream.hxx>` header first, followed by `<curses.h>`.

  If your program uses the `ios::clear()` function, undefine the `clear()` macro directly after the `#include <curses.h>` statement.

Chapter 2. complex Package

The complex package provides ways to perform arithmetical operations, such as initialization, assignment, input, and output, on complex values (that is, numbers with a real part and an imaginary part). Additionally, this package supports operations that are unique to complex values, such as principal argument operations, conjugate operations, and conversions to and from polar coordinates.

With the `c_exception` class and its `c_exception` function, the complex package also provides a mechanism for reporting and handling complex arithmetical errors.

Global Declarations

Global Declarations — These declarations are used by the complex package but they are not members of the `complex` class.

Header

```
#include <complex.hxx>
```

Alternative Header

```
#include <complex.h>
```

Descriptions

```
typedef int (*cxxl_p_complex_error_t)(c_exception &error_information);
static const complex_zero (0, 0);
cxxl_p_complex_error_t set_complex_error(cxxl_p_complex_error_t
p_complex_error);
```

Type

`cxxl_p_complex_error_t`

Is the type of the `complex_error` function.

Data

`static const complex_zero (0, 0)`

Is a constant object of type `complex` and value 0 created in each module that uses the complex package.

Function

`cxxl_p_complex_error_t set_complex_error (cxxl_p_complex_error_t p_complex_er`

Causes the function pointed to by `p_complex_error` to be called instead of the `complex_error` function on subsequent complex arithmetical errors. If `set_complex_error()` previously has not been called, then it returns 0; otherwise, it returns the address of the last function passed to it.

See the section called “Other Function” of `c_exception` class for a description of the error-handling function.

complex class

complex class — Provides a representation of, and lets you perform operations on, complex values.

Header

```
#include <complex.hxx>
```

Alternative Header

```
#include <complex.h>
```

Declaration

```
class complex
{
    friend complex   polar(double, double = 0);
    friend double    abs(const complex &);
    friend double    norm(const complex &);
    friend double    arg(const complex &);
    friend double    arg1(const complex &);
    friend complex   conj(const complex &);
    friend complex   sin(const complex &);
    friend complex   sinh(const complex &); // c_exception OVERFLOW
    friend complex   cos(const complex &);
    friend complex   cosh(const complex &); // c_exception OVERFLOW
    friend complex   tan(const complex &);
    friend complex   tanh(const complex &);
    friend double    imag(const complex &);
    friend double    real(const complex &);
    friend complex   log(const complex &); // c_exception SING
    // c_exception OVERFLOW UNDERFLOW
    friend complex   exp(const complex &);
    friend complex   pow(double, const complex &);
    friend complex   pow(const complex &, int);
    friend complex   pow(const complex &, double);
    friend complex   pow(const complex &, const complex &);
    friend complex   sqrt(const complex &);
    friend complex   sqr(const complex &);
    friend complex   operator-(const complex &);
    friend complex   operator+(const complex &, const complex &);
    friend complex   operator-(const complex &, const complex &);
    friend complex   operator*(const complex &, const complex &);
    friend complex   operator/(const complex &, const complex &);
    friend int       operator==(const complex &, const complex &);
    friend int       operator!=(const complex &, const complex &);
    friend ostream   &operator<<(ostream &, const complex &);
    friend istream   &operator>>(istream &, complex &);

public:

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

```
                    complex();

    inline complex   &operator-=(const complex &);
    inline complex   &operator+=(const complex &);
    complex          &operator*=(const complex &);
    complex          &operator/=(const complex &);
};
```

Description

This class contains methods to perform complex value operations. These include arithmetical, assignment, and comparison operators for complex values; Cartesian and polar coordinates; mixed-mode arithmetic; and mathematical functions for complex values equivalent to standard mathematical functions.

Exception Handling

When a complex arithmetical error is detected, a `c_exception` object is created with one of the following values for `type`:

Value	Error Description
`OVERFLOW`	Value too large to be represented
`SING`	Function undefined for argument
`UNDERFLOW`	Value too small to be represented

This object is then passed to the `complex_error` function (see c_exception class).

Constructors and Destructors

`complex()`

Constructs and initializes a complex value to 0.

`complex(double x, double y = 0)`

Constructs and initializes a complex value from Cartesian coordinates.

Overloaded Operators

`complex operator + (const complex &z1, const complex &z2)`

Returns the arithmetical sum of the complex values *z1* and *z2*.

`complex operator – (const complex &z1)`

Returns the arithmetical negation of a complex value.

`complex operator – (const complex &z1, const complex &z2)`

Returns the arithmetical difference of complex values. That is, *z2* is subtracted from *z1*.

`complex operator * (const complex &z1, const complex &z2)`

Returns the arithmetical product of the complex values *z1* and *z2*.

`complex operator / (const complex &z1, const complex &z2)`

Returns the arithmetical quotient of complex values. That is, *z1* is divided by *z2*.

`inline complex &operator += (const complex &z1)`

Assigns the arithmetical sum of complex values to the complex object on the left side of an equation. That is, *z1+=z2* is equivalent to *z1=z1+z2*.

`inline complex &operator –= (const complex &z1)`

Assigns the arithmetical difference of two complex numbers to the complex object on the left side of an equation. That is, *z1–=z2* is equivalent to *z1=z1–z2*.

`complex &operator *= (const complex &z2)`

Assigns the arithmetical product of two complex numbers to the complex object on the left side of an equation. That is, *z1*=z2* is equivalent to *z1=z1*z2*.

`complex &operator /= (const complex &z2)`

Assigns the arithmetical quotient of two complex numbers to the complex object on the left side of an equation. That is, *z1/=z2* is equivalent to *z1=z1/z2*.

`ostream &operator << (ostream &s, const complex &z1)`

Sends a complex value to an output stream in the fo`(real,imag)`rmat. It returns the left argument *s*.

`istream &operator >> (istream &s, complex &z1)`

Takes a complex value from an input stream. The numbers may be of the forms `(real,imag)` or `(real)`, where `real` and `imag` are what the iostream package accepts for parameters of type `double`. The iostream package also determines how to handle white space. This operator returns the left argument *s*. The following input format omissions will cause an error:

- Parenthesis missing before a complex value
- Comma missing before the imaginary part of a complex value, if any
- Parenthesis missing after the complex value

`int operator == (const complex &z1, const complex &z2)`

Compares two complex values and returns a nonzero value if the two numbers are equal; otherwise, it returns 0.

`int operator != (const complex &z1, const complex &z2)`

Compares two complex values and returns a nonzero value if the two numbers are not equal; otherwise, it returns 0.

Other Functions

`double abs(const complex &z1)`

Returns the absolute value (magnitude) of a complex value.

`double arg(const complex &z1)`

Returns the angle, in radians, of a complex value. The result is normalized such that it is greater than or equal to 0, and less than 2 * π.

`double arg1(const complex &z1)`

Returns the principal value of the angle, in radians, of a complex value. The result is normalized such that it is greater than −π, and less than or equal to π.

`complex conj(const complex &z1)`

Returns the conjugate of a complex value; that is, if the number is `(real, imag)`, then the result is `(real, -imag)`.

`complex cos(const complex &z1)`

Returns the cosine of a complex value.

`complex cosh(const complex &z1)`

Returns the hyperbolic cosine of a complex value. The value of `real(z1)` must be small enough so that `exp(real(z1))` does not overflow; otherwise, the function creates a `c_exception` object and invokes the `complex_error` function.

`complex exp(const complex &z1)`

Returns the value of e (2.71828...) raised to the power of a complex value. The conditions described for `cosh()` must be met; otherwise, it creates a `c_exception` object and invokes the `complex_error` function.

`double imag(const complex &z1)`

Returns the imaginary part of a complex value.

`complex log(const complex &z1)`

Returns the natural logarithm (base e, 2.71828...) of a complex value. The conditions described for `cosh()` must be met; otherwise, it creates a `c_exception` object and invokes the `complex_error` function.

`double norm(const complex &z1)`

Returns the square of the absolute value (magnitude) of a complex value.

`complex polar(double rho, double theta = 0)`

Creates a complex value given a pair of polar coordinates (magnitude *rho* and angle *theta*, in radians).

`complex pow(const complex &z1, int i2)`

Returns the value of *z1* raised to the power of *i2*.

`complex pow(const complex &z1, double x2)`

Returns the value of *z1* raised to the power of *x2*.

complex pow(double z1, const complex &z2)

Returns the value of *z1* raised to the power of *z2*.

complex pow(const complex &z1, const complex &z2)

Returns the value of *z1* raised to the power of *z2*.

double real(const complex &z1)

Returns the real part of a complex value.

complex sin(const complex &z1)

Returns the sine of a complex value.

complex sinh(const complex &z1)

Returns the hyperbolic sine of a complex value. The conditions described for `cosh()` must be met; otherwise, it creates a `c_exception` object and invokes the `complex_error` function.

complex sqr(const complex &z1)

Returns the square of a complex value.

complex sqrt(const complex &z1)

Returns the square root of a complex value.

complex tan(const complex &z1)

Returns the tangent of a complex value.

complex tanh(const complex &z1)

Returns the hyperbolic tangent of a complex value. The conditions described for `cosh()` must be met; otherwise, it creates a `c_exception` object and invokes the `complex_error` function.

Examples

1. `complex zz(3,-5);`

   Declares `zz` to be a complex object and initializes it to the value of real part `3` and imaginary part `-5`.

2. `complex myarray[30];`

   Declares an array of 30 complex objects, all initialized to (0,0).

3. ```
   complex zz;
   while (!(cin >> zz).eof())
       cout << zz << endl;
   ```

   Reads a stream of complex values [for example, (`3.400000,5.000000`)] and writes them in the default format [for example, (`3.4, 5`)].

4. `complex cc = complex (3.4,5);`
   ```

```
cout << real(cc) << "+" << imag(cc) << "*i";
```

Prints out `3.4` as the real part of a complex object and `5` as the imaginary part. The result is `3.4+5*i`.

c_exception class

c_exception class — Contains information on a complex arithmetical exception.

Header

```
#include <complex.hxx>
```

Alternative Header

```
#include <complex.h>
```

Declaration

```
class c_exception
{
    friend complex   exp(const complex &);
    friend complex   sinh(const complex &);
    friend complex   cosh(const complex &);
    friend complex   log(const complex &);
    friend int       complex_error(c_exception &);

public:
    int              type;
    char             *name;
    complex          arg1;
    complex          arg2;
    complex          retval;

public:
    c_exception(char *, const complex &, const
    complex & = complex_zero);
};
```

Description

Objects of this class handle exceptions for complex arithmetic. This includes information on functions, parameters, error types, and default return values.

Data Members

`complex arg1`

Is the left argument of the function that incurred the error.

`complex arg2`

Is the right argument of the function that incurred the error.

`char *name`

Is the name of the function that incurred the error.

`complex retval`

Is the value to be returned by the function that incurred the error. You may use the `complex_error(c_exception &)` function to change this value.

`int type`

Is one of these kinds of error: `SING`, `OVERFLOW`, or `UNDERFLOW`.

Constructor

`c_exception(char *function_name, const complex &function_arg1, const complex &func`

Constructs a complex arithmetical exception object, with reference to the name and arguments of the function that incurred the error.

Other Function

`int complex_error (c_exception &error_information)`

Is the default error-handling function that is called by certain complex arithmetical functions in this package (namely, `cosh`, `exp`, `log`, and `sinh`) when those functions detect an arithmetical error. You may replace this function with your own function that takes an identical parameter list and returns a value as specified in the following table:

Return Value from Error-handling Function	Action Taken by Complex Arithmetical Function
0	Set the global value `errno`; if the error type is `SING`, print an error message.
non 0	Do not set `errno`; do not print an error message.

To substitute your own error-handling function, pass a pointer to your function to the `set_complex_error` function. (See the section called “Function”).

The complex arithmetical functions that invoke the error handling function always return the value specified in *error_information*`.retval`. Your error-handling function may set this value.

Chapter 3. generic Package

The generic package provides ways to simulate parameterized types by allowing the instantiation of class declarations using the macro facilities of the VSI C++ preprocessor. You can use the generic package to construct container classes. The actual types of the data members are passed at compile time as parameters to the class when you use the class name.

To declare a `generic` type:

1. Define a name for the class and specify the number of type parameters:

   ```
   #define YOUR_CLASS_NAME(TYPE_PARAMETER_NAME)
    name2(TYPE_PARAMETER_NAME, YOUR_CLASS_NAME)
   ```

   To specify two type parameters, use the `name3` macro.

2. Define the class body as a macro:

   ```
   #define YOUR_CLASS_NAMEdeclare(TYPE_PARAMETER_NAME) class {…};
   #define YOUR_CLASS_NAMEimplement(TYPE_PARAMETER_NAME) ...
   ```

3. Declare the actual class:`declare(YOUR_CLASS_NAME, ACTUAL_TYPE_NAME)`

   By substituting one or another class of *ACTUAL_TYPE_NAME*, you can declare multiple instances of the generic class template with various component types. For example, depending on the type parameter you use, you can declare such types as *list of* `int`*s*, *list of* `String`*s*, or *list of lists of* `String`*s*.

   If it is not a type name, *ACTUAL_TYPE_NAME* must be a `typedef` name.

   You must do this in each compilation unit that uses the parameterized type with a given parameter.

4. Define the functions or static data of the actual class. `implement(YOUR_CLASS_NAME, ACTUAL_TYPE_NAME)`

   You must do this once in each program that uses the parameterized type with a given parameter.

5. Declare an instance of the class you have declared by specifying objects of type *YOUR_CLASS_NAME(ACTUAL_TYPE_NAME)*, as follows:`YOUR_CLASS_NAME(ACTUAL_TYPE_NAME) object1, object2;`

Global Declarations

Global Declarations — These declarations are used by the generic package but they are not members of any class.

Header

```
#include <generic.hxx>
```

Alternative Header

```
#include <generic.h>
```

Compile-Time Parameters

TYPE, *TYPE1*, *TYPE2* – The types for which this class is parameterized; *TYPE*, *TYPE1*, or *TYPE2* must be an identifier.

CLASS – The class that is parameterized. For a vector of integers, for example, *CLASS* is `vector` and *TYPE* is `int`.

Declarations

```
typedef int  (*GPT)(int, char *);
int          genericerror(int n, char *msg);
```

Type

`GPT`

Is a pointer to a generic error-handling function.

Function

`int genericerror (int n, char *msg)`

Is the default error-handling function; it prints an error number (*n*) and message (*msg*) on `cerr` and calls `abort()`.

Macros

Macros provide preprocessor facilities for simulating parameterized types. The following macros are defined for the generic package:

`callerror(CLASS, TYPE, N, S)`

Calls the current error handler for a given instance of a parameterized class. *CLASS* denotes the name of the generic class (for example, `vector`). *TYPE* denotes the type parameter for which to instantiate the generic class (for example, `int` to get a vector of integers); the type must be an identifier (for example, `char*` is not valid). *N* denotes the first argument to pass to the error handler; the default is the function `genericerror(int, char*)`. *S* denotes the second argument to pass to the error handler.

`declare(CLASS, TYPE)`

Declares the class specified by a macro with the name of the generic class. The word `declare` follows the class name (for example, `vectordeclare`). It also defines the inline member functions of the class. *CLASS* denotes the name of the generic class (for example, `vector`). *TYPE* denotes the type parameter for which to instantiate the generic class (for example, `int` to get a vector of integers). The type must be an identifier (for example`char*`, is not valid).

`declare2(CLASS, TYPE1,TYPE2)`

Declares the class specified by a macro with the name of the generic class. The name is followed by the word `declare2`. The `declare2` macro differs from the `declare` macro only in that you use it to declare two type parameters, *TYPE1* and *TYPE2*.

`errorhandler(CLASS, TYPE)`

Is the name of the pointer to the error handler for a given instance of a parameterized class (for example, `intvectorhandler` to handle errors for a vector of integers). *CLASS* denotes the name of the generic class (for example, `vector`). *TYPE* denotes the type parameter for which to instantiate the generic class (for example, `int` to get a vector of integers). The type must be an identifier (for example, `char*` is not valid).

`implement(CLASS, TYPE)`

Defines the noninline member functions of a class, specified by a macro with the name of the generic class. The name is followed by the word `implement` (for example, `vectorimplement`). The `implement` macro takes the same arguments as the `declare` macro.

`implement2(CLASS, TYPE1,TYPE2)`

Defines the noninline member functions of a class, specified by a macro with the name of the generic class. The name is followed by the word `implement2`. The `implement2` macro differs from the `implement` macro only in that you use it to declare two type parameters, *TYPE1* and *TYPE2*.

`name2(S1,S2)`

Concatenates two identifier segments to form a new identifier using the ## operator.

`name3(S1,S2,S3)`

Concatenates three identifier segments to form a new identifier using the ## operator.

`name4(S1,S2,S3,S4)`

Concatenates four identifier segments to form a new identifier using the ## operator.

`set_handler(CLASS, TYPE, HANDLER)`

Specifies a function as the current error handler for a given instance of a parameterized class. Initially, the error-handling function is set to `genericerror(int, char*)`. *CLASS* denotes the name of the generic class (for example, `vector`). *TYPE* denotes the type parameter for which to instantiate the generic class (for example, `int` to get a vector of integers); the type must be an identifier (for example, `char*` is not valid). *HANDLER* denotes a pointer to the function you want to set to the new error handler. Also, you can use the `set_handler` macro in a function declaration or definition.

Example

The following program shows the use of the `genericerror` function and associated macros:

```
extern "C"
{
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
}

#include <generic.hxx>

#define my_vector(T) name2(T, my_vector)

// Declare a vector of objects of type T (the class and extern data)
#define my_vectordeclare(T) \
    class my_vector(T) \
    { \
```

```
    private: \
        int s; \
        T *p; \
    public: \
        my_vector(T)(int); \
        ~my_vector(T)(); \
        T &operator[](int); \
    }; \
    extern GPT errorhandler(my_vector, T); \
    extern GPT set_handler(my_vector, T, GPT);

// Implement a vector of objects of type T
// (Define the functions and global data)
#define my_vectorimplement(T) \
    my_vector(T)::my_vector(T)(int size) \
    { \
        s = size; \
        p = new T[size]; \
    } \
    my_vector(T)::~my_vector(T)() \
    { \
        delete[] p; \
    } \
    T &my_vector(T)::operator[](int i) \
    { \
        if(i < 0 || i >= s) \
        { \
            callerror(my_vector, T, i, "Index out of bounds"); \
            static T error_object; \
            return error_object; \
        } \
        return p[i]; \
    } \
    GPT errorhandler(my_vector, T) = &genericerror; \
    GPT set_handler(my_vector, T, GPT new_genericerror) \
    { \
        GPT old_genericerror = errorhandler(my_vector, T); \
        errorhandler(my_vector, T) = new_genericerror; \
        return old_genericerror; \
    }

// Declare and implement vector of int
declare(my_vector, int)
implement(my_vector, int)

// Error-handling function
my_handler(
    int n,
    char *msg
    )

{
    fflush(stderr);
    printf("in my_handler(%d,\"%s\")\n", n, msg);
    fflush(stdout);
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    my_vector(int) v1(10);

    GPT old_error_handler;

    // Set the handler to a function that does not abort
    old_error_handler = set_handler(my_vector, int, &my_handler);
    v1[12345] = 0;

    // Restore the handler and cause an error
    // This should abort
    old_error_handler = set_handler(my_vector, int, old_error_handler);
    v1[12345] = 0;

    return EXIT_SUCCESS;
}
```

See Also

Chapter 11

Chapter 4. iostream Package

Classes in the iostream package provide methods to handle input and output streams, including reading and writing built-in data types. You also can extend certain methods described here to handle class types.

This package includes, among others, the classes `ios` and `streambuf`, and the subclasses derived from these base classes. Figure 4.1 shows the inheritance structure of the iostream package. In the diagram, arrows point from the base classes to derived classes.

The `istream` (input stream) class supports input operations (extractions); the `ostream` (output stream) class supports output operations (insertions). The `iostream` class derives from both `istream` and `ostream`, and supports both extractions and insertions.

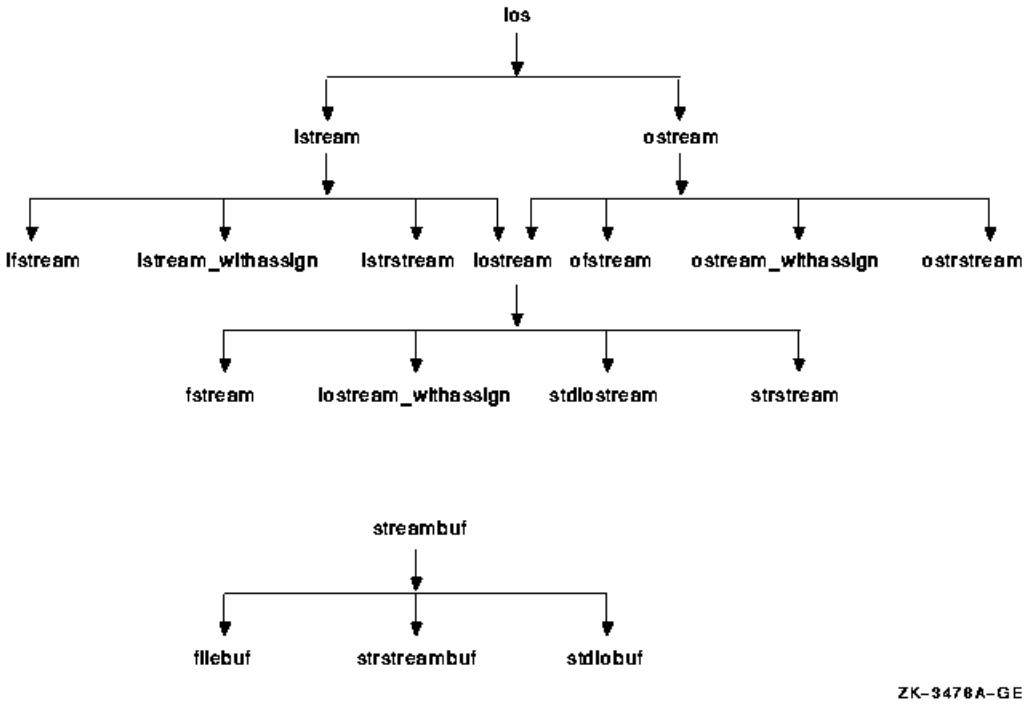
The following stream objects are predefined:

`cin`	An `istream_withassign` object linked to standard input
`cout`	An `ostream_withassign` object linked to standard output
`cerr`	An `ostream_withassign` object linked to standard error that supports unbuffered output
`clog`	An `ostream_withassign` object linked to standard error that supports buffered output

To generate output, you apply the insertion operator (<<) to `cout`, as shown in the following example:

```
cout << "Hello\n" ;
```

Figure 4.1. Inheritance Diagram for the iostream Package



Obtaining input is similar to generating output, except that you apply the extraction operator (>>) to `cin`, as shown in the following example:

```
int eye, jay ;
cin >> eye >> jay ;
```

If you include these fragments of code in a program, your system expects users to type in two integer values (for `eye` and `jay`) from a terminal. The iostream package supplies predefined extraction and insertion operators for all built-in data types, including `char*`.

This package also supports file manipulation. To connect a specific file to your program, instantiate one of the following class types:

`ifstream` (for file input)
`ofstream` (for file output)
`fstream` (for both input and output)

To format within character arrays, the iostream package includes the following associated class types:

`istrstream` (for fetching characters from an array)
`ostrstream` (for storing characters into an array)
`strstream` (for both fetching and storing characters into an array)

Note

On systems with IEEE floating-point arithmetic, certain values may be printed as symbols for Infinity (for example, `INF`) or Not a Number (for example, `NaN`).

Deriving Your Own Class from ios

If you derive your own class from the `ios` class, or from one of its derived classes, the `ios` subobject must be initialized properly during instantiation. Specifically, you must ensure that the `streambuf` pointer within the `ios` subobject is valid.

To do this, you can specify the `ios(streambuf *)` constructor as a member initializer for your class constructor. Optionally, you can call the `ios::init(streambuf *)` member function.

Thread Safety

The predefined stream objects, `cerr`, `cin`, `clog`, and `cout` are thread safe only for individual calls into the VSI C++ Class Library. You must provide synchronization around sequences of calls. For more information on synchronizing access to predefined stream objects, see the section on Global Declarations in this chapter.

User-defined stream objects are not thread safe, so you must provide synchronization around individual calls as well as sequences of calls. For more information on synchronizing access to user-defined objects, see Chapter 6 and the section on Global Declarations in this chapter.

The `ios` member function `sync_with_stdio()` is not thread safe. If your application calls this function, it must make the call before any threads use `cerr`, `cin`, `clog`, or `cout`.

Global Declarations

Global Declarations — These declarations are used by the iostream package but they are not members of any class.

Header

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declarations

```
typedef long    streamoff
typedef long    streampos

ios             &dec(ios &s);
ios             &hex(ios &s);
ios             &oct(ios &s);
ios             &lock(ios &s);
ios             &unlock(ios &s);

istream         &ws(istream &i);
ostream         &endl(ostream &o);
ostream         &ends(ostream &o);
ostream         &flush(ostream &o);
```

Types

`typedef long streamoff`

Is the type representing a character offset into a stream. For more information, see the description of the `seekoff` and `seekpos` functions in the `streambuf` class.

`typedef long streampos`

Is the type representing a character position in a stream. For more information, see the description of the `seekoff` and `seekpos` functions in the `streambuf` class.

Manipulators

The following functions insert values into a stream, extract values from a stream, or specify the conversion base format. For more information on the conversion base format flags, see the iostream class.

`ios &dec(ios &s)`

Sets the conversion base format for *s* to decimal, essentially clearing the `ios::oct` and `ios::hex` flags and setting the `ios::dec` flag.

`ios &hex(ios &s)`

Sets the conversion base format for *s* to hexadecimal, essentially clearing the `ios::oct` and `ios::dec` flags and setting the `ios::hex` flag.

`ios &oct(ios &s)`

Sets the conversion base format for *s* to octal, essentially clearing the `ios::dec` and `ios::hex` flags and setting the `ios::oct` flag.

`istream &ws(istream &i)`

Extracts (skips) white-space characters from *i*.

`ostream &endl(ostream &o)`

Ends a line by inserting a new-line character into *o* and flushing *o*.

`ostream &ends(ostream &o)`

Ends a string by inserting a null `'/0'` character into *o*.

`ostream &flush(ostream &o)`

Flushes *o*.

Synchronizing Access to Predefined Stream Objects

The following unparameterized manipulators are for use in synchronizing access to the predefined stream objects, `cerr`, `cin`, `clog`, and `cout`:

`ios &lock(ios &s)`

Locks *s* if *s* is one of the predefined stream objects.

`ios &unlock(ios &s)`

Unlocks *s* if *s* is one of the predefined stream objects.

If your application needs to lock two or more of these objects at the same time, your application must adhere to the following locking order:

1. `cin`
2. `cerr`
3. `clog`
4. `cout`

For example, if your application needs to lock both `cerr` and `cout`, lock `cerr` first and `cout` second. The unlocking order is not important.

Keep in mind that when your application calls a member function for a predefined stream object, the member function will typically lock the object for the duration of the call. Therefore, if your application has locked one of the stream objects and then uses another, this use must also adhere to the predefined locking order. For example, your application should not send output to `cerr` while `cout` is locked.

The locking order necessarily matches the default **ties** between the stream objects as follows:

`cin` is tied to `cout`
`cerr` is tied to `cout`
`clog` is tied to `cout`
`cout` has no ties

Any input/output operation on a stream object causes the iostream package to flush the object to which it is tied. Thus, an output to `cerr` flushes `cout`.

Examples

1. ```
   #include <iostream.hxx>
   #include <iomanip.hxx>

   int main ()
   {
      int value = 10;

      cout << hex << value << ',';   // Change the base conversion format
                                     // to hexadecimal; note that the
                                     // default is decimal as set by
                                     //the ios constructors.

      cout << value << ',';          // The base conversion format set in
                                     // the previous line is still active.

      cout << dec << value << endl;  // Change the base conversion format
                                     // to decimal; lastly, insert a
                                     // new-line character into the
                                     // stream and flush cout.
      return 0;
   }
   ```

   The output is a,a,10.

2. ```
   #include <string.hxx>
   #include <iostream.hxx>

   void print_name(String &name)
   {
      cout << lock << "Hello, " << name << endl << unlock;
   }
   ```

   This synchronizes access to the `cout` object so that the `"Hello, "`, name, and new-line character are written to `cout` as a single unit. If you do not use the `lock` and `unlock` manipulators in this example, another thread could possibly insert its own text into `cout` in the midst of your output.

Header

```
include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Declarations

```
SMANIP(long)        resetiosflags(long);
SMANIP(long)        setiosflags(long);
SMANIP(int)         setfill(int);
SMANIP(int)         setprecision(int);
SMANIP(int)         setw(int w);

SMANIPREF(Mutex)    lock(Mutex &m)
SMANIPREF(Mutex)    unlock(Mutex &m)
```

Functions

These functions are used for extending the iostream package with user-defined parameterized manipulators.

`SMANIP(long) resetiosflags(long x)`

In the stream (`ios` or a stream derived from `ios`), clears the format flags denoted by *x*.

`SMANIP(int) setfill(int x)`

Sets the fill character to be the value specified by *x*. The fill character is a data member of the `ios` class; however, setting it with this function affects only output streams.

`SMANIP(long) setiosflags(long x)`

In the stream (`ios` or a stream derived from `ios`), turns on the format flags denoted by *x*. If you are setting a flag that is part of a collection (for example, `basefield`), note that this manipulator does *not* clear the other flags in the collection.

`SMANIP(int) setprecision(int x)`

Sets the variable that controls the number of digits inserted by the floating-point inserter to be *x*. This variable is a data member of the `ios` class; however, setting it with this function affects only output streams.

`SMANIP(int) setw(int w)`

In the stream (`ios` or a stream derived from `ios`), sets the field width of the stream to *w*.

Synchronizing Access to User-Defined Stream Objects

The following parameterized manipulators are for use in synchronizing access to user-defined stream objects. To use these manipulators, you must first define a `Mutex` object, which you then pass to the manipulator. The association of a `Mutex` object with a stream object is not enforced by the iostream package. This association is enforced only by you, the programmer. Refer to Chapter 6 for information on the `Mutex` class.

`SMANIPREF(Mutex) lock(Mutex &m)`

Locks the recursive Mutex represented by *m*.

`SMANIPREF(Mutex) unlock(Mutex &m)`

Unlocks the recursive Mutex represented by *m*.

Examples

1. ```
   char c;
   cin >> resetiosflags(ios::skipws)
       >> c
       >> setiosflags(ios::skipws);
   ```

   Turns off the flag (resets it to 0) that tells the extractor (>>) to skip leading white space and then turns that flag back on again (sets it to 1).
   ```

2. ```
   cout.fill(*)
   cout.setf(ios::left,ios::adjustfield);
   cout << setw(6) << 23 << "," ;
   cout.fill(%);
   cout.setf(ios::right,ios::adjustfield);
   cout << setw(4) << 34 << "\n" ;
   ```

   Places padding characters (specified by the fill state variable) after the first number and before the second number. The output is `23****,%%34`.

3. ```
   #include <string.hxx>
   #include <fstream.hxx>
   #include <mutex.hxx>
   #include <iomanip.hxx>

   main ()
   {
      String name("Henry");
      void print_name (String &, ostream &, Mutex &);

      ofstream mystream(1);
      Mutex mystream_lock;

      print_name(name, mystream, mystream_lock);
      return 0;
   }

      void print_name(String &name, ostream &stream, Mutex &stream_lock)
   {
         stream << lock(stream_lock) << "Hello, " << name << endl
                << unlock(stream_lock);
   }
   ```

   This example associates a `Mutex` object with a stream object to synchronize access to the stream. The Mutex is locked before using the stream and then unlocked afterwards. For the synchronization to work properly, each thread that uses this stream must perform the same lock/unlock sequence with the same Mutex.

See Also

IMANIP(TYPE) class

IOMANIP(TYPE) class

OMANIP(TYPE) class

SMANIP(TYPE) class

filebuf class

filebuf class — Provides a data buffer abstraction for input/output facilities through file descriptors.

Header

```
#include <fstream.hxx>
```

Alternative Header

```
#include <fstream.h>
```

Declaration

```
class filebuf: public streambuf
{
public:
    static const int    openprot;

                        filebuf();
                        filebuf(int fd);
                        filebuf(int fd, char *p, int len);
                        ~filebuf();

    filebuf             *attach(int fd);
    filebuf             *close();
    int                 fd();
    int                 is_open();
    filebuf             *open(const char *name, int mode,
                        int prot = openprot);

    virtual int         overflow(int = EOF);
    virtual streampos   seekoff(streamoff, seek_dir, int mode);
    virtual streampos   seekpos(streampos, int mode);
    virtual streambuf   *setbuf(char *p, int len);
    virtual int         sync();
    virtual int         underflow();
};
```

Description

This class specializes the `streambuf` class to use a file as a repository of characters. Writing to the file consumes characters; reading from the file produces characters. Files that allow searches are said to be seekable. When a file is readable and writable, the `filebuf` object permits character insertion and extraction.

If your program expects a buffer to be allocated when none was allocated, then the iostream package allocates a default buffer with a length specified by `BUFSIZ` as defined in `stdio.h`. The package then issues the following warning:

```
Warning; a null pointer to streambuf was passed to ios::init()
```

Data Member

`const int openprot = 0644`

Provides default protection for the `open()` function.

Constructors and Destructors

`filebuf()`

Constructs a `filebuf` object that is initially closed.

filebuf(int fd)

Constructs a `filebuf` object connected to file descriptor *fd*.

filebuf(int fd, char *p, int len)

Constructs a `filebuf` object connected to file descriptor *fd*, which is initialized to use the reserve area (buffer) starting at *p* and containing *len* bytes.

~filebuf()

Deletes a `filebuf` object.

Member Functions

filebuf *attach(int fd)

Connects the `filebuf` object to an open file whose descriptor is passed through the *fd* argument. It normally returns a reference to the `filebuf` object, but returns 0 if the `filebuf` object is connected to an open file.

filebuf *close()

Flushes any waiting output, closes the file descriptor, and disconnects a `filebuf` object. Unless an error occurs, the `filebuf` object's error state will be cleared. The `close()` function returns the address of the `filebuf` object unless errors occur, in which case this function returns 0. Even if errors occur, `close()` leaves the file descriptor and `filebuf` object closed.

int fd()

Returns the file descriptor associated with a `filebuf` object. If the `filebuf` object is closed, `fd` returns `EOF`.

int is_open()

Returns a nonzero value when a `filebuf` object is connected to a file descriptor; otherwise, it returns 0.

filebuf *open(const char *name, int mode, int prot)

Opens a file with the name specified by *name* and connects a `filebuf` object to it. If the file does not exist, the function tries to create it with the protection mode *prot* unless `ios::nocreate` is specified in *mode*. By default, *prot* is `filebuf::openprot`.

The function fails if the `filebuf` object is open. The `open()` function normally returns the address of the `filebuf` object, but returns 0 if an error occurs. The members of `open_mode` are bits that may be joined together by `or` (because this joining takes an `int`, `open()` takes an `int` rather than an `open_mode` argument). For an explanation of the meanings of these bits in `open_mode`, see the Enumerated Types section for the `ios` class.

virtual int overflow(int c)

Called to consume characters in classes derived from `streambuf`. If *c* is not `EOF`, this function must also either save *c* or consume it. Although it can be called at other times, this function usually is called when the put area is full and an attempt is being made to store a new character. The normal action is

to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new put area, and (if *c* `!= EOF`) store *c* using `sputc()`. A call to `overflow(c)` should return `EOF` to indicate an error; otherwise, it should return something else.

`virtual streampos seekoff(streamoff off, seek_dir dir, int mode)`

Moves the get pointer, put pointer, or both as designated by the *off* and *dir* arguments. It may fail if the file does not support seeking, or if the attempted motion is otherwise invalid (for example, attempting to seek a position before the beginning of the file). The *off* argument is interpreted as a count relative to the place in the file specified by *dir*. The *mode* argument is ignored. A call to `seekoff()` returns the new position or `EOF` if a failure occurs. After a failure, the position of the file is undefined.

`virtual streampos seekpos(streampos pos, int mode)`

Moves the file to a position *pos*. The *mode* argument is ignored. The function normally returns *pos* but it returns `EOF` on failure.

`virtual streambuf *setbuf(char *p, int len)`

Sets up the reserve area as the number of bytes specified in the second argument, beginning at the pointer specified in the first argument. If the pointer is null, or the number of bytes is less than 1, the `filebuf` object is unbuffered. This function normally returns a pointer to the `filebuf` object; however, if the `filebuf` object is open and a buffer is allocated, then no changes are made to the reserve area and to the buffering status, and `setbuf()` returns 0.

`virtual int sync()`

Tries to get the state of the get pointer, the put pointer, or both, to agree (synchronize) with the state of the file to which the `filebuf` object is connected. This means that the function may write characters to the file if some of the characters have been buffered for output, or the function may try to reposition (seek) the file if characters have been read and buffered for input. Normally `sync()` returns 0, but it returns `EOF` if synchronization is not possible.

When certain characters must be written together, the program should use `setbuf()` (or a constructor) to ensure that the reserve area is at least as large as the number of characters to be written together. Your program can then call `sync()`, store the characters, and then call `sync()` once again.

`virtual int underflow()`

Called in classes derived from `streambuf` to supply characters for fetching; that is, to create a condition in which the get area is not empty. If the function is called when characters occupy the get area, it should create a nonempty area and return the next character (which it should also leave in the get area). If no more characters are available, `underflow()` should return `EOF` and leave an empty get area.

See Also

ios class

streambuf class

fstream class

fstream class — Supports formatted and unformatted input from and output to files.

Header File

```
#include <fstream.hxx>
```

Alternative Header

```
#include <fstream.h>
```

Declaration

```
class fstream: public iostream
{
public:
                 fstream();
                 fstream(const char *name, int mode,
                     int prot = filebuf::openprot);
                 fstream(int fd);
                 fstream(int fd, char *p, int len);
                 ~fstream();

    void         attach(int fd);
    void         close();
    void         open(const char *name, int mode,
                     int prot = filebuf::openprot) ;
    filebuf      *rdbuf();
    void         setbuf(char *p, int len);
};
```

Description

This class specializes the `iostream` class to files by using a `filebuf` object to do the input and output. Your program can perform common operations, such as opening and closing files, without explicitly mentioning `filebuf` objects.

Constructors and Destructors

`fstream()`

Constructs an unopened `fstream` object.

`fstream(int fd)`

Constructs an `fstream` object connected to the file whose descriptor is passed through the *fd* argument. The file must be open.

`fstream(int fd, char *p, int len)`

Constructs an `fstream` object connected to a file whose descriptor is passed through the *fd* argument, and also initializes the associated `filebuf` object to use the *len* bytes starting at *p* as the reserve area. If *p* is null or *len* is 0, the `filebuf` object is unbuffered.

`fstream(const char *name, int mode, int prot)`

Constructs an `fstream` object and opens the file specified by the *name* argument. The *mode* and *prot* arguments specify the file open mode and protection. By default, *prot* is `filebuf::openprot`. If the open action fails, the error state (`io_state`) of the constructed `fstream` object indicates failure.

`~fstream()`

Deletes an `fstream` object.

Member Functions

void attach(int fd)

Connects an `fstream` object to a file whose descriptor is passed through the *fd* argument. A failure occurs when the `fstream` object is connected to a file, in which case `ios::failbit` is set in the `filebuf` object's error state.

void close()

Closes any associated `filebuf` object and consequently breaks the connection of the `fstream` object to the file. The error state of the `fstream` object is cleared except on failure. A failure occurs when the call to the `filebuf` object's `close()` function fails.

void open(const char *name, int mode, int prot)

Opens a file with the file name specified by *name* and connects the `fstream` object to it. If the file does not exist, the function tries to create it with the protection specified by the *prot* argument unless `ios::nocreate` is set. By default, *prot* is `filebuf::openprot`.

Failure occurs if the `fstream` object is open or when the call to the `filebuf` object's `open()` function fails, in which case `ios::failbit` is set in the `filebuf` object error state. The members of `open_mode` are bits that may be joined together by `or` (because this joining takes an `int`, `open()` takes an `int` rather than an `open_mode` argument). For an explanation of the meanings of these bits in `open_mode`, see the Enumerated Types section for the `ios` class.

filebuf *rdbuf()

Returns a pointer to the `filebuf` object associated with the `fstream` object. This function has the same meaning as `ios::rdbuf()`, but has a different type.

void setbuf(char *p, int len)

Calls the associated `filebuf` object `setbuf()` function to request space for a reserve area. A failure occurs if the `filebuf` object is open or if the call to `rdbuf()->setbuf` fails for any other reason.

IAPP(TYPE) class

IAPP(TYPE) class — For an `istream` object, declares predefined parameterized applicators.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `istream` object. It must be an identifier.

Description

```
class IAPP(TYPE)
{
public:
    IAPP(TYPE)(istream &(*f)(istream &, TYPE));
    IMANIP(TYPE) operator()(TYPE a);
};
```

Constructor

`IAPP(TYPE) (istream &(*f) (istream &, TYPE))`

Creates an applicator; **f* is the left operand of the insertion operator.

Operator

`IMANIP(TYPE) operator () (TYPE a)`

Casts an object of type *a* into a manipulator function for an `istream` object.

See Also

IMANIP(TYPE) class

ifstream class

ifstream class — Supports formatted and unformatted input from files.

Header File

```
#include <fstream.hxx>
```

Alternative Header

```
#include <fstream.h>
```

Declaration

```
class ifstream: public istream
{
public:
                ifstream();
                ifstream(const char *name, int mode = ios::in,
                     int prot = filebuf::openprot);
                ifstream(int fd);
                ifstream(int fd, char *p, int len);
                ~ifstream();

    void        attach(int fd);
    void        close();
```

```
    void        open(const char *name, int mode = ios::in,
                     int prot = filebuf::openprot);
    filebuf     *rdbuf();
    void        setbuf(char *p, int len);
};
```

Description

This class specializes the `istream` class to files by using a `filebuf` object to do the input. Your program can perform common operations, such as opening and closing files, without explicitly mentioning `filebuf` objects.

Constructors and Destructors

`ifstream()`

Constructs an unopened `ifstream` object.

`ifstream(int fd)`

Constructs an `ifstream` object connected to a file whose descriptor is passed through the *fd* argument. The file must already be open.

`ifstream(int fd, char *p, int len)`

Constructs an `ifstream` object connected to a file whose descriptor is passed through the *fd* argument, and also initializes the associated `filebuf` object to use the *len* bytes starting at *p* as the reserve area. If *p* is null or *len* is 0, the `filebuf` object is unbuffered.

`ifstream(const char *name, int mode, int prot)`

Constructs an `ifstream` object and opens the file with the file name specified by *name*. The *mode* and *prot* arguments specify the file open mode and protection. By default, *prot* is `filebuf::openprot`. If the open fails, the error state (`io_state`) of the constructed `ifstream` object indicates failure.

`~ifstream()`

Deletes an `ifstream` object.

Member Functions

`void attach(int fd)`

Connects an `ifstream` object to a file whose descriptor is passed through the *fd* argument. A failure occurs when the `ifstream` object is connected to a file, in which case `ios::failbit` is set in the `ifstream` object error state.

`void close()`

Closes any associated `filebuf` object and consequently breaks the connection of the `ifstream` object to the file. The error state of the `fstream` object is cleared except on failure. A failure occurs when the call to the `filebuf` object's `close()` function fails.

`void open(const char *name, int mode, int prot)`

Opens a file specified by the *name* argument and connects the `ifstream` object to it. If the file does not exist, the function tries to create it with the protection specified by the *prot* argument unless `ios::nocreate` is set. By default, *prot* is `filebuf::openprot`.

Failure occurs if the `ifstream` object is open or when the call to the `filebuf` object `open()` function fails, in which case `ios::failbit` is set in the `filebuf` object error state. The members of `open_mode` are bits that may be joined together by `or` (because this joining takes an `int`, `open()` takes an `int` rather than an `open_mode` argument). For an explanation of the meanings of these bits in `open_mode`, see the Enumerated Types section for the `ios` class.

`filebuf *rdbuf()`

Returns a pointer to the `filebuf` object associated with the `ifstream` object. This function has the same meaning as `ios::rdbuf()` but has a different type.

`void setbuf(char *p, int len)`

Calls the associated `filebuf` object `setbuf()` function to request space for a reserve area. A failure occurs if the `filebuf` object is open or if the call to `rdbuf()->setbuf` fails for any other reason.

IMANIP(TYPE) class

IMANIP(TYPE) class — For an `istream` object, declares the predefined parameterized manipulators and provides macros for user-defined parameterized manipulators.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `istream` object. It must be an identifier.

Declaration

```
class IMANIP(TYPE)
{
public:
    IMANIP(TYPE)(istream &(*f)(istream &, TYPE), TYPE a);
    friend istream &operator>>(istream &s, IMANIP(TYPE) &m);
};
```

Description

These manipulators serve the `istream` class by producing some useful effect, such as embedding a function call in an expression containing a series of insertions and extractions. You also can use manipulators to shorten the long names and sequences of operations required by the `iostream` class.

In its simplest form, a manipulator takes an `istream&` argument, operates on it in some way, and returns it.

Constructor

`IMANIP(TYPE)(istream &(*f)(istream &, TYPE), TYPE a)`

Creates a manipulator; **f* is the left operand of the extractor operator.

Operator

`istream &operator >> (istream &s, IMANIP(TYPE) &m)`

Takes data from an `istream` object.

IOAPP(TYPE) class

IOAPP(TYPE) class — For an `iostream` object, declares predefined parameterized applicators.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `iostream` object. It must be an identifier.

Declaration

```
class IOAPP(TYPE)
{
public:
    IOAPP(TYPE)(iostream &(*f)(iostream &, TYPE));
    IOMANIP(TYPE) operator()(TYPE a);
};
```

Constructor

`IOAPP(TYPE)(iostream &(*f)(iostream &, TYPE))`

Creates an applicator.

Operator

`IOMANIP(TYPE) operator () (TYPE a)`

Casts an object of type *a* into a manipulator function for an `iostream` object.

See Also

IOMANIP(TYPE) class

IOMANIP(TYPE) class

IOMANIP(TYPE) class — For an `iostream` object, declares predefined parameterized manipulators and provides macros for user-defined parameterized manipulators.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `iostream` object. It must be an identifier.

Declaration

```
class IOMANIP(TYPE)
{
public:
    IOMANIP(TYPE)(iostream &(*f)(iostream &, TYPE), TYPE a);
    friend istream &operator>>(iostream &s, IOMANIP(TYPE) &m);
    friend ostream &operator<<(iostream &s, IOMANIP(TYPE) &m);
};

IOMANIPdeclare(int);
IOMANIPdeclare(long);
```

Description

These manipulators serve the `iostream` class by producing some useful effect, such as embedding a function call in an expression containing a series of insertions and extractions. You can also use manipulators to shorten the long names and sequences of operations required by the `iostream` class.

In its simplest form, a manipulator takes an `iostream&` argument, operates on it in some way, and returns it.

Two `ios` manipulators for using `Mutex` objects, `lock` and `unlock`, come in both parameterized and unparameterized forms. The parameterized manipulators let users synchronize `iostream` objects, the parameter being a user-defined `Mutex` object. To use parameterized manipulators, you must include `iomanip.hxx`. Unparameterized manipulators let users synchronize the predefined stream objects: `cerr`, `cin`, `clog`, and `cout`.

For examples of using the `lock` and `unlock` manipulators, see Chapter 6 and the section on Global Declarations in this chapter.

Constructor

`IOMANIP(TYPE)(iostream &(*f)(iostream &, TYPE), TYPE a)`

Creates a manipulator.

Macro

`IOMANIPdeclare(TYPE)`

Declares the manipulators (and the manipulator classes) that have an `operator()` member function for type *TYPE*.

Operators

`ostream &operator << (iostream &s, IOMANIP(TYPE) &m)`

Sends data to an `iostream` object.

`istream &operator >> (iostream &s, IOMANIP(TYPE) &m)`

Takes data from an `iostream` object.

ios class

ios class — Contains state variables common to most of the other classes in the iostream package.

Header

`#include <iostream.hxx>`

Alternative Header

`#include <iostream.h>`

Declaration

```
class ios
{
public:
    enum io_state       { goodbit = 0, eofbit = 01,
                          failbit = 02, badbit = 04 };
    enum open_mode      { in = 01, out = 02, ate = 04,
                          app = 010, trunc = 020,
                          nocreate = 040, noreplace = 0100 };
    enum seek_dir       { beg = 0, cur = 01, end = 02 };

    enum                { skipws = 01,
                          left = 02, right = 04, internal = 010,
                          dec = 020, oct = 040, hex = 0100,
                          showbase = 0200, showpoint = 0400,
                          uppercase = 01000,
                          showpos = 02000,
                          scientific = 04000, fixed = 010000,
                          unitbuf = 020000, stdio = 040000 };

    static const long   basefield;
    static const long   adjustfield;
    static const long   floatfield;
```

```
                        ios(streambuf *);
    virtual             ~ios();

    inline int          bad() const;
    static long         bitalloc();
    inline void         clear(int state = 0);
    inline int          eof() const;
    inline int          fail() const;
    inline char         fill() const;
    char                fill(char);
    inline long         flags() const;
    long                flags(long);
    inline int          good() const;
    long                &iword(int);
    inline int          operator!();
    inline              operator void *();
    inline int          precision() const;
    int                 precision(int);

    void                *&pword(int);
    inline streambuf    *rdbuf();
    inline int          rdstate() const;
    long                setf(long setbits, long field);
    long                setf(long);
    static void         sync_with_stdio();
    inline ostream      *tie() const;
    ostream             *tie(ostream *);
    long                unsetf(long);
    inline int          width() const;
    int                 width(int n);
    static int          xalloc();

protected:
                        ios();
    void                init(streambuf *);
    inline void         setstate(int state);

};
```

Description

Classes derived from the `ios` class provide an interface for transferring formatted and unformatted information into and out of `streambuf` objects.

Enumerated Types

`io_state`

Represents a collection of bits (flags) that describe the internal error states of an object. The values are as follows:

`goodbit`	No errors occurred.
`eofbit`	End-of-file encountered during an extraction operation.
`failbit`	Extraction or conversion failed but the stream is still usable.

`badbit`	A severe error, usually in an operation on the associated `streambuf` object, from which recovery is unlikely.

`open_mode`

Represents a collection of bits (flags) for specifying the mode of the `open()` function. Use this data type with objects of the `fstream`, `ifstream`, and `ofstream` classes. The values are as follows:

`app`	Performs a seek to the end-of-file. This appends to the end of the file any subsequent data written to the file. `ios::app` implies `ios::out`.
`ate`	Performs a seek to the end-of-file during an `open()` operation. `ios::ate` does not imply `ios::out`.
`in`	Opens the file for input. Constructions and open operations of `ifstream` objects imply `ios::in`. For `fstream` objects, `ios::in` signifies that input operations should be allowed if possible. Including `ios::in` in the modes of an `ofstream` object is legal, implying that the original file (if it exists) should not be truncated.
`out`	Opens the file for output. Constructions and open operations of `ofstream` objects imply `ios::out`. For `fstream` objects, `ios::out` indicates that output operations are allowed.
`trunc`	Truncates (discards) the contents of the file (if it exists). `ios::trunc` is implied if `ios::out` is specified (including implicit specification for `ofstream` objects), and neither `ios::app` nor `ios::ate` is specified.
`nocreate`	Causes an `open()` operation to fail if the file does not exist.
`noreplace`	Causes an `open()` operation to fail if the file exists.

`seek_dir`

Represents a collection of bits for positioning get and put pointers. Use this data type with functions of the `filebuf`, `istream`, `ostream`, and `streambuf` classes. The values are as follows:

`beg`	Indicates the beginning of the stream
`cur`	Indicates the current position
`end`	Indicates the end of the stream (end-of-file)

Data Members

`const long adjustfield`

Collectively specifies the flags (bits) that control padding (`left`, `right`, and `internal`).

`const long basefield`

Collectively specifies the flags that control base conversion (`dec`, `hex`, and `oct`).

`const long floatfield`

Collectively specifies the flags that control floating-point value conversion (`fixed` and `scientific`).

Note

When you set a flag that is part of `adjustfield`, `basefield`, or `floatfield`, you must ensure that the other flags within the collection are cleared. Only one flag within the collection should be set at any one time.

Be aware that the `setiosflags(flag)` manipulator and the `setf(flag)` member function set only the flag or flags that you specify. If the flag you specify is part of a collection, these do not clear the other flags in the collection.

The `setf(flag, field)` member function is useful for setting fields within a collection. Also, the `hex`, `oct`, and `dec` manipulators do ensure that the other flags within the `basefield` collection are cleared.

Constructors and Destructors

`ios()`

Constructs an `ios` object with the effect undefined. It lets derived classes inherit the `ios` class as a virtual base class. The object is initialized with the following default values:

Element	Default Value
`fill()`	The space character
`flags()`	`ios::dec|ios::skipws`
`precision()`	6
`rdstate()`	`ios::goodbit`
`width()`	0

`ios(streambuf *b)`

Constructs an `ios` object, associating the constructed `ios` object with the `streambuf` object pointed to by *b*. The object is initialized with the same default values as the `ios()` constructor.

`virtual ~ios()`

Deletes an `ios` object.

Overloaded Operators

When defined, the following operators allow convenient checking of the error state of an `ios`.

`int operator !()`

Returns nonzero if `failbit` or `badbit` is set in the error state, which allows the use of such expressions as `if (!cin)` ...

`int operator void *()`

Converts an `ios` object to a pointer so that it can be compared to 0. The conversion returns a nonzero value (not meant for further use) if neither `failbit` nor `badbit` is set in the error state. This allows the use of such expressions as

`if (cin)` ...

and `if (cin >> x)` …

Other Member Functions

`int bad() const`

Returns a nonzero value if `badbit` is set in the error state; otherwise, it returns 0.This usually indicates that some operation on `rdbuf()` has failed, and that continued operations on the associated `streambuf` object may not be possible.

`long bitalloc()`

Returns a long integer with a single, previously unallocated bit set. This gives you an additional flag should you need one (to pass to `ios::set()`, for example).

`void clear(int state)`

Stores an integer value as the error state. A 0 value clears all bits.

`int eof() const`

Returns a nonzero value if `eofbit` is set in the error state; otherwise, it returns 0.This bit is usually set during an extraction and when an end-of-file has been encountered.

`int fail() const`

Returns a nonzero value if either `badbit` or `failbit` is set in the error state; otherwise, it returns 0. This usually indicates that some extraction or conversion operation has failed, but that the stream remains usable; once `failbit` clears, operations on the stream can usually continue.

`char fill() const`

Returns the variable currently used as the fill (padding) character.

`char fill(char c)`

Sets *c* as the fill (padding) character if one is needed (see `width ()`) and returns the previous value. The default fill character is a space. The `right`, `left`, and `internal` flags determine positioning of the fill character. A parameterized manipulator, `setfill`, is also available for setting the fill character.

`long flags() const`

Returns the current format flags.

`long flags(long f)`

Resets all the format flags to those specified in *f* and returns the previous settings. The flags are as follows:

`skipws`	For scalar operations, instructs the arithmetical extractor to skip white space before beginning conversion. As a precaution against looping, arithmetical extractors signal an error if the next character is white space and the skip variable is not set.

`left`
`right`
`internal`
	Control padding of values. The `left` flag adds a fill character after a value, `right` adds a fill character before a value, and `internal` adds a fill character after any leading sign or base indication, but before the value. Right-adjustment is the default if none of these flags are set. The fields are collectively identified by the static member `ios::adjustfield`. The fill character is controlled by the `fill()` function and the width after the padding is controlled by the `width()` function.

`dec`
`oct`
`hex`
	Control the conversion base of a value. Insertions are in decimal if none of these flags are set. Extractions follow VSI C++ lexical conventions for integral constants. The flags are collectively identified by the static member `ios::basefield`. The manipulators `hex`, `dec`, and `oct` are also available for setting the conversion base.

`showbase`	Converts insertions to an external form that can be read according to the VSI C++ lexical conventions for integral constants. By default, `showbase` is not set.
`showpos`	Inserts a plus sign (+) into a decimal conversion of a positive integral value.
`uppercase`	Uses an uppercase X for hexadecimal conversion when `showbase` is set, or uses uppercase E to print floating-point numbers in scientific notation. By default, `uppercase` is not set.
`showpoint`	Specifies that trailing zeros and decimal points appear in the result of a floating-point conversion.
`scientific` `fixed`	Control the format to which a floating-point value is converted for insertion into a stream. These two flags are collectively identified by the static member `ios::floatfield`. The `scientific` flag converts the value using scientific notation, with one digit before the decimal point. Depending on the `uppercase` flag, an E or an e introduces the exponent. The `fixed` flag converts the value to decimal notation. For both flags, the `precision` function determines the number of digits following the decimal point (6 is the default). If neither flag is set, then scientific notation is used only if the exponent from the conversion is less than –4 or greater

	than the precision. If `showpoint` is not set, trailing zeros are removed from the result and a decimal point appears only if followed by a digit.
`unitbuf`	Causes `ostream::osfx()` to perform a flush after each insertion. Unit buffering constitutes a performance compromise between buffered and unbuffered output.
`stdio`	Causes `ostream::osfx()` to flush `stdout` and `stderr` after each insertion.

`int good() const`

Returns a nonzero value if the error state has no bits set; otherwise, it returns 0.

`void init(streambuf *b)`

Initializes the `ios` object; intended for use by classes derived from `ios`.

`long& iword(int i)`

Returns a reference to the *i*th user-defined word, where *i* is an index into an array of words allocated by `ios::xalloc`.

`int precision() const`

Returns the precision format state variable.

`int precision(int i)`

Sets the precision format state variable to *i* and returns the previous value. The variable controls the number of significant digits inserted by the floating-point inserter. The default is 6. A parameterized manipulator, `setprecision`, is also available for setting the precision.

`void *&ios::pword(int i)`

Returns a reference to the *i*th user-defined word, where *i* is an index into an array of words allocated by `ios::xalloc`. This function differs from `iword()` only in type.

`streambuf *ios::rdbuf()`

Returns a pointer to the `streambuf` object that was associated with an `ios` object when the `ios` object was constructed.

`int rdstate() const`

Returns the current error state.

`long setf(long setbits)`

Makes available to the `streambuf` object associated with an `ios` object the format flags marked in *setbits* and returns the previous settings. A parameterized manipulator, `setiosflags`, performs the same function. If you are setting a flag that is part of a collection (for example, `basefield`), note that this manipulator does *not* clear the other flags in the collection.

`long setf(long setbits, long field)`

Clears, in the `streambuf` object associated with an `ios` object, the format flags specified by *field*, then resets these flags to the settings marked in *setbits*. It returns the previous settings. Specifying 0 in *setbits* clears all the bits specified in *field*, as does the parameterized manipulator, `resetioflags`.

void setstate(int state)

Changes only the bits specified in the state argument.

void sync_with_stdio()

Solves problems that arise with mixing `stdio` and `iostream` objects. When first called, the `sync_with_stdio()` function resets the standard iostream functions (`cin`, `cout`, `cerr`, and `clog`) to be streams using `stdiobuf` objects. Subsequently, input and output using these streams may be mixed with input and output using the corresponding `FILE` parameters (`stdin`, `stdout`, and `stderr`), and properly synchronized. The `sync_with_stdio()` function makes `cout` and `cerr` unit buffered (see `ios::unitbuf` and `ios::stdio`). Invoking `sync_with_stdio()` degrades performance variably; the shorter the strings being inserted, the greater the degradation.

ostream *ios::tie() const

Returns the tie variable (see the following member function description).

ostream *ios::tie(ostream *osp)

Sets the tie variable to *osp* and returns its previous value. The tie variable supports automatic flushing of `ios` objects. The `ios` object that the tie variable points at is flushed if the variable is not null, and an `ios` object either needs more characters or has characters to be consumed. By default, `cin` is initially tied to `cout` so that attempts to get more characters from standard input result in flushing standard output. Additionally, `cerr` and `clog` are tied to `cout` by default. By default, the tie variable is set to 0 for other `ios` objects.

long unsetf(long setbits)

Unsets, in the `streambuf` object associated with an `ios` object, the bits set in *setbits*; it returns the previous settings.

int width() const

Returns the field-width format variable (see the following member function description). The field width setting within the `ios` class is ignored during single character output: operator<<(char) and operator<<(unsigned char).

int width(int n)

Sets the field-width format variable to *n* and returns the previous value. The field width specifies a minimum number of characters for inserters. When the variable is 0 (the default), inserters insert only as many characters as needed to represent the value being inserted. When the variable is nonzero, and the value being inserted needs fewer than field-width characters to be represented, inserters insert at least that many characters using the fill character to pad the value. Numeric inserters do not truncate values even if the value being inserted is more than field-width characters. After each insertion or extraction, the field-width format variable resets to 0. A parameterized manipulator, `setw`, is also available for setting the field width.

int xalloc()

Returns a previously unused index into an array of words available for use by derived classes as format state variables.

Examples

1. ```
   cout.width(6);
   cout << x << " " << y;
   ```

   Outputs `x` in at least six characters, but uses only as many characters as needed for the separating space and `y`.

   In the following examples, `mystrm` is an `ios` object.

2. ```
   mystrm.clear(ios::badbit|s.rdstate())
   ```

   Sets the `badbit` member of the `io_state` enumerated data type without clearing previously set bits.

3. ```
   mystrm.setf(ios::hex,ios::basefield)
   ```

   Changes the conversion base in `mystrm` to be hexadecimal.

iostream class

iostream class — Provides the means to both insert into and extract from a single sequence of characters.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```
```

Declaration

```
class iostream: public istream, public ostream
{
public:
                iostream(streambuf *);
    virtual     ~iostream();

protected:
                iostream();
};
```

Description

This class combines the `istream` and `ostream` classes. You use it to carry out bidirectional operations (inserting into and extracting from a single sequence of characters).

Constructors and Destructors

`iostream()`

Constructs an `iostream` object, in undefined form, to enable inheritance by derived classes.

`iostream(streambuf *b)`

Constructs an `iostream` object. It initializes `ios` state variables and associates the `iostream` object with the `streambuf` object pointed to by *b*.

`virtual ~iostream()`

Deletes an `iostream` object.

iostream_withassign class

iostream_withassign class — Adds an assignment operator and a constructor with no operands to the `iostream` class.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class iostream_withassign: public iostream
{
public:
                   iostream_withassign();
    virtual       ~iostream_withassign();
```

```
    iostream_withassign &operator=(iostream &);
    iostream_withassign &operator=(streambuf *);
};
```

Description

This class adds an assignment operator and a constructor with no operands to the `iostream` class.

Constructors and Destructors

`iostream_withassign()`

Constructs an `iostream_withassign` object; it does no initialization.

`virtual ~iostream_withassign()`

Deletes an `iostream_withassign` object; no user action is required.

Overloaded Operators

`iostream_withassign &operator = (iostream &)`

Associates `iostream->rdbuf()` with an `iostream_withassign` object and initializes the entire state of that object.

`iostream_withassign &operator = (streambuf *)`

Associates `streambuf*` with an `iostream_withassign` object and initializes the entire state of that object.

istream class

istream class — Supports interpretation of characters extracted from an associated `streambuf` object.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class istream : virtual public ios
{
public:
                        istream(streambuf *);
    virtual             ~istream();

    inline int          gcount();
    istream             &get(char *ptr, int len,
                              char delim = '\n');
    istream             &get(unsigned char *ptr, int len,
```

```
                                char delim = '\n');
    istream             &get(char &);
    inline istream      &get(unsigned char &);
    istream             &get(streambuf &sb, char delim = '\n');
    int                 get();
    istream             &getline(char *ptr, int len,
                                char delim = '\n');
    istream             &getline(unsigned char *ptr, int len,
                                 char delim = '\n');
    istream             &ignore(int len = 1,
                                int delim = EOF);
    int                 ipfx(int need = 0);
    void                isfx();
    int                 peek();
    istream             &putback(char);
    istream             &read(char *s, int n);
    inline istream      &read(unsigned char *s, int n);
    istream             &seekg(streampos);
    istream             &seekg(streamoff, seek_dir);
    void                skipwhite();
    int                 sync();
    streampos           tellg();
    istream             &operator>>(char *);
    istream             &operator>>(char &);
    istream             &operator>>(short &);
    istream             &operator>>(int &);
    istream             &operator>>(long &);
    istream             &operator>>(float &);
    istream             &operator>>(double &);
    istream             &operator>>(unsigned char *);
    istream             &operator>>(unsigned char &);
    istream             &operator>>(unsigned short &);
    istream             &operator>>(unsigned int &);
    istream             &operator>>(unsigned long &);
    istream             &operator>>(streambuf *);
    inline istream      &operator>>(istream &(*f)(istream &));
    istream             &operator>>(ios &(*f)(ios &));

protected:
                        istream();

};
```

Description

This class provides facilities for formatted and unformatted extraction from `streambuf` objects.

Constructors and Destructors

`istream(streambuf *sb)`

Constructs an `istream` object. It initializes `ios` state variables and associates the `istream` object with the buffer pointed to by *sb*.

`virtual ~istream()`

Deletes an `istream` object.

Overloaded Operators

The following operators are all formatted input extractors. Given the expression *ins* >> *x*, these operators extract characters from *ins* and convert them to the variable *x*. The argument to the operator determines the type of *x*. Extractions are performed only if a call to `ipfx(0)` returns a nonzero value. Errors are indicated by setting the error state of *ins*. `ios::failbit` means that characters in *ins* did not represent the required type. `ios::badbit` means that attempts to extract characters failed. *ins* is always returned. The details of conversion depend on the values of the *ins* object format state flags and variables, and the type of *x*. Extractions that use `width` reset it to 0; otherwise, the extraction operators do not change the value of the `istream` object format state.

`istream &operator >> (char &x)`

Extracts a character and stores it in *x*.

`istream &operator >> (char *x)`

Extracts characters and stores them in the array pointed at by *x*, until a white-space character is found in the `iostream` object. The action leaves the terminating white-space character in the `iostream` object. If the `iostream` object's `width()` is nonzero, it is taken to be the size of the array and no more than `width()`–1 characters are extracted. A terminating null character (`'\0'`) is always stored, even if nothing else is done because of the `iostream` object's error state. The `iostream` object's `width()` is reset to 0.

`istream &operator >> (short &x)`

Extracts characters and converts them to an integral value according to the conversion specified in the `iostream` object's format flags. Converted values are stored in *x*. The first character can be a sign (- or +). After that, the conversion is octal if `ios::oct` is set in the `iostream` object's `flags`, decimal if `ios::dec` is set, or hexadecimal if `ios::hex` is set.

The first nondigit that is left in the `iostream` object terminates the conversion. If no conversion base flag is set, the conversion proceeds according to the VSI C++ lexical conventions: if the first characters (after the optional sign) are `0x` or `0X`, the conversion is hexadecimal; if the first character is `0`, the conversion is octal; otherwise, the conversion is decimal. If no digits are available (not counting the `0` in `0x` or `0X` during hex conversion), `ios::failbit` is set.

`istream &operator >> (float &x)`

Extracts characters and converts them according to the VSI C++ syntax for a `float` value or a `double` value. Converted values are stored in *x*. If no digits are available in the `iostream` object, or if the `iostream` object does not begin with a well formed floating-point or `double` number, `ios::failbit` is set.

`istream &operator >> (streambuf *b)`

Keeps getting characters from `ios` and inserting them into the buffer *b* until `EOF` is reached, if `ios::ipfx(0)` returns nonzero. Always returns the `iostream` object.

`istream &operator >> (ios &(*f)(ios &))`

Calls an `ios` object manipulator function *f* for an `istream` object.

`istream &operator >> (istream &(*f)(istream &))`

Calls an `istream` object manipulator function *f* for an `istream` object.

Other Member Functions

The unformatted input extractors, `get`, `getline`, `ignore`, and `read`, are among these functions. Before performing any extractions, these extractors, plus the unformatted function `peek` (which returns the next character without extracting it), call `ipfx(1)` and proceed only if a nonzero value is returned.

`int gcount()`

Returns the number of characters extracted by the last unformatted input function (`get`, `getline`, `ignore`, and `read`). Note that formatted input functions can call unformatted input functions and also reset this number.

`int get()`

Extracts a character and returns it, or returns `EOF` if the extraction encounters the end-of-file. It never sets `ios::failbit`.

`istream &get(char &ptr)`

Extracts a single character and stores it in *&ptr*.

`istream &get(char *ptr, int len, char delim)`

Extracts characters and stores them in the byte array beginning at *ptr* and extending for *len* bytes. Extraction stops when any of the following conditions are met:

- The extractor encounters *delim* (*delim* is left in the `istream` object and not stored.)
- The `istream` object has no more characters.
- The array has only one byte left.

The function stores a terminating null, even if it does not extract any characters because of its error status. The extraction sets `ios::failbit` only if it reaches an end-of-file before storing any characters.

`istream &get(streambuf &sb, char delim)`

Extracts characters from an `istream` object `rdbuf()` function and stores them into *sb*. It stops if it encounters the end-of-file, if a store into *sb* fails, or if it encounters *delim* (which it leaves in the `istream` object). The function sets `ios::failbit` if the extraction stops because the store operation into *sb* fails.

`istream &getline(char *ptr, int len, char delim)`

Functions the same as `get(char *, int, char)` except that these extract a terminating *delim* character from an `istream` object. If *delim* occurs when exactly *len* characters have been extracted, a filled array is considered to be the cause of the termination and the extraction leaves this *delim* in the `istream` object.

`istream &ignore(int len, int delim)`

Extracts and discards up to *len* characters. Extraction stops prematurely if *delim* is extracted or the end-of-file is reached. If *delim* is `EOF`, it can never cause termination.

`int ipfx(int need)`

Returns 0 if the error state of an `istream` object is nonzero. If necessary (and if it is not null), the function flushes any `ios` tied to the `istream` object (see the description of `ios::tie()`). Flushing is considered necessary if *need* is set to 0 or if fewer than *need* characters are immediately available. If `ios::skipws` is set in the `istream` object's `flags()` function, and *need* is 0, then the function extracts the leading white-space characters from the `istream` object. The function returns 0 if an error occurs while skipping white space; otherwise, it returns a nonzero value.

void isfx()

Performs input suffix operations (used for internal processing).

int peek()

Begins by calling `ipfx(1)`. If that call returns 0, or if the `istream` object is at the end-of-file, the function returns `EOF`. Otherwise, it returns the next character without extracting it.

istream &putback(char c)

Tries to back up an `istream` object `rdbuf()` function. *c* must be the character before the get pointer belonging to the `istream` object `rdbuf()`. (Unless some other activity is modifying the `istream` object `rdbuf()`, this is the last character extracted from the `istream` object.) If *c* is not the character before the get pointer, the effect of the function is undefined; the backup may fail and set the error state. The `putback` function is a member of the `istream` object, but it never extracts characters so it does not call `ipfx`. However, it returns without doing anything if the error state is nonzero.

istream &read(char *s, int n)

Extracts *n* characters and stores them in the array begining at *s*. If it reaches the end-of-file before extracting *n* characters, the function stores whatever it can extract and sets `ios::failbit`. To determine the number of characters extracted, use the `istream` `gcount()` function.

istream &seekg(streampos)

Repositions the get pointer of an `istream` object `rdbuf()` function.

int sync()

Establishes consistency between internal data structures and the external source of characters. Calls an `istream` object `rdbuf()->sync()`, which is a virtual function, so the details depend on the derived class. Returns `EOF` to indicate errors.

void skipwhite()

Skips extracted white-space characters.

streampos tellg()

Returns the current position of the get pointer of an `istream` object `rdbuf()` function.

Examples

1. ```
   char c;
   cin.get(c);
   ```

   Extracts a single character from `cin`.

2. ```
   tmp.seekg(10,ios::cur)
   ```

Moves the point in a file from which information is read forward 10 bytes.

See Also

ios class

istream_withassign class

istrstream class

istream_withassign class

istream_withassign class — Adds an assignment operator and a constructor with no operands to the `istream` class.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class istream_withassign: public istream
{
public:
                        istream_withassign();
    virtual             ~istream_withassign();

    istream_withassign &operator=(istream &);
    istream_withassign &operator=(streambuf *);
};
```

Description

This class adds an assignment operator and a constructor with no operands to the `istream` class.

Constructors and Destructors

`istream_withassign()`

Constructs an `istream_withassign` object; it does no initialization.

`virtual ~istream_withassign()`

Deletes an `istream_withassign` object; no user action is required.

Overloaded Operators

`istream_withassign &operator = (istream &s)`

Associates an `istream` object's `rdbuf()` function with an `istream_withassign` object and initializes the entire state of that object.

`istream_withassign &operator = (streambuf *sb)`

Associates *sb* with an `istream_withassign` object and initializes the entire state of that object.

istrstream class

istrstream class — Specializes the `istream` class to perform extractions from arrays of bytes in memory.

Header File

```
#include <strstream.hxx>
```

Alternative Header

```
#include <strstream.h>
```

Declaration

```
class istrstream: public istream
{
public:
                        istrstream(char *);
                        istrstream(char *, int);

    strstreambuf        *rdbuf();
};
```

Description

Objects of this class perform in-core extractions from arrays of bytes in memory.

Constructors and Destructors

`istrstream(char *cp)`

Constructs an `istrstream` object and fetches characters from the (null terminated) string *cp*. The terminating null character does not become part of the sequence. Seeks (`istream::seekg()`) are permitted within the allocated space.

`istrstream(char *cp, int len)`

Constructs an `istrstream` object and fetches characters from the array beginning at *cp* and extending for *len* bytes. Seeks (`istream::seekg()`) are permitted anywhere within that array.

Member Function

`strstreambuf *rdbuf()`

Returns the `strstreambuf` object associated with the `istrstream` object.

OAPP(TYPE) class

OAPP(TYPE) class — For an `ostream` object, declares predefined parameterized applicators.

Header File

(`#include <iomanip.hxx>`

Alternative Header

`#include <iomanip.h>`

Compile-Time Parameter

TYPE — The type of the `ostream` object. It must be an identifier.

Declaration

```
class OAPP(TYPE)
{
public:
    OAPP(TYPE)(ostream &(*f)(ostream &, TYPE));
    OMANIP(TYPE) operator()(TYPE a);
};
```

Constructor

`OAPP(TYPE)(ostream &(*f)(ostream &, TYPE))`

Creates an applicator.

Operator

`OMANIP(TYPE) operator () (TYPE a)`

Casts an object of type *a* into a manipulator function for an `ostream` object.

See Also

OMANIP(TYPE) class

ofstream class

ofstream class — Supports output to files.

Header File

`#include <fstream.hxx>`

Alternative Header

`#include <fstream.h>`

Declaration

```
class ofstream: public ostream
{
public:
                 ofstream();
                 ofstream(const char *name, int mode = ios::out,
                         int prot = filebuf::openprot);
                 ofstream(int fd);
                 ofstream(int fd, char *p, int len);
                 ~ofstream();

    void         attach(int fd);
    void         close();
    void         open(const char *name, int mode = ios::out,
                         int prot = filebuf::openprot);
    filebuf      *rdbuf();
    void         setbuf(char *p, int len);
};
```

Description

This class specializes the `ostream` class to files using a `filebuf` object to do the output. Your program can perform common operations, such as opening and closing files, without explicitly mentioning `filebuf` objects.

Constructors and Destructors

`ofstream()`

Constructs an unopened `ofstream` object.

`ofstream(int fd)`

Constructs an `ofstream` object connected to a file whose descriptor is passed through the *fd* argument. The file must already be open.

`ofstream(int fd, char *p, int len)`

Constructs an `ofstream` object connected to a file whose descriptor is passed through the *fd* argument, and also initializes the associated `filebuf` object to use the *len* bytes starting at *p* as the reserve area. If *p* is null or *len* is 0, the `filebuf` object is unbuffered.

`ofstream(const char *name, int mode, int prot)`

Constructs an `ofstream` object and opens the file specified by the *name* argument. The *mode* and *prot* arguments specify the file open mode and protection. By default, *prot* is `filebuf::openprot`. If the open fails, the error state (`io_state`) of the constructed `ofstream` object indicates failure.

`~ofstream()`

Deletes an `ofstream` object.

Member Functions

void attach(int fd)

Connects an `ofstream` object to a file whose descriptor is passed through the *fd* argument. A failure occurs when the `ifstream` object is connected to a file, in which case `ios::failbit` is set in the `ofstream` object error state.

void close()

Closes any associated `filebuf` object and consequently breaks the connection of the `ofstream` object to the file. The error state of the `ofstream` object is cleared except on failure. A failure occurs when the call to the `filebuf` object `close()` function fails.

void open(const char *name, int mode, int prot)

Opens a file specified by the *name* argument and connects the `ofstream` object to it. If the file does not exist, the function tries to create it with the protection specified by the *prot* argument unless `ios::nocreate` is set. By default, *prot* is `filebuf::openprot`.

Failure occurs if the `ofstream` object is open or when the call to the `filebuf` object `open()` function fails, in which case `ios::failbit` is set in the `filebuf` object's error state. The members of `open_mode` are bits that may be joined together by `or` (and because this joining takes an `int`, `open()` takes an `int` rather than an `open_mode` argument). For an explanation of the meanings of these bits in `open_mode`, see the Enumerated Types section for the `ios` class.

filebuf *rdbuf()

Returns a pointer to the `filebuf` object associated with the `ofstream` object. This function has the same meaning as `ios::rdbuf()`, but has a different type.

void setbuf(char *p, int len)

Calls the associated `filebuf` object `setbuf()` function to request space for a reserve area. A failure occurs if the `filebuf` object is open or if the call to `rdbuf()->setbuf` fails for any other reason.

OMANIP(TYPE) class

OMANIP(TYPE) class — For an `ostream` object, declares predefined parameterized manipulators and provides macros for user-defined parameterized manipulators.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `ostream` object. It must be an identifier.

Declaration

```
class OMANIP(TYPE)
```

```
{
public:
    OMANIP(TYPE)(ostream &(*f)(ostream &, TYPE), T a );
    friend ostream &operator<<(ostream & s, OMANIP(TYPE) &m);
};
```

Description

These manipulators serve the `ostream` class by producing some useful effect, such as embedding a function call in an expression containing a series of insertions and extractions. You also can use manipulators to shorten the long names and sequences of operations required by the `ostream` class.

In its simplest form, a manipulator takes an `ostream&` argument, operates on it in some way, and returns it.

Constructor

`OMANIP(TYPE)(ostream &(*f)(ostream &, TYPE), T a )`

Creates a manipulator.

Operator

`ostream &operator << (ostream & s, OMANIP(TYPE) &m)`

Sends data to an `ostream` object.

ostream class

ostream class — Supports insertion into `streambuf` objects.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class ostream : virtual public ios
{
public:
                        ostream(streambuf *);
    virtual             ~ostream();

    ostream             &flush();
    int                 opfx();
    void                osfx();
    ostream             &put(char c);
    ostream             &seekp(streampos);
    ostream             &seekp(streamoff, seek_dir);
```

```
    streampos           tellp();
    ostream             &write(const char *ptr, int n);
    inline ostream      &write(const unsigned char *ptr, int n);
    ostream             &operator<<(const char *);
    ostream             &operator<<(char);
    inline ostream      &operator<<(short);
    ostream             &operator<<(int);
    ostream             &operator<<(long);
    ostream             &operator<<(float);
    ostream             &operator<<(double);
    ostream             &operator<<(const unsigned char *);
    inline ostream      &operator<<(unsigned char);
    inline ostream      &operator<<(unsigned short);
    ostream             &operator<<(unsigned int);
    ostream             &operator<<(unsigned long);
    ostream             &operator<<(void *);
    ostream             &operator<<(streambuf *);
    inline ostream      &operator<<(ostream &(*f)(ostream &));
    ostream             &operator<<(ios &(*f)(ios &));

protected:
                        ostream();
};
```

Description

Objects of this class perform formatted and unformatted insertions into `streambuf` objects.

Constructors and Destructors

`ostream(streambuf *b)`

Constructs an `istream` object. It initializes `ios` state variables and associates the buffer *b* with the `ostream` object.

`virtual ~ostream()`

Deletes an `ostream` object.

Overloaded Operators

The following operators are all formatted output inserters. Given the expression *outs* << *x*, these operators insert into *outs*.`rdbuf()` a sequence of characters representing *x*. The argument to the operator determines the type of *x*. Insertions are performed after a call to *outs*.`opfx()` only if that call returns nonzero. Errors are indicated by setting the error state of the `ostream` object. The `ostream` object is always returned.

Conversion of *x* to a sequence of characters depends on the type of *x* and on the values of the `ostream` object's format state flags and variables. Padding occurs after this representation is determined. If `width()` is greater than 0, and the representation contains fewer than `width()` characters, then the function adds enough `fill()` characters to bring the total number of characters to `ios::width()`. If `ios::left()` is set, the sequence is left-adjusted; that is, the function puts the padding after the sequence of characters. If `ios::right()` is set, the padding is added before the character sequence. If `ios::internal()` is set, the padding is added after any leading sign or base indication and before the characters that represent the value. `ios::width()` is reset to 0 but all other format variables

are unchanged. The full sequence (padding plus representation) is inserted into the `ostream` object `rdbuf()` function.

`ostream &operator << (char x)`

Inserts a character *x*. No special conversion is needed.

`ostream &operator << (const char *x)`

Inserts a sequence of characters up to (but not including) the terminating null of the string that *x* points at.

`ostream &operator << (short x)`

Inserts characters as follows:

- If *x* is positive, the representation contains a sequence of octal digits if `ios::oct` is set in the `ios` object format flags, decimal digits if `ios::dec` is set, or hexadecimal digits if `ios::hex` is set. If none of these flags are set, the conversion defaults to decimal.
- If *x* is negative, decimal conversion includes a minus sign (–) followed by decimal digits.
- If *x* is positive and `ios::showpos` is set, decimal conversion includes a plus sign (+) followed by decimal digits.
- Conversions other than decimal treat all values as unsigned.
- If `ios::showbase` is set, the hexadecimal representation contains `0x` before the hexadecimal digits or `0X` if `ios::uppercase` is set; the octal representation contains a leading 0.

`ostream &operator << (float x)`

Converts the arguments according to the current values of the `ostream` object's `precision()` function, the `ostream` object's `width()` function, and the `ostream` object's format flags: `ios::scientific`, `ios::fixed`, and `ios::uppercase`. The default value for the `ostream` object's `precision()` function is 6. If neither `ios::scientific` nor `ios::fixed` is set, the value of *x* determines whether the representation uses scientific or fixed notation.

`ostream &operator << (void *v)`

Converts pointers to integral values and then converts them to hexadecimal numbers as if `ios::showbase` was set.

`ostream &operator << (streambuf *sb)`

Given the expression *outs* << *sb*, inserts into *sb*.`rdbuf()` the sequence of characters that can be fetched from *sb*. When no more characters can be fetched from *sb*, insertion stops. This function does no padding. It always returns the `ostream` object.

`ostream &operator << (ios &(*f)(ios &))`

Calls an `ios` object manipulator function *f* for an `ostream` object.

`ostream &operator << (ostream &(*f)(ostream &))`

Calls an `ostream` object manipulator function *f* for an `ostream` object.

Other Member Functions

`ostream &flush()`

Calls the `ostream` object's `rdbuf()->sync()` function to consume (that is, write to the external file) any characters that may have been stored into a `streambuf` object but are not yet consumed.

`int opfx()`

Performs output prefix actions. If the error state of the `ostream` object is nonzero, it returns immediately. If the value of the `ostream` object's `tie()` function is not null, it is flushed. The function returns nonzero except when the error state of the `ostream` object is nonzero.

`void osfx()`

Performs output suffix actions before returning from inserters. If `ios::unitbuf` is set, this function flushes the `ostream` object. If `ios::stdio` is set, the function flushes `stdout` and `stderr`. It is called by all predefined inserters, and should also be called by user-defined inserters after any direct manipulation of the `streambuf` object. It is not called by the binary output functions.

`ostream &ostream::put(char c)`

Inserts *c* into the `ostream` object's `rdbuf()` function. It sets the error state if the insertion fails.

`ostream &seekp(streampos)`

Repositions the put pointer of the `ostream` object's `rdbuf()` function.

`streampos tellp()`

Returns the current position of the put pointer belonging to the `ostream` object's `rdbuf()` function.

`ostream &write(const char *ptr, int n)`

Inserts the *n* characters starting at *ptr* into the `ostream` object's `rdbuf()` function. These characters may include zeros; that is, *ptr* need not be a null-terminated string.

Example

```
char c = 'Z';
cout.put(c);
```

Inserts a single character (Z) into `cout`.

See Also

ostream_withassign class

ostrstream class

ostream_withassign class

ostream_withassign class — Adds an assignment operator and a constructor with no operands to the `ostream` class.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class ostream_withassign: public ostream
{
public:
                        ostream_withassign();
    virtual             ~ostream_withassign();

    ostream_withassign  &operator=(ostream &);
    ostream_withassign  &operator=(streambuf *);
};
```

Description

This class adds an assignment operator and a constructor with no operands to the `ostream` class.

Constructors and Destructors

`ostream_withassign()`

Constructs an `ostream_withassign` object; it does no initialization.

`virtual ~ostream_withassign()`

Deletes an `ostream_withassign` object; no user action is required.

Overloaded Operators

`ostream_withassign &operator = (ostream &s)`

Associates *s*.`rdbuf()` with the `ostream_withassign` object and initializes the entire state of that object.

`ostream_withassign &operator = (streambuf *sb)`

Associates *sb* with an `ostream_withassign` object and initializes the entire state of that object.

ostrstream class

ostrstream class — Supports the insertion of characters into arrays of bytes in memory.

Header File

```
#include <strstream.hxx>
```

Alternative Header

```
#include <strstream.h>
```

Declaration

```
class ostrstream: public ostream
{

public:
                        ostrstream();
                        ostrstream(char *, int, int = ios::out);
                        ~ostrstream();

     int                pcount();
     strstreambuf       *rdbuf();
     char               *str();
};
```

Description

This class specializes the `ostream` class for in-core operations by providing members that insert characters into arrays of bytes in memory.

Constructors and Destructors

`ostrstream()`

Constructs an `ostrstream` object and dynamically allocates space to hold stored characters.

`ostrstream::ostrstream(char *cp, int n, int mode)`

Constructs an `ostrstream` object and stores characters into the array starting at *cp* and continuing for *n* bytes. If `ios::ate` or `ios::app` is set in *mode*, the function takes *cp* to be a null-terminated string and it begins storing at the null character; otherwise, it begins storing at *cp*. Seeks are allowed anywhere in the array.

`~ostrstream()`

Deletes an `ostrstream` object.

Member Functions

`int pcount()`

Returns the number of bytes that have been stored into the buffer. This function is useful when binary data has been stored and the `ostrstream` object `str()` function does not point to a null-terminated string.

`strstreambuf *rdbuf()`

Returns the `strstreambuf` associated with the `ostrstream` object.

`char *str()`

Returns a pointer to the array being used and freezes the array. After `str()` has been called, the effect of storing more characters into the `strstream` object is undefined. If the `strstream` object was constructed with an explicit array, the function returns a pointer to the array; otherwise, it returns a pointer to a dynamically allocated area. Until `str()` is called, deleting the dynamically allocated area is the responsibility of the `strstream` object. After `str()` returns, dynamic allocation becomes the responsibility of the user program.

Example

```
char *bptr = bf.str()
```

Initializes the variable *bptr* with the address of the array associated with the `ostrstream` object `bf`. This lets you manipulate the array through *bptr* just as you would any character array.

SAPP(TYPE) class

SAPP(TYPE) class — Defines parameterized applicators for an `ios` object.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `ios` object. It must be an identifier.

Declaration

```
class SAPP(TYPE)

{
public:
    SAPP(TYPE)(ios &(*f)(ios &, TYPE));
    SMANIP(TYPE) operator()(TYPE a);
};
```

Constructor

`SAPP(TYPE)(ios &(*f)(ios &, TYPE))`

Creates an applicator.

Operator

`SMANIP(TYPE) operator () (TYPE a)`

Casts an object of type *a* into a manipulator function for an `istream` or `ostream` object.

See Also

SMANIP(TYPE) class

SMANIP(TYPE) class

SMANIP(TYPE) class — Defines parameterized manipulators for an `ios` object.

Header File

```
#include <iomanip.hxx>
```

Alternative Header

```
#include <iomanip.h>
```

Compile-Time Parameter

TYPE — The type of the `ios` object. It must be an identifier.

Declaration

```
class SMANIP(TYPE)
{
public:
    SMANIP(TYPE)(ios &(*f)(ios &, TYPE), TYPE a);
    friend istream &operator>>(istream &i, SMANIP(TYPE) &m);
    friend ostream &operator<<(ostream &o, SMANIP(TYPE) &m);
};
```

Description

These manipulators serve the `ios` class by producing some useful effect, such as embedding a function call in an expression containing a series of insertions and extractions. You also can use manipulators to shorten the long names and sequences of operations required by the `ios` class.

In its simplest form, a manipulator takes an `ios&` argument, operates on it in some way, and returns it.

Constructor

`SMANIP(TYPE)(ios &(*f)(ios &, TYPE), TYPE a)`

Creates a manipulator.

Operators

`ostream &operator << (ostream &o, SMANIP(TYPE) &m)`

Sends data to an `ostream` object.

`istream &operator >> (istream &i, SMANIP(TYPE) &m)`

Takes data from an `istream` object.

stdiobuf class

stdiobuf class — Provides input/output facilities through `stdio FILE`.

Header File

```
#include <stdiostream.hxx>
```

Alternative Header

```
#include <stdiostream.h>
```

Declaration

```
class stdiobuf: public streambuf
{
public:
                        stdiobuf(FILE *f);

    virtual int         overflow(int = EOF);
    virtual streampos   seekoff(streamoff, seek_dir, int mode);
    FILE                *stdiofile();
    virtual int         sync();
    virtual int         underflow();
};
```

Description

This class specializes the `streambuf` class for `stdio FILE`. It uses unbuffered mode causing all operations to be reflected immediately in the `stdio FILE`.

Constructor

`stdiobuf(FILE *f)`

Constructs an empty `stdiobuf` object and connects it to the `stdio FILE` that the argument *f* points to.

Member Functions

`virtual int overflow(int c)`

Called to consume characters. If *c* is not `EOF`, this function must also either save *c* or consume it. Although it can be called at other times, this function is usually called when the put area is full and an attempt is being made to store a new character. The normal action is to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new put area, and (if *c* `!= EOF`) store *c* using `sputc()`. The `overflow(c)` function should return `EOF` to indicate an error; otherwise, it should return something else.

`virtual streampos seekoff(streamoff off, seek_dir dir, int mode)`

Repositions the abstract get and put pointers (not `pptr()` and `gptr()`). *mode* specifies whether to modify the put pointer (`ios::out` bit set), the get pointer, or both (`ios::in` bit set). *off* is

interpreted as a byte offset. For the meanings of *dir*, see the explanation of the enumerated type `seek_dir` in class `ios`.

A class derived from `streambuf` is not required to support repositioning. If the derived class does not, then `seekoff()` should return `EOF`. If the derived class does support repositioning, `seekoff()` should return the new position or `EOF` on error.

`FILE *stdiofile()`

Returns a pointer to the `stdio FILE` associated with the `stdiobuf` object.

`virtual int sync()`

Should consume any characters stored into the put area and, if possible, give back to the source any characters in the get area that have not been fetched. When `sync()` returns, there should be no unconsumed characters and the get area should be empty. If some kind of failure occurs, the function should return `EOF`.

`virtual int underflow()`

Called to supply characters for fetching; that is, to create a condition in which the get area is not empty. If this function is called when characters are in the get area, it should return the first character. If the get area is empty, it should create a nonempty get area and return the next character (which it should also leave in the get area). If no more characters are available, `underflow()` should return `EOF` and leave an empty get area.

stdiostream class

stdiostream class — Specializes the `iostream` class for `stdio FILE`.

Header File

```
#include <stdiostream.hxx>
```

Alternative Header

```
#include <stdiostream.h>
```

Declaration

```
class stdiostream: public iostream
{
public:
                stdiostream(FILE *f);
                ~stdiostream();

    stdiobuf    *rdbuf();
};
```

Description

This class specializes the `iostream` class for `stdio FILE`, and causes that class to use a `stdiobuf` object as its associated `streambuf` object.

In most other existing implementations, the `stdiostream` class is derived directly from the `ios` class rather than from the `iostream` class. Deriving the `stdiostream` class from the `ios` class limits its usefulness and, therefore, can be considered a historical mistake. Nevertheless, for maximum portability, you should use only those `stdiostream` features that originate from the `ios` class and avoid the features supplied by the `iostream` class.

Constructors and Destructors

`stdiostream(FILE *f)`

Constructs a `stdiostream` object whose `stdiobuf` object is associated with the `FILE` parameter that the *f* argument points to.

`~stdiostream()`

Deletes a `stdiostream` object and closes the associated `stdiobuf` object.

Member Function

`stdiobuf *rdbuf()`

Returns a pointer to the `stdiobuf` object associated with the `stdiostream` object.

streambuf class

streambuf class — Provides the buffer mechanism for streams.

Header File

```
#include <iostream.hxx>
```

Alternative Header

```
#include <iostream.h>
```

Declaration

```
class streambuf
{
public:
                        streambuf();
                        streambuf(char *p, int len);
    virtual             ~streambuf();
    void                dbp();

protected:
    int                 allocate();
    char                *base();
    int                 blen();

    virtual int  doallocate();
```

```
    char                *eback();
    char                *ebuf();
    char                *egptr();
    char                *epptr();
    void                gbump(int n);
    char                *gptr();
    char                *pbase();
    void                pbump(int n);
    char                *pptr();
    void                setb(char *b, char *eb, int a = 0);
    void                setg(char *eb, char *g, char *eg);
    void                setp(char *p, char *ep);
    int                 unbuffered();
    void                unbuffered(int n);

public:
    int                 fd();
    void                fd(int);
    FILE                *fp();
    void                fp(FILE *);
    int                 in_avail();
    int                 out_waiting();

    virtual int         overflow(int c = EOF);
    virtual int         pbackfail(int c);

    int                 sbumpc();

    virtual streampos   seekpos(streampos, int = ios::in
                        | ios::out);
    virtual streampos   seekoff(streamoff, seek_dir,
                        int = ios::in | ios::out);
    virtual streambuf   *setbuf(char *ptr, int len);

    streambuf           *setbuf(unsigned char *ptr, int len);
    streambuf           *setbuf(char *ptr, int len, int i);
    int                 sgetc();
    int                 sgetn(char *ptr, int n);
    int                 snextc();
    int                 sputbackc(char c);
    int                 sputc(int c = EOF);
    int                 sputn(const char *s, int n);
    void                stossc();

    virtual int         sync();
    virtual int         underflow();
};
```

Description

This class supports buffers into which you can insert (put) or extract (get) characters. It contains only the basic members for manipulating the characters. Also, several of its member functions are virtual; to implement virtual functions, you typically use a class derived from the `streambuf` class.

The protected members of the `streambuf` class present an interface to derived classes organized around the get, put, and reserve areas (arrays of bytes), which are managed cooperatively by the base and derived classes.

The reserve area is a sequence of characters with an associated get pointer, put pointer, or both. This area serves mainly as a resource in which to allocate space for the put and get areas. As characters enter and exit the reserve area, the put and get areas change but the reserve area remains fixed. A collection of character pointer values defines the three areas. These pointers infer a boundary condition; therefore, it may be helpful to consider such pointers as pointing just before the byte, even though they point directly at it.

Classes derived from `streambuf` vary in their handling of the get and put pointers. The simplest are unidirectional buffers that permit only get and put operations. Such classes serve as producers and consumers of characters. Queue-like buffers (such as `strstream` and `strstreambuf`) have a put and a get pointer that move independently of each other. In such buffers, stored characters are queued until later fetched. File-like buffers (such as `filebuf`) allow both get and put operations but have their get and put pointers linked together, so that when one pointer moves so does the other.

You can call virtual functions to manage the collections of characters in the get and put areas. Services supplied by virtual functions include fetching more characters from an ultimate producer and flushing a collection of characters to an ultimate consumer.

If your program expects a buffer to be allocated when none was allocated, then the iostream package allocates a default buffer.

Data Member

`void dbp()`

Writes directly on file descriptor 1 information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. What it prints out can be understood only in relation to the protected interface, but `dbp()` is a public domain function so that it can be called anywhere during debugging.

Constructors and Destructors

`streambuf()`

Constructs an empty buffer corresponding to an empty sequence.

`streambuf(char* base, int length)`

Constructs an empty buffer and then sets up the reserve area to be *length* bytes long starting at *base*.

`virtual ~streambuf()`

Deletes the reserve area if one is allocated.

Member Functions

`int allocate()`

Tries to set up a reserve area. If a reserve area already exists or is unbuffered, it returns 0 without doing anything. If the attempt to allocate space succeeds, `allocate()` returns 1; otherwise, it returns `EOF`. No nonvirtual member functions of `streambuf` call `allocate()`.

`char *base()`

Returns a pointer to the first byte of the reserve area. The space between `base()` and `ebuf()` is the reserve area.

`int blen()`

Returns the size, in type `char`, of the current reserve area.

`virtual int doallocate()`

In `streambuf`, it tries to allocate a reserve area using the `new` operator.

In classes derived from `streambuf`, this function is called when `allocate()` determines that space is needed. `doallocate()` is required to call `setb()`, to provide a reserve area, or to return `EOF` if it cannot. It is called only if both `unbuffered()` and `base()` are 0.

`char *eback()`

Returns a pointer to a lower bound on `gptr()`. The space between `eback()` and `gptr()` is available for putback operations.

`char *ebuf()`

Returns a pointer to the byte after the last byte of the reserve area.

`char *egptr()`

Returns a pointer to the byte after the last byte of the get area.

`char *epptr()`

Returns a pointer to the byte after the last byte of the put area.

`int fd()`

Returns the file descriptor associated with the `streambuf` object, if any; otherwise, it returns –1.

`void fd(int f)`

Sets the file descriptor associated with the `streambuf` object to *f*.

`FILE *fp()`

Returns the file pointer associated with the `streambuf` object, if any; otherwise, it returns 0.

`void fp(FILE *f)`

Sets the file pointer associated with the `streambuf` object to *f*.

`void gbump(int n)`

Increments `gptr()` by *n*, which can be a positive or a negative number. No checks are made on whether the new value of `gptr()`is in bounds.

`char *gptr()`

Returns a pointer to the first byte of the get area. The characters available are those between `gptr()` and `egptr()`. The next character fetched will be `*gptr()` unless `egptr()` is less than or equal to `gptr()`.

`int in_avail()`

Returns the number of characters immediately available in the get area for fetching. This number is the number of characters that can be fetched with confidence that an error will not be reported.

`int out_waiting()`

Returns the number of characters in the put area that have not been consumed (by the ultimate consumer).

`virtual int overflow(int c)`

In `streambuf`, this function should be treated as if its behavior is undefined; classes derived from `streambuf` should always define it.

In classes derived from `streambuf`, it is called to consume characters. If *c* is not `EOF`, `overflow(c)` also must either save *c* or consume it. Although it can be called at other times, this function is usually called when the put area is full and an attempt is being made to store a new character. The normal action is to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new put area, and (if *c* `!= EOF`) store *c* using `sputc()`. `overflow(c)` should return `EOF` to indicate an error; otherwise, it should return something else.

`virtual int pbackfail(int c)`

In `streambuf`, this function always returns `EOF`.

In classes derived from `streambuf`, this function is called when `eback()` equals `gptr()` and an attempt has been made to put *c* back. If this situation can be managed (for example, by repositioning an external file), `pbackfail(c)` should return *c*; otherwise, it should return `EOF`.

`char *pbase()`

Returns a pointer to the put area base. Characters between `pbase()` and `pptr()` are stored into the buffer but are not yet consumed.

`void pbump(int n)`

Increments `pptr()` by *n*, which can be positive or negative. No checks are made on whether the new value of `pptr()` is in bounds.

`char *pptr()`

Returns a pointer to the first byte of the put area. The space between `pptr()` and `epptr()` is the put area.

`int sbumpc()`

Moves the get pointer forward one character and returns the character it moved past. The function returns `EOF` if the get pointer is currently at the end of the sequence.

`virtual streampos seekoff(streamoff off, (ios::)seek_dir dir, int mode)`

In `streambuf`, this function returns `EOF`.

In classes derived from `streambuf`, it repositions the abstract get and put pointers (not `pptr()` and `gptr()`). *mode* specifies whether to modify the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) or both pointers. *off* is interpreted as a byte offset (it is a signed value). For the meanings of *dir*, see the explanation of the enumerated type `seek_dir` in class `ios`.

A class derived from `streambuf` is not required to support repositioning. If the derived class does not, then `seekoff()` should return `EOF`. If the derived class does support repositioning, `seekoff()` should return the new position or `EOF` on error.

`virtual streampos seekpos(streampos pos, int mode)`

In `streambuf`, this function returns `seekoff(streamoff(pos), ios::beg, mode)`. To define seeking in a derived class, you can often define `seekoff()` and use the inherited `streambuf::seekpos`.

In classes derived from `streambuf`, this function repositions the `streambuf` get pointer, put pointer, or both, to *pos*. *mode* specifies the affected pointers. `seekpos()` returns the argument *pos* or `EOF` if the class does not support repositioning or if an error occurs. `streampos(0)` signifies the beginning of the file; `streampos(EOF)` indicates an error.

`void setb(char *b, char *eb, int a)`

Sets `base()` to *b* and `ebuf()` to *eb*. The *a* argument controls whether the reserve area will be subject to automatic deletion. If *a* is nonzero, then *b* will be deleted when `base()` is changed by another call to `setb()`, or when the destructor is called for the `streambuf` object. If *b* and *eb* are both null, then the reserve area effectively does not exist. If *b* is nonnull, a reserve area exists even if *eb* is less than *b* (in which case the reserve area has 0 length).

`virtual streambuf *setbuf(char *ptr, int len)`

In `streambuf`, this function honors the request for a reserve area if there is none.

In classes derived from `streambuf`, this function offers for use as a reserve area the array at *ptr* with *len* bytes. Normally, if *ptr* or *len* is 0, the action is interpreted as a request to make the `streambuf` object unbuffered. The derived class has the choice of using or not using this area by accepting or ignoring the request. `setbuf()` should return a reference to the `streambuf` object if the derived class honors the request; otherwise, it should return 0.

`streambuf *setbuf(char *ptr, int len, int i)`

Offers the *len* bytes starting at *ptr* as the reserve area. If *ptr* is null, or *len* is 0 or negative, then the function requests an unbuffered state. Whether the offered area is used or a request for an unbuffered state is honored depends on details of the derived class. `setbuf()` normally returns a reference to the `streambuf` object, but if the derived class does not accept the offer or honor the request, `setbuf()` returns 0.

`void setg(char *eb, char *g, char *eg)`

Sets `eback()` to *eb*, `gptr()` to *g*, and `egptr()` to *eg*.

`void setp(char *p, char *ep)`

Sets `base()` and `pptr()` to *p* and `epptr()` to *ep*.

`int sgetc()`

Returns the character after the get pointer; it does not move the get pointer. It returns `EOF` if no character is available.

`int sgetn(char *ptr, int n)`

Fetches *n* characters following the get pointer and copies them to the area starting at *ptr*. If fewer than *n* characters occur before the end of the sequence, `sgetn()` fetches the characters that remain. It repositions the get pointer after the fetched characters and returns the number of characters fetched.

`int snextc()`

Moves the get pointer forward one character and returns the character after the new position. If the pointer is at the end of the sequence, either before or after moving forward, the function returns `EOF`.

`int sputbackc(char c)`

Moves the get pointer back one character. *c* must be the current content of the sequence just before the get pointer. The underlying mechanism may back up the get pointer or may rearrange its internal data structures so that *c* is saved. The effect is undefined if *c* is not the character before the get pointer. The function returns `EOF`, by calling `pbackfail()`, when it fails. The conditions under which it can fail depend on the details of the derived class.

`int sputc(int c)`

Stores *c* after the put pointer and moves the put pointer past the stored character (usually this extends the sequence). The function returns `EOF` when an error occurs. Conditions that can cause errors depend on the derived class.

`int sputn(const char *s, int n)`

Stores after the put pointer the *n* characters starting at *s*, and moves the put pointer past them. It returns the number of characters successfully stored. Normally *n* characters are successfully stored, but fewer characters may be stored when errors occur.

`void stossc()`

Moves the get pointer ahead one character. If the pointer started at the end of the sequence, `stossc()` has no effect.

`virtual int sync()`

In `streambuf` this function returns 0 if the get area is empty and no unconsumed characters are present; otherwise, it returns `EOF`.

In classes derived from `streambuf`, this function is called to let derived classes examine the state of the put, get, and reserve areas, and to synchronize these areas with any external representation. Normally `sync()` should consume any characters stored into the put area and, if possible, give back to the source any characters in the get area that have not been fetched. When `sync()` returns, no unconsumed characters should remain and the get area should be empty. If some kind of failure occurs, `sync()` should return `EOF`.

`int unbuffered()`

Returns the current buffering state flag, which is independent of the actual allocation of a reserve area. This function's primary purpose is to find out if a reserve area is being allocated automatically by `allocate()`.

void unbuffered(int n)

Sets the value of the current buffering state flag. If *n* equals 0, then the `streambuf` object is buffered; otherwise it is unbuffered. This function's primary purpose is to control whether a reserve area is allocated automatically by `allocate()`.

virtual int underflow()

In `streambuf`, this function should be treated as if its behavior is undefined; classes derived from `streambuf` must define it.

In classes derived from `streambuf`, it is called to supply characters for fetching; that is, to create a condition in which the get area is not empty. If this function is called when characters are in the get area, it should return the first character. If the get area is empty, it should create a nonempty get area and return the next character (which it should also leave in the get area). If no more characters are available, `underflow()` should return `EOF` and leave an empty get area.

Example

```
static const int bufsize = 1024;
char buf[bufsize] ;
int p, g ;
do {
        in->sgetc() ; ❶
        g = in->in_avail() ;  ❷
        if (g > bufsize) g = bufsize ;  ❸
        g = in->sgetn(buf,g) ;
        p = out->sput(buf,g) ;
        out->sync() ;  ❹
        if (p!=g) error("output error");
        } while (g > 0)
```

Provides a way to pass characters into the `in` and `out` arrays as soon as the characters become available (as when someone types them from a terminal) as follows:

❶ Ensures at least one character is immediately available in the `in` array (unless the get pointer is at the end of the sequence).
❷ Returns the number of characters immediately available.
❸ Checks that chunks in which the characters become available are less than `bufsize`, and that they fit into the arrays.
❹ Sends characters put into the `out` array to the ultimate consumer.

strstream class

strstream class — Specializes the `iostream` class for storing in and fetching from arrays of bytes.

Header File

```
#include <strstream.hxx>
```

Alternative Header

```
#include <strstream.h>
```

Declaration

```
class strstream: public iostream
{
public:
                    strstream();
                    strstream(char *, int, int);

    strstreambuf    *rdbuf();
    char            *str();
};
```

Description

This class specializes the `iostream` class for storing in and fetching from arrays of bytes. It handles all predefined data types, and provides an extensive set of options for performing input and output on these data types.

Constructors and Destructors

`strstream()`

Constructs an `strstream` object and dynamically allocates space to hold stored characters.

`strstream(char *cp, int n, int mode)`

Constructs an `strstream` object. It stores characters into the array starting at *cp* and continuing for *n* bytes. If `ios::ate` or `ios::app` is set in *mode*, *cp* is presumed to be a null-terminated string and storing begins at the null character; otherwise, storing begins at *cp*. Seeks are permitted anywhere in the array.

Member Functions

`strstreambuf *rdbuf()`

Returns a pointer to the `strstreambuf` object associated with a `strstream` object.

`char *str()`

Returns a pointer to an explicit array, to be used as the associated `strstreambuf` object, if the `strstream` object was constructed with such an array; otherwise, it returns a pointer to a dynamically allocated area. Until `str()` is called, deleting the dynamically allocated area is the responsibility of the `strstream` object. After `str()` returns, dynamic allocation becomes the responsibility of the user program. After `str()` has been called, the effect of storing more characters into the `strstream` object is undefined.

strstreambuf class

strstreambuf class — Specializes the `streambuf` class for input and output performed on arrays of bytes in memory.

Header File

```
#include <strstream.hxx>
```

Alternative Header

```
#include <strstream.h>
```

Declaration

```
class strstreambuf: public streambuf
{
public:
                        strstreambuf();
                        strstreambuf(char *, int, char *);
                        strstreambuf(int);
                        strstreambuf(unsigned char *, int,
                        unsigned char *);
                        strstreambuf(void *(*a)(long),
                        void (*f)(void *));

    void                freeze(int n = 1);
    virtual int         overflow(int);
    virtual streambuf   *setbuf(char *, int);
    char                *str();
    virtual int         underflow();
};
```

Description

Objects of this class let you use an array of bytes (a string of characters) in memory as a `streambuf` object for stream input/output operations on various kinds of data. Mapping between abstract get and put pointers and `char *` pointers is direct in the sense that a `char *` is interpreted as logically pointing immediately ahead of the `char` it actually points to. Moving the pointers corresponds to incrementing and decrementing the `char *` values.

To accommodate the need for strings of arbitrary length, this class supports a dynamic mode. When a `strstreambuf` object is in dynamic mode, space for the character is allocated as needed. When the sequence is extended too far, it is copied to a new array.

If your program expects a buffer to be allocated when none was allocated, then the iostream package allocates a default buffer, with a length specified by `BUFSIZ` as defined in `stdio.h`. The package then issues the following warning:

```
Warning; a null pointer to streambuf was passed to ios::init()
```

Constructors and Destructors

`strstreambuf()`

Constructs an empty `strstreambuf` object in dynamic mode. This means that space is automatically allocated to accommodate characters put into the `strstreambuf` object (using the `new` and `delete` operators). Because this may require copying the original characters, programs that have many characters to insert should use `setbuf()` to inform the `strstreambuf` object about the needed allocation of space, or to use one of the constructors that follow.

`strstreambuf(int n)`

Constructs an empty `strstreambuf` object in dynamic mode. The initial allocation of space is at least *n* bytes.

`strstreambuf(char *ptr, int n, char *pstart)`

Constructs a `strstreambuf` object to use the bytes starting at *ptr*. The `strstreambuf` object is in static mode; it does not grow dynamically. If *n* is positive, then the *n* bytes starting at *ptr* are used as the `strstreambuf` object. If *n* is 0, *ptr* is presumed to point to the beginning of a null-terminated string and the bytes of that string (not including the terminating null character) constitute the `strstreambuf` object. If *n* is negative, then the `strstreambuf` object is presumed to continue indefinitely.

The get pointer is initialized to *ptr*. The put pointer is initialized to *pstart*. If *pstart* is not null, then the initial sequence for fetching (the get area) consists of the bytes between *ptr* and *pstart*. If *pstart* is null, then storing operations are treated as errors and the initial get area consists of the entire array.

`strstreambuf(void *(*a)(long n), void (*f)(void *ptr))`

Constructs an empty `strstreambuf` object in dynamic mode. *a* is used as the allocator function in dynamic mode. The argument passed to *a* is a `long` denoting the number of bytes to be allocated. If the *a* argument is null, the `new` operator is used. *f* is used to free (or delete) get, put, or reserve areas returned by *a*. The argument to *f* becomes a pointer to the array allocated by *a*. If *f* is null, the `delete` operator is used.

Member Functions

`void freeze(int n)`

Inhibits (freezes) automatic deletion of the current array if *n* is nonzero, or permits (unfreezes) automatic deletion if *n* is 0. Deletion normally occurs when more space is needed, or when the `strstreambuf` object is being destroyed. Only space obtained through dynamic allocation is free. Storing characters into a `strstreambuf` that was dynamically allocated and is now frozen causes an error (the effect is undefined). If you want to resume storing characters in such a `strstreambuf` object you can thaw (unfreeze) it.

`virtual int overflow(int c)`

In classes derived from `streambuf`, it is called to consume characters. If *c* is not `EOF`, `overflow(c)` also must either save *c* or consume it. Although it can be called at other times, this function is usually called when the put area is full and an attempt is being made to store a new character. The normal action is to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new put area, and (if *c* `!= OF`) store *c* using `sputc()`. `overflow(c)` should return `EOF` to indicate an error; otherwise, it should return something else.

`virtual streambuf *setbuf(char *ptr, int n)`

Causes the `strstreambuf` object to remember *n* (if *ptr* is 0); this ensures that at least *n* bytes are allocated during the next dynamic mode allocation.

`char *str()`

Returns a pointer to the first character in the current array and freezes the `strstreambuf` object. If the `strstreambuf` object was constructed with an explicit array, the function returns a pointer to that array. If the `strstreambuf` object is in dynamic allocation mode but nothing has been restored yet, the returned pointer is null.

`virtual int underflow()`

In classes derived from `streambuf`, it is called to supply characters for fetching; that is, to create a condition in which the get area is not empty. If this function is called when characters are in the get area, it should return the first character. If the get area is empty, it should create a nonempty get area and return the next character (which it should also leave in the get area). If no more characters are available, `underflow()` should return `EOF` and leave an empty get area.

Chapter 5. Messages Package

The Messages package provides a way to retrieve messages stored in a catalog or file that is separate from your program. It consists of a single class, `Messages`, that retrieves the text of a message.

Processing a message file on an OpenVMS system requires a message set number and a message number. A message set number is an OpenVMS message identification code, including a facility code (bits 16 through 27) and a facility-specific bit (bit 15); all other bits should be 0. A message number is an integer from 1 to 8191. To process the message file, use the OpenVMS Message Utility (see the OpenVMS Message Utility Manual for details) and link the resulting object code into one of the following:

- Your program
- A shareable image that your program is linked against
- A shareable image that is then specified with the `set message` command

Messages class

Messages class — Retrieves message text for a message number.

Header File

```
#include <messages.hxx>
```

Alternative Header

None.

Declaration

```
class Messages
{
public:
   Messages(const char *filename_arg, int set_arg = 0,
        const char *default_file_location_arg = (const char *)(NULL));
   ~Messages();

   const char *text(int msg_arg, const char *fallback_text_arg,
         int set_arg = 0);
};
```

Constructors and Destructors

`Messages(const char *filename_arg, int set_arg, const char *default_file_loca`

Constructs a `Messages` object.

`~Messages()`

Deletes a `Messages` object.

Member Function

`const char *text(int msg_arg, const char *fallback_text_arg, int set_arg)`

Returns the text of the message specified by the *msg_arg* argument. The *fallback_text_arg* argument indicates the text to return if the message cannot be found. The *set_arg* argument specifies the message set number; a value of 0 causes the system to use the set number provided to the constructor.

Example

The following is a sample message source file:

```
.TITLE MESSAGES_EXAMPLE_MSG Example messages -- VMS message catalog
.IDENT '1.0'
.FACILITY EXAMPLE, 1 /PREFIX=EXAMPLE_
.BASE 0
.SEVERITY WARNING ! we just want a 0 in the severity field
SET <> ! message set number
.SEVERITY ERROR
EXAMPLE_ERROR <This is an example error message>
.END
Entering the following OpenVMS Message Utility commands set the
appropriate options and compile this file:
$ set message/nofac/nosev/noid
$ message/lis MESSAGES_EXAMPLE_MSG
```

Entering the following OpenVMS Message Utility commands set the appropriate options and compile this file:

```
$ set message/nofac/nosev/noid
$ message/lis MESSAGES_EXAMPLE_MSG
```

The following program retrieves the sample error message:

```
#include <iostream.hxx>
#include <messages.hxx>
const char *message_file_name = (const char *)(NULL);
const char *message_file_location = (const char *)(NULL);
#pragma __extern_model __save
#pragma __extern_model __globalvalue
extern int EXAMPLE_SET;
#pragma __extern_model __restore
int message_set_example = EXAMPLE_SET;
Messages m_example (message_file_name, message_set_example,
message_file_location);
int main()
{
cout <<
"text of example message 1: " <<
m_example.text(1, "fallback message 1") <<
"\n";
cout <<
"text of example message 2: " <<
m_example.text(2, "fallback message 2") <<
"\n";
return 0;
}
```

The following compiler command compiles the program:

```
$ cxx/lis MESSAGES_EXAMPLE
```

Entering the following link and run sequence retrieves the text of the error message and displays the second fallback message:

```
$ link MESSAGES_EXAMPLE,MESSAGES_EXAMPLE_MSG
$ run/nodeb messages_example
text of example message 1: This is an example error message
text of example message 2: fallback message 2
```

Chapter 6. Mutex Package

The Mutex package provides a way to synchronize access to user-defined objects. It consists of a single class, `Mutex`, that manages the creation, locking and unlocking of `Mutex` objects.

Construction of a `Mutex` object creates a recursive mutex that users can lock and unlock using the appropriate member functions or parameterized manipulators. A **recursive mutex** is a mutex that can be locked many times by the same thread without causing the thread to enter a deadlock state. To completely unlock this kind of mutex, the thread must unlock the mutex the same number of times that the thread locked the mutex.

Note

User-defined objects are not automatically thread safe. Users must supply synchronization for such objects if they are shared between threads.

Mutex class

Mutex class — Provides a means whereby users can synchronize access to user-defined objects.

Header File

```
#include <mutex.hxx>
```

Alternative Header

```
#include <mutex.h>
```

Declaration

```
class Mutex
{
public:
             Mutex();
             ~Mutex();

   void      lock();
   void      unlock();
   int       trylock();
};
```

Description

The synchronization process consists of locking and unlocking `Mutex` objects associated with user-defined objects. VSI recommends that users create a `Mutex` object for each user-defined object that needs to be synchronized between threads. Users are then responsible for locking and unlocking the `Mutex` object to coordinate access to the associated object.

To do the locking and unlocking, you can use the `lock` and `unlock` member functions (see Example). Alternatively, if a user-defined object is derived from the `istream` or `ostream` classes, you can use

the `lock` and `unlock` parameterized manipulators, where the parameter is the `Mutex` object (see the Global Declarations section in Chapter 4).

Constructors and Destructors

`Mutex()`

Constructs a `Mutex` object, in effect creating but not locking a recursive mutex.

`~Mutex()`

Deletes a `Mutex` object.

Member Functions

`void lock()`

Locks a recursive mutex. If the mutex is locked by another thread, the current thread is blocked until the mutex becomes available.

`void unlock()`

Unlocks a recursive mutex.

`int trylock()`

Immediately returns to the caller a value of 0 if the mutex is already locked by another thread. Otherwise, this function locks the mutex and returns a value of 1.

Example

```
#include <string.hxx>
#include <mutex.hxx>
⋮
String string1;
Mutex string1_lock;

string1_lock.lock();
string1 = "Hello, ";
string1 += "how are you?";
cout << string1;
string1_lock.unlock();
```

This example synchronizes a sequence of operations on a `String` object, using the `lock()` and `unlock()` member functions.

Chapter 7. Objection Package

The Objection package provides a way to implement simple error handling. You can use this package to catch run-time errors encountered in using classes, and to change or restore actions associated with such errors.

Global Declaration

Global Declaration — This `typedef` is used by, but is not a member of, the `Objection` class.

Header

```
#include <objection.hxx>
```

Alternative Header

```
#include <Objection.h>
```

Declaration

```
typedef int Objection_action(const char*);
```

Type

`Objection_action`

Is the type of an action routine that can be called by the function `Objection::raise`.

Objection class

Objection class — Provides the capability to handle and report errors.

Header

Alternative Header

```
#include <Objection.h>
```

Declaration

```
class Objection
{

public:
                        Objection();
                        Objection(Objection_action *);
    int                 raise(const char * = "");
    Objection_action    *appoint(Objection_action *);
```

```
    Objection_action  *appoint();
    Objection_action  *ignore();
};
```

Description

This class provides ways to handle objections. An **objection** is a potential error condition that your program can encounter. The user appoints an error-handling function. An `Objection` object's `raise()` function invokes the appointed function by passing it a character string that contains an error message. At any point in your program, you can appoint a new error-handling function, reappoint the original function, or specify that an objection be ignored.

Constructors

`Objection()`

Constructs an `Objection` object with no default action (error handler).

`Objection(Objection_action *new_action)`

Constructs an `Objection` object with a pointer to the default error handler. The handler is a function that takes one parameter of type `const char *msg` and returns an `int`. See the `raise()` member function for more information.

Member Functions

`Objection_action *appoint()`

Specifies that the handler for the objection is the default error handler (if one exists) and returns the previous action associated with the specified objection. Specifies that the objection not be ignored.

`Objection_action *appoint(Objection_action *new_action)`

Specifies a new handler for the objection and returns the previous action associated with the specified objection. Specifies that the objection not be ignored.

`Objection_action *ignore()`

Specifies that the objection be ignored (no error handler is invoked if the objection is raised). This function returns the previous action associated with the specified objection.

`int raise(const char *msg = "")`

Raises a specified objection, passing a string (error message) to an error handler (if one exists). If no handler exists, or if the handler returns a 0, the default handler is called. The `raise` function returns the value returned by the last handler it called.

If no default handler exists, then the function returns 0. A 0 is also returned if the objection is ignored. Generally, the return of a nonzero value means that the error handling succeeded, and the return of a 0 value means the error handling failed.

The following example changes the default error handler for the `stack(int)::overflow_error` objection:

```
#include <stdlib.h>
#include <vector.hxx>
#include <objection.hxx>

vectordeclare(int)
stackdeclare(int)

vectorimplement(int)
stackimplement(int)

stack(int) s(10);

int error(const char *errmsg)
{
    cerr << "ERROR TRAPPED: " << errmsg << " - ABORTING\n";
    cerr.flush();
    abort();
    return 0;
}

void main()
{
    Objection_action *save_action;
    save_action = stack(int)::overflow_error.appoint(error);
    for(int i=0; i<100; i++)  //push too many things onto stack
        s.push(i);
    stack(int)::overflow_error.appoint(save_action);
}
```

When this example executes, the following message prints out:

```
ERROR TRAPPED: Stack underflow - ABORTING
%SYSTEM-F-OPCCUS, opcode reserved to customer fault at PC=00010BE5,
PSL=03C00000
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name        line       rel PC    abs PC

                                               0000012D  00010BE5
                                               0000000E  00009346
OBJECTION_EXAMP error               5984       00000045  00003D29
CXXL_OBJECTION  Objection::raise     779       00000026  00008F5A
OBJECTION_EXAMP main                5993       0000005B  00003D87
                                               00000072  0002DB5E
```

Note

The message printed on your system differs somewhat from that shown here.

Chapter 8. Stopwatch Package

The Stopwatch package provides ways to measure intervals of program execution time. The package consists of a single class, `Stopwatch`. Typically, you use this class during the performance-tuning phase of program development.

Stopwatch class

Stopwatch class — Provides the means to measure intervals of time between specified program events.

Header

```
#include <stopwatch.hxx>
```

Alternative Header

```
#include <Stopwatch.h>
```

Declaration

```
class Stopwatch
{
public:
                      Stopwatch();

   void               start();
   void               stop();
   void               reset();
   int                status() const;
   double             system() const;
   double             user() const;
   double             real() const;

   static double      resolution();
};
```

Description

Objects of this class measure program execution time and return the result in floating-point seconds. The class includes the `start`, `stop`, and `reset` functions familiar to users of a mechanical stopwatch.

You can time the entire program or select certain portions of the program to time; for example, a specified loop or program module. You can create a different `Stopwatch` object for each independent program activity, and name each according to the activity you intend to measure.

Constructor

`Stopwatch()`

Constructs a `Stopwatch` object with both time and running status initialized to 0.

Member Functions

double real() const

Returns real time (clock time) in double-precision, floating-point seconds. You can call this function while the stopwatch is running.

void reset()

Resets the current time measurement to 0 without affecting the value of `status()`. If `status()` is initially nonzero, time measurement continues uninterrupted after resetting.

double resolution()

Returns the (system dependent) resolution of measured time in double-precision, floating-point seconds.

void start()

Begins measuring program execution time when `status()` is initially 0 (`status()` becomes nonzero as a consequence of the call). If `status()`is initially nonzero, the call has no effect.

int status() const

Indicates whether the stopwatch is running (returns a value of 1) or not running (returns a value of 0).

void stop()

Halts measurement of program execution time when `status()` is initially nonzero (`status()` becomes 0 as a consequence of the call). If `status()`is initially 0, the call has no effect.

double system() const

Returns the system CPU time in double-precision, floating-point seconds. You can call this function while the stopwatch is running.

double user() const

Returns the user CPU time in double-precision, floating-point seconds. You can call this function while the stopwatch is running.

System Environment

On OpenVMS systems, user time returns the total accumulated CPU time, and system time returns 0. Resolution is 1/100 second.

Example

```
Stopwatch w ;
w.start() ;
//…
// some computation you want to time goes here
//…
w.stop() ;
cout << "elapsed time was " << w.user() << "\n";
```

Displays the number of seconds the computation takes to run. The result is a double-precision value.

Chapter 9. String Package

The String package consists of the single class `String`. This class provides ways to assign, concatenate, and compare character strings. This class also provides methods for substring creation and for vector access to a character string.

String class

String class — Provides the capabilities for manipulating sequences of characters.

Header

```
#include <string.hxx>
```

Alternative Header

None.

Declaration

```
class String
{
    friend ostream  &operator<<(ostream &, const String &);
    friend istream  &operator>>(istream &, String &);
    friend int      operator==(const String &, const String &);
    friend int      operator==(const String &, const char *);
    friend int      operator==(const char *, const String &);
    friend int      operator!=(const String &, const String &);
    friend int      operator!=(const String &, const char *);
    friend int      operator!=(const char *, const String &);
    friend int      operator<(const String &, const String &);
    friend int      operator<(const String &, const char *);
    friend int      operator<(const char *, const String &);
    friend int      operator>(const String &, const String &);
    friend int      operator>(const String &, const char *);
    friend int      operator>(const char *, const String &);
    friend int      operator<=(const String &, const String &);
    friend int      operator<=(const String &, const char *);
    friend int      operator<=(const char *, const String &);
    friend int      operator>=(const String &, const String &);
    friend int      operator>=(const String &, const char *);
    friend int      operator>=(const char *, const String &);
    friend String   operator+(const String &, const String &);
    friend String   operator+(const String &, const char *);
    friend String   operator+(const char *, const String &);

public:
                    String();
                    String(const String &);
                    String(const char *);
                    String(const char &);
                    ~String();
```

```
    String          &operator=(const String &);
    String          &operator=(const char *);
                    operator char * () const;
                    operator const char * () const;
    String          &operator+=(const String &);
    String          &operator+=(const char *);
    String          operator()(int, int) const;
    unsigned int    length() const;
    String          upper() const;
    String          lower() const;
    int             match(const String &) const;
    int             index(const String &) const;
    char            operator[](int) const;
    char            &operator[](int);
};
```

Description

This class provides the means for manipulating sequences of characters, each of which is of the type `char`. For some applications, the services provided are like those provided by the traditional C string library (`strcpy`, `strcmp`, and so forth), but are more efficient and convenient in the context of VSI C++. Overloaded operators provide ways to assign, concatenate, and compare strings. New operators provide simple notations for substring creation and vector access into the string.

All comparisons are lexicographic, with the ordering dependent on the character set in which the string is encoded.

An index value of 0 indicates the first character in a `string` object.

Constructors and Destructors

`String()`

Constructs a `String` object initialized to an empty string.

`String(const char *s)`

Constructs a `String` object and initializes it to the null-terminated sequence of characters.

`String(const char &c)`

Constructs a `String` object with a reference to a `char` datum to initialize the string.

`String(const String &x)`

Constructs a `String` object with a reference to another `String` to initialize the first `String`.

`~String()`

Deletes a `String` object; no user action is required.

Overloaded Operators

`String operator + (const char *s, const String &x)`

Concatenates a null-terminated sequence of characters to a `String` object.

`String operator + (const String &x, const char *s)`

Concatenates a `String` object with a null-terminated sequence of characters.

`String operator + (const String &x, const String &y)`

Concatenates a `String` object with another `String` object.

`String &operator = (const char *s)`

Assigns a `String` object to a null-terminated sequence of characters.

`String &operator = (const String &x)`

Assigns a `String` object to another `String` object.

`int operator < (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is less than a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator < (const String &x, const char *s)`

Tests if a `String` object is less than a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator < (const String &x, const String &y)`

Compares two `String` objects to determine if the first is less than the second; if so, it returns 1. Otherwise, it returns 0.

`int operator > (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is greater than a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator > (const String &x, const char *s)`

Tests if a `String` object is greater than a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator > (const String &x, const String &y)`

Compares two `String` objects to determine if the first is greater than the second; if so, it returns 1. Otherwise, it returns 0.

`String &operator += (const char *st2)`

Concatenates a null-terminated sequence of characters to a `String` object.

`String &operator += (const String &st2)`

Concatenates a `String` object to another `String` object.

`ostream &operator << (ostream &s, const String &x)`

Inserts the sequence of characters represented by *x* into the stream *s*.

`istream &operator >> (istream &s, String &x)`

Extracts characters from *s* using the istream extraction operator, then stores characters in *x*, replacing the current contents of *x* and dynamically allocating *x* as necessary.

`int operator == (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is equal to a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator == (const String &x, const char *s)`

Tests if a `String` object is equal to a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator == (const String &x, const String &y)`

Compares two `String` objects to determine equality. If one is equal to the other, it returns 1; otherwise, it returns 0.

`int operator != (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is not equal to a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator != (const String &x, const char *s)`

Tests if a `String` object is not equal to a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator != (const String &x, const String &y)`

Compares two `String` objects to determine inequality. If they are not equal, the function returns 1; otherwise, it returns 0.

`int operator <= (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is less than or equal to a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator <= (const String &x, const char *s)`

Tests if a `String` object is less than or equal to a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator <= (const String &x, const String &y)`

Compares two `String` objects to determine if the first is less than or equal to the second; if so, it returns 1. Otherwise, it returns 0.

`int operator >= (const char *s, const String &x)`

Tests if a null-terminated sequence of characters is equal to or greater than a `String` object; if so, it returns 1. Otherwise, it returns 0.

`int operator >= (const String &x, const char *s)`

Tests if a `String` object is equal to or greater than a null-terminated sequence of characters; if so, it returns 1. Otherwise, it returns 0.

`int operator >= (const String &x, const String &y)`

Compares two `String` objects to determine if the first is equal to or greater than the second; if so, it returns 1. Otherwise, it returns 0.

`String operator () (int index, int count) const`

Creates a new `String` object defined as a substring of the current `String`, with *index* as the starting character and *count* as the length of the substring.

`char operator [] (int position) const`

Returns the character at the requested position within the string. If the position is past the end of the string, it returns 0. If the position is negative, the results are undefined.

`char &operator [] (int position)`

Returns a reference to the character at the requested position within the string. This reference is potentially invalid after any subsequent call to a non-const member function for the object. If the position is past the end of the string or if the position is negative, the results are undefined.

Other Member Functions

`int index(const String &x) const`

Returns the index value of the first position where an element of a `String` object coincides with the value of *x*.

`unsigned int length() const`

Returns the length (number of characters) in a `String` object.

`String lower() const`

Returns a new `String` object constructed from a `String` except that every character is lowercase regardless of its original case.

`int match(const String &x) const`

Compares two strings and returns the first index position at which they differ; it returns –1 if the strings match completely. The `String` argument can be a character pointer.

`String upper() const`

Returns a new `String` constructed from a `String` except that every character is uppercase regardless of its original case.

Examples

1.
```
String x ("The Times of John Doe");
char *y = "Pink Triangles";

if (x != y) cout << "We have two different strings.\n";
```

```
x = y;

cout << x;
```

The first line of this example provides a character string to the constructor for initialization. The overloaded operators (!=, <<, and =) accept either two `String` objects or a `String` and a null-terminate sequence of characters. The last line prints out the following character string:

```
Pink Triangles
```

2.
```
String x ("The Times of John Doe");

String a (x(18,3));     // Substring is "Doe"
String b (x);           // b contains all of x
```

In this example, the creation of object `a` provides a substring of object `x` to the constructor for object `a`. The substring begins at position 18 and has a length of 3 characters. The next line creates the object `b` and initializes it to contain the same value as `x`.

3.
```
String x ("World");
String y;

y = "Hello";
y += ", " + x + ".\n";

cout << y;
```

This example shows string concatenation. The last line prints out the following message:

```
Hello, World.
```

Chapter 10. task Package

Note

The task package is not supported on the Linux Alpha platform.

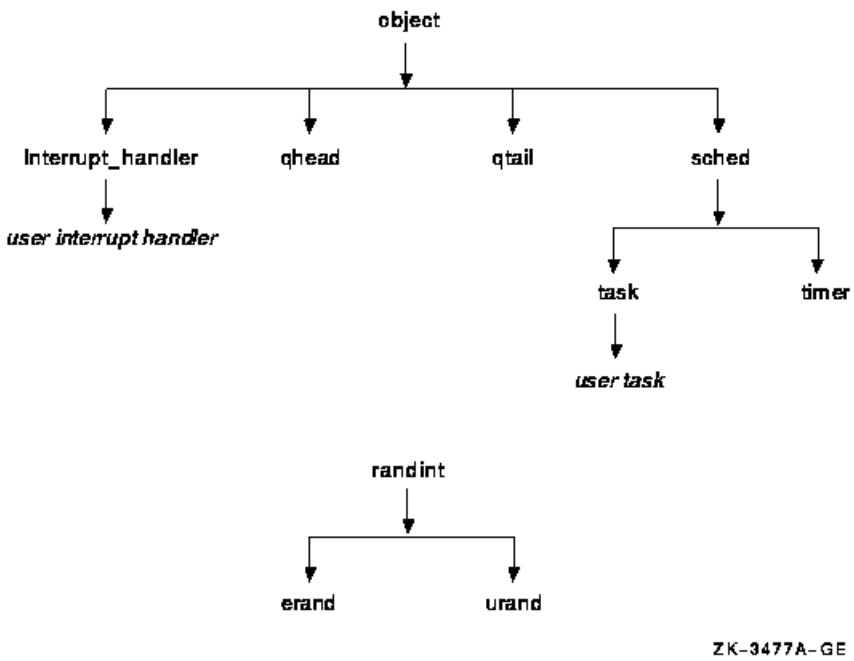
The task package provides coroutine support. A **coroutine**, or **task**, is a subroutine that can suspend execution to allow other tasks to run. Static data is shared among all tasks; automatic and register data is allocated separately for each task. Only one task can execute at a time, even on a multiprocessor system.

Programming with tasks can be particularly appropriate for simulations or other applications that can be reasonably represented as sets of concurrent activities.

This package includes the `object` and `randint` classes, the subclasses derived from these classes, and the `histogram` class.

Figure 10.1 shows the inheritance structure of the task package.

Figure 10.1. Inheritance Diagram for the task Package



Also note the following:

- The `sched` and `task` classes are intended for use only as base classes.
- The task package makes use of the threads library.
- The task package is not thread safe. You cannot create tasks simultaneously from different threads.

Global Declarations

Global Declarations — The `typedef`, `enum`, and `extern` declarations are used by one or more classes in the task package but they are not members of any particular class.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
typedef int(*PFIO)(int, object*);
typedef void(*PFV)();

enum
{
    VERBOSE = 1 << 0,
    CHAIN = 1 << 1,
    STACK = 1 << 2,
};

enum qmodetype
{
    EMODE,
    WMODE,
    ZMODE
};

enum
{
    E_OLINK = 1,
    E_ONEXT = 2,
    E_GETEMPTY = 3,
    E_PUTOBJ = 4,
    E_PUTFULL = 5,
    E_BACKOBJ = 6,
    E_BACKFULL = 7,
    E_SETCLOCK = 8,
    E_CLOCKIDLE = 9,
    E_RESTERM = 10,
    E_RESRUN = 11,
    E_NEGTIME = 12,
    E_RESOBJ = 13,
    E_HISTO = 14,
    E_STACK = 15,
    E_STORE = 16,
    E_TASKMODE = 17,
    E_TASKDEL = 18,
    E_TASKPRE = 19,
    E_TIMERDEL = 20,
    E_SCHTIME = 21,
```

```
    E_SCHOBJ = 22,
    E_QDEL = 23,
    E_RESULT = 24,
    E_WAIT = 25,
    E_FUNCS = 26,
    E_FRAMES = 27,
    E_REGMASK = 28,
    E_FUDGE_SIZE = 29,
    E_NO_HNDLR = 30,
    E_BADSIG = 31,
    E_LOSTHNDLR = 32,
    E_TASKNAMEOVERRUN = 33
};

extern int _hwm;
```

Types

enum Print Function Arguments

The *verbosity* argument to `print` member functions uses the following values:

Value	Explanation
`0`	Requests a brief report
`CHAIN`	Requests information about tasks on the object's remember chain, and about other objects on the object's `o_next` chain
`STACK`	Requests information about the run-time stack
`VERBOSE`	Requests detailed information on the class object

To combine several requests, use the bitwise inclusive operator (|). For example:

```
p->print(VERBOSE|CHAIN);
```

enum qmodetype

The following values are used by the `qhead` and `qtail` classes for managing queues:

Value	Explanation
`EMODE`	Generates a run-time error if full on enqueue or empty on dequeue
`WMODE`	Suspends task execution if full on enqueue or empty on dequeue
`ZMODE`	Returns NULL if full on enqueue or empty on dequeue

enum Exception Codes

Descriptions of the `E_` codes are given in the Exception Handling sections of the appropriate classes.

`PFIO`

Is a pointer to a function returning `int`, which takes arguments of the types `int` and `object *`.

PFV

Is a pointer to a function returning `void`, which takes no arguments.

Other Data

extern int _hwm

Can be set to a nonzero value before creation of the first task to keep track of the maximum stack size ("high water mark"). The maximum stack size can be printed by the `task::print()` function.

erand class

erand class — Objects of the `erand` class are generators of exponentially distributed random numbers.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class erand: public randint
{
public:
    int mean;

        erand(int m);

    int draw();
};
```

Member Data

int mean

Is the mean of the generated random numbers.

Constructor

erand(int m)

Constructs an `erand` object with *m* as the mean for the generated random numbers.

Member Function

int draw()

Returns the next random integer generated by the object.

See Also

randint class

histogram class

histogram class — Objects of the `histogram` class are generators of histograms.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class histogram
{
public:
    int   l;
    int   r;
    int   binsize;
    int   nbin;
    int   *h;
    long  sum;
    long  sqsum;

          histogram(int n_bins = 16, int left = 0, int right = 16);
          ~histogram();

    void  add(int sample);
    void  print();
};
```

Description

Objects of this class generate histograms. Each such object has `nbin` bins, spanning a range from `l` to `r`.

Exception Handling

When a run-time error occurs, the following error code is passed to the `object::task_error()` function:

Value	Error Description
`E_HISTO`	Cannot construct a histogram with less than 1 bucket or the left not less than the right

Member Data

`int binsize`

Is the size of the range covered by an individual bin.

`int *h`

Is a pointer to a vector of `nbin` integers. Each element of the vector is the number of samples placed into that bin by the `add()` function.

`int l`

Is the lower (left) end of the range of samples.

`int nbin`

Is the total number of bins.

`int r`

Is the higher (right) end of the range of samples.

`long sqsum`

Is the sum of the squares of the integers added to a bin by the `add()` function.

`long sum`

Is the sum of the integers added to a bin by the `add()` function.

Constructors and Destructors

`histogram(int n_bins = 16, int left = 0, int right = 16)`

Constructs a `histogram` object. The arguments are all optional: *n_bins* specifies the number of bins, *left* specifies the initial left end of the range and *right* specifies the initial right end of the range. At instantiation, the member data are initialized as follows:

The count in each bin is set to 0.
The value of `l` is *left*
the value of `r` is *right*
`nbin` is set to *n_bins*
The values of `sqsum` and `sum` are 0.

`~histogram()`

Deletes a `histogram` object.

Member Functions

`void add(int sample)`

Adds one to the bin specified by *sample*. If *sample* is outside the range of `l` to `r`, the range expands by either decreasing `l` or increasing `r`; however, `nbin` remains constant. Thus, the range covered by one bin doubles if the total histogram doubles.

void print()

Prints on `cout` the number of entries for each nonempty bin.

Interrupt_handler class

Interrupt_handler class — Interrupt handlers let tasks wait for external events (system signals), and allow the declaration of handler functions for these events.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class Interrupt_handler: public object
{
public:
                     Interrupt_handler(int);
                     ~Interrupt_handler();

    virtual void     print(int verbosity, int internal_use = 0);
    virtual int      pending();
    virtual objtype  o_type();

private:
    virtual void interrupt();
};
```

Description

Interrupt handlers allow tasks to wait for signals. You can use classes derived from the `Interrupt_handler` class to overload the `interrupt()` function. When the signal is raised, the task package immediately calls the `interrupt()` function. The task package then schedules its own internal interrupt alerter task for execution. Control returns to the task (if any) that was running when the signal was raised. When control returns to the scheduler, the interrupt alerter runs and schedules for execution those tasks that were waiting for the interrupt handler.

If the run chain (see the `sched` class) is empty, the scheduler does not cause the program to exit if there are any interrupt handlers that have been created but not yet destroyed.

If an `interrupt()` function is not needed, you can use the `Interrupt_handler` class without deriving another class from it.

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
E_NO_HNDLR	Cannot handle a signal for which there is no handler
E_BADSIG	Cannot handle a signal with an invalid signal number
E_LOSTHNDLR	Cannot delete an Interrupt_handler that is not on the stack of them for the given signal

Constructors and Destructors

Interrupt_handler(int signal_to_catch)

Constructs a new `Interrupt_handler` object that waits for a specified signal.

~Interrupt_handler()

Deletes an `Interrupt_handler` object.

Member Functions

virtual void interrupt()

Does nothing but lets classes derived from the `Interrupt_handler` class overload this function to specify actions. Because it is private, you cannot call it directly.

virtual objtype o_type()

Returns `object::INTHANDLER`.

virtual int pending()

Returns 0 on the first call after the signal is raised; otherwise, it returns a nonzero value.

virtual void print(int verbosity, int internal_use = 0)

Prints information about the interrupt handler. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

System Environment

The thread system exception handling uses OpenVMS conditions and does not interact directly with signals.

Example

```
extern "C" {
#include <stdlib.h>
}
#include <signal.h>
#include <task.hxx>
#include <iostream.hxx>

class floating_exception: public Interrupt_handler
{
    virtual void interrupt();
public:
```

```
    floating_exception(): Interrupt_handler(SIGFPE) {};
};

void floating_exception::interrupt()
{
    cout << "In floating_exception::interrupt -
    Floating exception caught!\n";
    cout.flush();
}

int main()
{
    floating_exception sigfpe_handler;
    raise(SIGFPE);
    return EXIT_SUCCESS;
}
```

This example prints out the following message:

```
In floating_exception::interrupt - Floating exception caught!
```

object class

object class — Base class for other classes in the task package and for user-defined classes of objects to be placed in queues (see the `qhead class` and qtail class classes).

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class object
{
public:
    enum objtype
    {
        OBJECT,         // class object
        TIMER,          // class timer
        TASK,           // class task
        QHEAD,          // class qhead
        QTAIL,          // class qtail
        INTHANDLER      // class Interrupt_handler
    };

    object  *o_next;

    static  PFIO error_fct;

            object();
            ~object();
```

```
    void    alert();
    void    forget(task *p_task_to_forget);
    void    remember(task *p_task);
    int     task_error(int error_code);

    virtual objtype o_type();
    virtual int pending();
    virtual void print(int verbosity, int internal_use = 0);

    static  int task_error(int error_code, object *object_with_problem);
    static  task *this_task();
};
```

Description

This class is a base class for many other classes within the task package. You also can use it to derive user classes to be placed in the task package's queues and so forth. All objects derived from the `object` class can declare the virtual function `object::pending()`, which the scheduler uses to determine if an object is ready or not ready. You can provide each kind of `object` with its own method of determining its state of readiness. Each pending `object` contains a list (the remember chain) of the waiting `task` objects.

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
`E_OLINK`	Cannot delete an object with a remembered task
`E_ONEXT`	Cannot delete an object that is on a list
`E_STORE`	Cannot allocate more memory

Member Data

`PFIO error_fct`

Points to a function to be called by the `task_error` function. For more information, see the `task_error` function.

`object *o_next`

Points to the next object in the queue or run chain.

Constructors and Destructors

`object()`

Constructs an `object` object.

`~object()`

Deletes an `object` object.

Member Functions

void alert()

Changes the state of all `task` objects remembered by the `object` from `IDLE` to `RUNNING`, puts the `task` objects on the scheduler's run chain, and removes the `task` objects from the remembering object's remember chain. You must call the `object::alert` function for the `object` when the state of an `object` changes from pending to ready.

void forget(task *p_task_to_forget)

Removes, from the remembering `object` object's remember chain, all occurrences of the `task`, denoted by the *p_task_to_forget* argument.

virtual objtype o_type()

Returns `object::OBJECT`.

virtual int pending()

Always returns a nonzero value.

In classes derived from `object`, `pending()` returns the ready status of an object: 0 if an `object` object is ready and a nonzero value if the `object` object is pending. Classes derived from the `object` class must define `pending()` if waiting is instituted. By default, `object::pending` returns a nonzero value.

virtual void print(int verbosity, int internal_use = 0)

Prints an object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

void remember(task *p_task)

Puts a `task` for a pending `object` on the remember chain and suspends the `task`, when that `task` attempts an operation on the pending `object`. Remembered `task` objects are alerted when an object of the `object` class becomes ready.

int task_error(int error_code)

Is obsolete. Calling `p->task_error(e)` is equivalent to calling `object::task_error(e,p)`.

static int task_error(int error_code, object *object_with_problem)

Called when a run-time error occurs. The *error_code* argument represents the error number and the *object_with_problem* argument represents a pointer to the `object` that called `task_error()`. The `object::task_error()` function examines the variable `error_fct` and calls this function if it is not `NULL`. If the function returns 0, `task_error()` returns to its caller, which may retry the operation. (An infinite loop may result if no appropriate recovery is made.) If the function returns a nonzero value, `task_error()` calls `exit(error_code)`. Otherwise, `task_error()` gives the error number as an argument to `print_error()`, which prints an error message on `cout` and `task_error()` calls `exit(error_code)`.

The *object_with_problem* argument may be `NULL` if no particular object can be associated with the error.

static task *this_task()

Returns a pointer to the `task` object currently running.

qhead class

qhead class — Abstraction for the head of a list of items arranged in a first-in, first-out singly linked list.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class qhead: public object
{
public:
              qhead(qmodetype modetype = WMODE, int size = 10000);
              ~qhead();

    qhead     *cut();
    object    *get();
    int       putback(object *new_queue_element);
    int       rdcount();
    int       rdmax();
    qmodetype rdmode();
    void      setmode(qmodetype modetype);
    void      setmax(int size);
    void      splice(qtail *delete_tail);
    qtail     *tail();

    int       pending();
    void      print(int verbosity, int internal_use = 0);
    objtype   o_type();
};
```

Description

This class provides facilities for taking objects off a queue. A queue is a data structure with an associated list of objects of the `object` class, or a class derived from the `object` class in first-in, first-out order. All access to a queue is through either the attached `qhead` or attached `qtail` object. You create a queue by creating either a `qhead` or a `qtail` object. The other end of the queue is created automatically. You can then obtain a pointer to the tail with the `qhead::tail` function.

Objects have definitions for when they are ready and pending (not ready). The `qhead` objects are ready when the queue is not empty and pending when the queue is empty.

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
E_BACKFULL	Cannot putback an object into a full queue
E_BACKOBJ	Cannot putback an object into a queue if the object is on another queue
E_GETEMPTY	Cannot get an object from an empty queue
E_QDEL	Cannot delete a queue that has an object in the queue
E_STORE	Cannot allocate more memory

Constructors and Destructors

qhead(qmodetype modetype = WMODE, int size = 10000)

Constructs a qhead object. The *modetype* argument determines what happens when an object of the qhead class is pending. The choices are WMODE (wait mode), EMODE (error mode), or ZMODE (0 mode); the default is WMODE (see the get() function for more information). The *size* argument sets the maximum length of the queue attached to a qhead object; the default is 10,000.

The maximum size of the queue does not affect the amount of memory occupied by the queue when the queue is empty.

~qhead()

Deletes a qhead object.

Member Functions

qhead *cut()

Splits a queue into two queues. One queue has a new qhead object, which the return value points to, and the original qtail object; it contains the objects from the original queue. The other queue has the original qhead object and a new qtail object; this queue is empty. You can use this function to insert a filter into an existing queue without changing the queue's appearance to functions that access the ends of the queue, and without halting the flow through the queue of objects.

object *get()

Returns a pointer to the object at the head of the queue when the queue is not empty. The object is removed from the queue. If the queue is empty, behavior depends on the mode of the qhead object. In WMODE, a task that executes qhead::get() on an empty queue suspends until that queue is not empty. In EMODE, executing qhead::get() on an empty queue causes a run-time error. In WMODE, executing qhead::get() on an empty queue returns the NULL pointer instead of a pointer to an object.

virtual objtype o_type()

Returns object::QHEAD.

int pending()

Specifies that get operations on a queue must wait until an object is put in the queue. It returns a nonzero value if the queue attached to a qhead object is empty; otherwise, it returns 0.

void print (int verbosity, int internal_use = 0)

Prints a `qhead` object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

`int putback(object *new_queue_element)`

Inserts at the head of the queue the `object` that the *new_queue_element* argument points to, and returns a value of 1 on success. This lets the `qhead` object operate as a stack (hence, the name `putback`). Space must be available in the queue for it to succeed. Calling `qhead::putback()` for a full queue causes a run-time error in both `EMODE` and `WMODE` and returns `NULL` in `ZMODE`.

`int rdcount()`

Returns the current number of objects in the queue attached to a `qhead` object.

`int rdmax()`

Returns the maximum length of the queue.

`qmodetype rdmode()`

Returns the current mode of a `qhead` object, which can be `EMODE`, `WMODE`, or `ZMODE`.

`void setmode(qmodetype modetype)`

Sets the mode of a `qhead` object to *modetype*, which can be `EMODE`, `WMODE`, or `ZMODE`.

`void setmax(int size)`

Sets *size* as the maximum length of the queue attached to a `qhead` object. You can set *size* to a number less than the current number of objects of the `object` class, but that means you cannot put any more objects of the `object` class on the queue until the length of the queue has been reduced below the limit you set.

`void splice(qtail *delete_tail)`

Forms a single queue by appending a queue attached to a `qhead` object onto the queue referenced in the argument. Typically, this reverses the action of a previous `qhead::cut()` function. The extra `qhead` and `qtail` objects are deleted. Waiting tasks resume execution if merging the two creates a nonempty queue (if the task was trying to get) or an empty queue (if the task was trying to put).

`qtail *tail()`

Creates a `qtail` object for the queue attached to a `qhead` object (if none exists) and returns a pointer to the new `qtail` object.

qtail class

qtail class — Abstraction for the tail of a list of items in a first-in, first-out singly linked list.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class qtail: public object
{
    friend class qhead;

public:
              qtail(qmodetype modetype = WMODE, int size = 10000);
              ~qtail();

    qtail     *cut();
    qhead     *head();
    int       put(object *new_queue_element);
    int       rdspace();
    int       rdmax();
    qmodetype rdmode();
    void      setmode(qmodetype modetype);
    void      setmax(int size);
    void      splice(qhead *delete_head);

    int       pending();
    void      print(int verbosity, int internal_use = 0);
    objtype   o_type();
};
```

Description

This class provides facilities for putting objects into a queue. A queue is a data structure with an associated list of objects of the `object` class, or a class derived from the `object` class in first-in, first-out order. All access to a queue is through either the attached `qhead` or `qtail` object. You create a queue by creating either a `qhead` or a `qtail` object. The other end of the queue is created automatically. You can then obtain a pointer to the head with the `qtail::head` function.

Objects have definitions for when they are ready and pending (not ready). The `qtail` objects are ready when the queue is not full and pending when the queue is full.

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
`E_PUTFULL`	Cannot put an object into a full queue
`E_PUTOBJ`	Cannot put an object into queue if the object is on another queue
`E_QDEL`	Cannot delete a queue that has an object in the queue
`E_STORE`	Cannot allocate more memory

Constructors and Destructors

`qtail(qmodetype modetype = WMODE, int size = 10000)`

Constructs a `qtail` object. The *modetype* argument specifies the mode (set by the constructor) that controls what happens when an object of the `qtail` class is pending. The choices are `WMODE` (wait

mode), `EMODE` (error mode), or `ZMODE` (0 mode); `WMODE` is the default. (See the `put()` function for more information.) The *size* argument specifies the maximum length of the queue attached to a `qhead` object; the default is 10,000.

The maximum size of the queue does not affect the amount of memory occupied by the queue when the queue is empty.

~qtail()

Deletes a `qtail` object.

Member Functions

qtail *cut()

Splits a queue into two queues. One queue has a new `qtail` object (to which the return value points) and the original `qhead` object; it contains the objects from the original queue. The other queue has the original `qtail` object and a new `qhead` object; this queue is empty. You can use this function to insert a filter into an existing queue, without changing the queue's appearance to functions that access the ends of the queue, and without halting the flow through the queue of objects.

qhead *head()

Creates a `qhead` object for the queue attached to a `qtail` object (if none exists) and returns a pointer to the new `qhead` object.

virtual objtype o_type()

Returns `object::QTAIL`.

int pending()

Specifies that get operations on a queue must wait until an object is put in the queue. It returns a nonzero value if the queue is empty; otherwise, it returns 0.

virtual void print(int verbosity, int internal_use = 0)

Prints a `qtail` object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

int put(object *new_queue_element)

Adds the `object` denoted by the *new_queue_element* argument to the tail of the queue attached to a `qtail` object; returns a value of 1 on success. If the queue is full, the behavior depends on the mode of the `qtail` object. In `WMODE`, an object of class `task` that executes `qhead::put()` on a full queue suspends until that queue is not full. Calling `qhead::put()` for a full queue causes a run-time error in `EMODE` and returns `NULL` in `ZMODE`.

int rdspace()

Returns the number of `object` objects that can be inserted into the queue before it becomes full.

int rdmax()

Returns the maximum length of the queue.

qmodetype rdmode()

Returns the current mode of a `qtail` object, which can be `EMODE`, `WMODE`, or `ZMODE`.

`void setmode(qmodetype modetype)`

Sets the mode of a `qtail` object to *modetype*, which can be `EMODE`, `WMODE`, or `ZMODE`.

`void setmax(int size)`

Sets *size* as the maximum length of the queue. You can set *size* to a number less than the current number of objects of the `object` class, but that means you cannot put any more objects of the `object` class on the queue until the length of the queue has been reduced below the limit you set.

`void splice(qhead *delete_head)`

Forms a single queue by appending a queue attached to a `qtail` onto the queue referenced in the argument. Typically, this reverses the action of a previous `qtail::cut()`. The extra `qhead` and `qtail` objects are deleted. Waiting tasks resume execution if merging the two queues creates a nonempty queue (if the task was trying to get) or an empty queue (if the task was trying to put).

randint class

randint class — Objects of the `randint` class generate uniformly distributed random numbers.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class randint
{
public:
           randint(long seed=0);
    int    draw();
    float  fdraw();
    void   seed(long seed);
};
```

Description

Objects of this class generate uniformly distributed random numbers. Each random-number generator object produces a sequence that is independent of other random-number generator objects.

Constructor

`randint(long seed)`

Constructs an object of the `randint` class. The *seed* argument is used as the seed and is optional. Different seeds produce different sequences of generated numbers; not all seeds produce useful sequences.

Member Functions

`float fdraw()`

Returns the next random number generated by the object. The number is a floating-point value in the range 0 to 1.

`int draw()`

Returns the next random number generated by the object. The number is an integer value in the range from 0 to `RAND_MAX`, which is defined in the ANSI C header, `stdlib.h`.

`void seed(long seed)`

Reinitializes the object with the seed *seed*.

Example

```
extern "C" {
#include <stdlib.h>
}
#include <task.hxx>
#include <iostream.hxx>
main()
 {
     randint gen;
     int i=0;
     float sum;
     for (i=0; i<1000; i++)
          sum += gen.fdraw();
     cout<<"Average is " << sum/1000. << "\n";
     return EXIT_SUCCESS;
 }
```

This example prints the average of 1000 floating-point random numbers.

sched class

sched class — Responsible for scheduling and for the functionality common to `task` and `timer` objects.

Header

`#include <task.hxx>`

Alternative Header

`#include <task.h>`

Declaration

```
class sched: public object
{
```

```
public:
    enum statetype
    {
        IDLE = 1,
        RUNNING = 2,
        TERMINATED = 4
    };

protected:
                sched();

public:
    static task  *clock_task;
    static PFV   exit_fct;

    void         cancel(int result);
    int          dont_wait();
    sched        *get_priority_sched();
    int          keep_waiting();
    statetype    rdstate();
    long         rdtime();
    int          result();

    int          pending();
    virtual void print(int verbosity, int internal_use = 0);
    virtual void setwho(object *alerter);

    static long  get_clock();
    static sched *get_run_chain();
    static int   get_exit_status();
    static void  set_exit_status(int);
    static void  setclock(long);
};

#ifdef CXXL_DEFINE_CLOCK
#define clock (sched::get_clock())
#endif
#define run_chain (sched::get_run_chain())
```

Description

This class provides facilities for checking on the state of a task, manipulating the simulated clock, canceling a task, and checking on the result of a task.

You can create instances of classes derived from the `sched` class, but you cannot create instances of the `sched` class itself.

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
`E_CLOCKIDLE`	Cannot advance the clock when the clock_task is `RUNNING` or `TERMINATED`

Value	Error Description
E_NEGTIME	Cannot delay a negative amount of time
E_RESOBJ	Cannot resume a task or timer if it is already on another queue
E_RESRUN	Cannot resume a RUNNING task
E_RESTERM	Cannot resume a TERMINATED task
E_SCHOBJ	Cannot use class sched other than as a base class
E_SCHTIME	Cannot execute something at a time that has already passed
E_SETCLOCK	Cannot set the clock after it has advanced past 0

Member Data

static task *clock_task

Points to the task clock if one exists.

static PFV exit_fct

Points to the exit function if one exists.

Constructor

sched()

Constructs a sched object initialized to the IDLE state and delay 0.

Member Functions

void cancel(int result)

Puts an object into the TERMINATED state without suspending the caller (that is, without invoking the scheduler); sets the result of the object to *result*.

int dont_wait()

Returns the number of calls to keep_waiting(), minus the number of calls to the dont_wait() function, excluding the current call. The return value of this function should equal the number of objects of the object class waiting for external events before the current dont_wait() call.

long get_clock()

Returns the value of the clock in simulated time units.

int get_exit_status()

Returns the exit status of the task program. When a task program terminates successfully (without calling task_error), the program calls exit(i) where *i* is the value passed by the last caller of sched::set_exit_status().

sched *get_priority_sched()

Returns a pointer to a system task's interrupt_alerter if the system gets an awaited signal. If no interrupt occurs, this function returns 0.

sched *get_run_chain()

Returns a pointer to the run chain, the linked list of ready objects belonging to classes derived from the `sched` class (`task` and `timer` objects).

int keep_waiting()

Keeps the scheduler from exiting when no tasks exist that can be run (an external event could enable an `IDLE task` to be run). This function should be called when the user program creates an object that waits for an external event. Afterward, when such an object destructs, a call should go to the `dont_wait()` function. The `keep_waiting()` function returns the number of calls (not counting the current call) minus the number of calls to the `dont_wait()` function.

int pending()

Returns 0 if the object is in the `TERMINATED` state; otherwise, it returns a nonzero value.

virtual void print(int verbosity, int internal_use = 0)

Prints a `sched` object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

statetype rdstate()

Returns the state of the object: `RUNNING`, `IDLE`, or `TERMINATED`.

long rdtime()

Returns the simulated clock time at which to run the object.

int result()

Returns the result of a `sched` object (as set by the `task::resultis()`, `task::cancel()`, or `sched::cancel()` function). If the object is not yet `TERMINATED`, the calling `task` suspends and waits for the object to terminate. A `task` calling `result()` for itself causes a run-time error.

void setclock(long new_clock)

Initializes the simulated clock to a time specified by the *new_clock* argument. You can use this function once before the simulated clock has advanced without causing a run-time error. To advance the clock after the initial setting, call the `task::delay` function.

void set_exit_status(int new_exit_status)

Sets the exit status of the task program. When a task program terminates successfully (without calling `task_error`), the program calls `exit(i)`, where *i* is the value passed by the last caller of `sched::set_exit_status()`.

virtual void setwho(object *alerter)

Records which object alerted the object. The *alerter* argument should represent a pointer to the `object` that caused the task package to alert the `sched`.

Macros

The VSI C++ Class Library Class Library supplies the following macros for compatibility with older VSI C++ Class Library implementations:

clock

Calls `sched::get_clock()`. For this macro to be defined, you must define `CXXL_DEFINE_CLOCK` on the command line when invoking the compiler, or in your source code before including the task package header.

run_chain

Calls `sched::get_run_chain()`.

task class

task class — Serves as the basis for coroutines.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class task: public sched
{
public:
    enum modetype
    {
        DEDICATED = 1,
        SHARED = 2
    };
protected:
                    task(char *name = (char *)NULL,
                       modetype mode = DEFAULT_MODE, int stacksize = 0);
public:
    task            *t_next;
    char            *t_name;

                    ~task();

    void            cancel(int);
    void            delay(long);
    long            preempt();
    void            resultis(int);
    void            setwho(object *);
    void            sleep(object *object_waiting_for = (object *)NULL);
    void            wait(object *);
    int             waitlist(object * ...);
    int             waitvec(object **);
    object          *who_alerted_me();

    virtual void    print(int verbosity, int internal_use = 0);
    virtual objtype o_type();
```

```
    static        task *get_task_chain();
};
```

Description

This class is used only as a base class; all coroutine classes are derived from it. All work for an object of a given coroutine type occurs within the constructor for that type. The coroutine class must be exactly one level of derivation from the task class. When the object is created, the constructor takes control and runs until halted by one of the following functions: `wait()`, `waitlist()`, `waitvec()`, `sleep()`, or `resultis()`.

When a `task` executes a blocking function on an `object` that is ready, the operation succeeds immediately and the `task` continues running; if the object is pending, the `task` waits. Control then returns to the scheduler, which selects the next `task` from the ready list or run chain. When a pending object becomes ready, the system puts any `task` waiting for that object back on the run chain.

A `task` can be in one of the following states:

`RUNNING`	Running or ready to run
`IDLE`	Waiting for a pending `object`
`TERMINATED`	Completed; not able to resume running (but you can retrieve the result)

Exception Handling

When a run-time error occurs, the appropriate error code from the following table is passed to the `object::task_error()` function:

Value	Error Description
`E_RESULT`	Cannot call `result()` on `thistask`
`E_STACK`	Cannot extend stack
`E_STORE`	Cannot allocate more memory
`E_TASKDEL`	Cannot delete a task that is `IDLE` or `RUNNING`
`E_TASKMODE`	Cannot create a task with a mode other than `DEDICATED` or `SHARED`
`E_TASKNAMEOVERRUN`	Internal error: data overrun when building default task name
`E_TASKPRE`	Cannot preempt a task that is `IDLE` or `TERMINATED`
`E_WAIT`	Cannot call `wait()` on `thistask`

Member Data

`task *t_next`

Points to the text task on the chain of all `task` objects; it is equal to `NULL` if there are no more tasks.

`char *t_name`

Points to the null-terminated task name passed to the constructor. If no name was passed to the constructor, then the constructor creates a unique name (and *t_name* points to it). If the constructor created the name, then the destructor deletes the name.

Constructors and Destructors

`task(char *name = (char *)NULL, modetype mode = DEFAULT_MODE, int stacksize = 0)`

Constructs a `task` object. All three arguments are optional and have default values. If you supply a character pointer, *name* is used as the `task` object's name. The argument *mode* must be `DEDICATED` or `SHARED` (or omitted) but only `DEDICATED` is implemented; thus, the *mode* argument has no effect. The argument *stacksize* specifies the minimum size of the `task` object's stack. By default, the stack size is the same as the default for the underlying thread system.

Note

With `DEDICATED` stacks, the addresses of parameters to a constructor derived from the `task` class change. This change occurs between the time when the base class (`task`) constructor is called by the derived class constructor and when the first statement in the derived class constructor begins executing.

Constructors for the `task` class and the classes derived from the `task` class cannot be inlined. These classes perform actions that start up a child task (in a new thread) and then resume execution of the parent task.

`~task()`

Deletes an object of the `task` class. It deletes the task name if the constructor created the name.

Member Functions

`void cancel(int result)`

Puts a task object into the `TERMINATED` state without suspending the calling `task` (that is, without invoking the scheduler); sets the result of the object to *result*.

`void delay(long delay)`

Suspends a task object for the time specified by *delay*. A delayed task is in the `RUNNING` state. The task object resumes at the current time on the system clock, plus the time specified by *delay*. Only calling `delay()`, or waiting for a timer, advances the clock.

`task *get_task_chain()`

Returns a pointer to the first task on the list of all task objects linked by `next_t` pointers.

`virtual objtype o_type()`

Returns `object::TASK`.

`long preempt()`

Suspends a `RUNNING` object of the `task` class making it `IDLE`. Returns the number of time units left in the task's delay. Calling this function for an `IDLE` or `TERMINATED` task causes a run-time error.

`virtual void print(int verbosity, int internal_use = 0)`

Prints a task object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

void resultis(int result)

Sets the return value of a `task` object to be the value of *result*; it puts the `task` object in the `TERMINATED` state. To examine the result, call the `sched::result()` function. The constructor for a class derived from `task` must not return by any of the following actions:

- Executing a return statement
- Throwing an exception
- Not catching an exception thrown by a subroutine

The end of a constructor for a class derived from the `task` class and the `main` function must call the `resultis()` function. A task is pending until its stage changes to `TERMINATED`. For more information, see `sched::pending()`.

void setwho(object *alerter)

Keeps track of which object alerted the object. The *alerter* argument should represent a pointer to the `object` that caused the task package to alert the `task`.

void sleep(object *object_waiting_for)

Suspends a `task` object unconditionally (that is, it puts the `task` object in the `IDLE` state). The argument *object_waiting_for* is optional; if it is pointing to a pending object, the object remembers the task. When the object is no longer pending, the task is rescheduled. If you do not supply an argument, the event that causes the `task` object to resume remains unspecified.

void wait(object *object_waiting_for)

Suspends a `task` object (it puts the `task` object in the `IDLE` state) until that `object` is ready, if *object_waiting_for* points to an `object` that is pending. If *object_waiting_for* points to an `object` that is ready (not pending), then `task::wait` does not suspend the `task` object.

int waitlist(object *first_object_waiting_for ...)

Suspends a `task` object to wait for one of a list of objects to become ready. The `waitlist()` function takes a list of `object` pointers linked by `o_next` and terminated by a NULL argument. If any of the arguments point to a ready `object`, then the `task` object is not suspended. When one of the objects pointed to in the argument list is ready, `waitlist()` returns the position in the list of the object that caused the return; position numbering starts at 0.

int waitvec(object **object_waiting_for_vector)

Differs from `waitlist()` only in that `waitvec()` takes as an argument the address of a vector holding a list of pointers to objects and terminating NULL. When one of the objects pointed to in the argument vector is ready, `waitvec()` returns the position in a vector of the object that caused the return; position numbering starts at 0.

object *who_alerted_me()

Returns a pointer to the `object` whose state change, from pending to ready, caused a task to be put back on the run chain (put in the `RUNNING` state).

Example

```
long t = sched::get_clock;
```

```
delay(10000);
```

Delays a task so that it resumes executing at *t*+10,000.

timer class

timer class — A timer delays for a specified amount of simulated time.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class timer: public sched
{
public:
                    timer(long delay);
                    ~timer();

    void            reset(long delay);
    void            setwho(object *alerter);

    virtual void    print(int verbosity, int internal_use = 0);
    virtual objtype o_type();
};
```

Description

Objects of this class are timers. When a timer is created its state is `RUNNING`, and it is scheduled to change its state to `TERMINATED` after a specified number of time units. When the timer becomes `TERMINATED`, tasks waiting for it are scheduled to resume execution.

Exception Handling

When a run-time error occurs, the following error code is passed to the `object::task_error()` function:

Value	Error Description
`E_TIMERDEL`	Cannot delete a timer that is `IDLE` or `RUNNING`

Constructors and Destructors

`timer(long delay)`

Constructs an object of the `timer` class and schedules it for *delay* time units after the current clock time.

~timer()

Deletes an object of the `timer` class; the timer's state must be `TERMINATED`.

Member Functions

virtual objtype o_type()

Returns `object::TIMER`.

virtual void print(int verbosity, int internal_use = 0)

Prints a `timer` object on `cout`. The *verbosity* argument specifies the information to be printed. Do not supply a value for the *internal_use* parameter.

void reset(long delay)

Sets the state of the timer to `RUNNING` (even if it was `TERMINATED`) and reschedules it to terminate after the specified delay from the current simulated time.

void setwho(object *alerter)

Returns `NULL`.

Example

```
extern "C" {
#include <stdlib.h>
}
#include <task.hxx>
#include <iostream.hxx>

class DelayTask: public task
{
public:
    DelayTask(char *, long);
};

// This task just does a delay, much like a timer.
DelayTask::DelayTask(char *task_name, long delay_length):
           task(task_name)
{
    cout << "at beginning of DelayTask, clock is "
         << sched::get_clock() << "\n";
    delay(delay_length);
    cout << "at end of DelayTask, clock is "
         << sched::get_clock() << "\n";
    thistask->resultis(0);
}

int main()
{
    cout << "at beginning of main\n";

    cout << "creating task\n";
    DelayTask delay_task1("delay_task1", 100);
```

```
    cout << "creating timer\n";
    timer *pt1 = new timer(10);
    cout << "waiting for timer\n";
    thistask->wait(pt1);
    cout << "clock is " << sched::get_clock() << "\n";

    cout << "resetting timer\n";
    pt1->reset(1000);
    cout << "waiting for timer\n";
    thistask->wait(pt1);
    cout << "clock is " << sched::get_clock() << "\n";

    cout << "at end of main\n";
    thistask->resultis(0);
    return EXIT_SUCCESS;
}
```

This code generates the following output:

```
at beginning of main
creating task
at beginning of DelayTask, clock is 0
creating timer
waiting for timer
clock is 10
resetting timer
waiting for timer
at end of DelayTask, clock is 100
clock is 1010
at end of main
```

urand class

urand class — Objects of the `urand` class generate uniformly distributed random integers within a given range from a low to a high value.

Header

```
#include <task.hxx>
```

Alternative Header

```
#include <task.h>
```

Declaration

```
class urand: public randint
{

public:
    int  low;
    int  high;

         urand(int arg_low, int arg_high);

    int  draw();
};
```

Data Members

int low

Is the lower bound of the range of generated random numbers.

int high

Is the upper bound of the range of generated random numbers.

Constructor

urand(int arg_low, int arg_high)

Constructs an object of the `urand` class. Generated random numbers are uniformly distributed from *arg_low* to *arg_high*.

Member Function

int draw()

Returns the next random integer generated by the object.

See Also

randint class

Chapter 11. vector Package

The vector package provides ways to define vectors or stacks of objects of any type by using the macro expansion capability of the VSI C++ preprocessor.

To declare a generic vector:

1. Include the header `<vector.hxx>` in your program and declare the `vector` class as follows:

   ```
   declare(vector, TYPE)
   ```

   TYPE may be any valid VSI C++ type name. Make sure you define the `declare` macro in every file that references this new vector data type.

2. Expand the implementation of all function bodies as follows:

   ```
   implement(vector, TYPE)
   ```

   This `implement` macro must appear once in a program.

3. Declare objects of type `vector` and *TYPE* and use the index operator to reference these objects. The following is an example of declaration and referencing:

   ```
   class MyType {/*...*/};
   declare(vector,MyType)
   implement(vector,MyType)
   vector(MyType) vec1(100), vec2(5);
   MyType x,y;
   //…
   if(vec2[4] == y) vec1[98] = x;
   ```

The *TYPE* parameter must be an identifier. If it is not a class name, a fundamental type, or a type name, create a name for the type using a `typedef` declaration. For example:

```
typedef char *PCHAR;
declare(vector, PCHAR)
implement(vector, PCHAR)
implement(vector, PCHAR)

void f()
{
    vector(PCHAR) ptrvec(10);
    char *p = "Text";

    ptrvec[0] = p;
    // ...
}
```

Thread Safety

The generation of error messages within the vector package is not thread safe; the package relies on static members to handle the current error message and there is no synchronization between threads. If this creates a problem for your application, VSI recommends that you define a single `Mutex` object to synchronize all use of the vector package. For more information on synchronizing access to user-defined objects, see Chapter 6.

stack(TYPE) class

stack(TYPE) class — Provides a generic (parameterized) data abstraction for a fixed-sized stack of objects of some given type.

Header

```
#include <vector.hxx>
```

Alternative Header

```
#include <vector.h>
```

Compile-Time Parameter

TYPE — The type of the objects in the stack. It must be an identifier.

Declaration

```
class stack(TYPE): private vector(TYPE)
{
public:
                     stack(TYPE)(int); // objection size_error
                     stack(TYPE)(stack(TYPE) &);

    void             push(TYPE);       // objection overflow_error
    TYPE             pop();            // objection underflow_error
    TYPE             &top();           // objection no_top_error
    int              full();
    int              empty();
    int              size();
    int              size_used();

    static Objection overflow_error;
    static Objection underflow_error;
    static Objection no_top_error;
};
```

Description

This class provides a generic (parameterized) data abstraction for a fixed-sized stack of objects of some given type.

Before a `stack` object can be declared or implemented, the base class, a `vector` object with the same type parameter, must also be declared and implemented. To declare a `stack` object you need to both declare and implement the base vector class and the stack class.

Exception Handling

Exceptions are implemented with the Objection package. The initial action function for all objections prints an error message on `cerr` and calls `abort()`.

Constructors

`stack(TYPE)(int size)`

Constructs a `stack` object with room for *size* elements in the stack. If *size* is less than or equal to 0, the objection `vector(TYPE)::size_error` is raised.

`stack(TYPE)(stack(TYPE) &src)`

Constructs a `stack` object that takes the initial values of the elements from another `stack` object of the same type and size.

Member Data

The following objections are raised for the stack errors described.

`static Objection no_top_error`

Attempted to reference the top of an empty stack.

`static Objection overflow_error`

Attempted to push too many elements onto the stack.

`static Objection underflow_error`

Attempted to pop an empty stack.

Member Functions

`int empty()`

Returns TRUE if the stack is empty; otherwise, it returns FALSE.

`int full()`

Returns TRUE if the stack is full; otherwise, it returns FALSE.

`TYPE pop()`

Pops an element off the top of the stack. If the stack underflows, the objection `stack(TYPE)::underflow_error` is raised.

`void push(TYPE new_elem)`

Pushes an element onto the stack. If the stack overflows, the objection `stack(TYPE)::overflow_error` is raised.

`int size()`

Returns the maximum number of elements in the stack.

`int size_used()`

Returns the number of elements currently used in a generic stack.

TYPE &top()

Returns a reference to the element on the top of the stack. If the stack is empty, the objection `stack(TYPE)::no_top_error` is raised.

Example

```
declare(vector, int)
implement(vector, int)
declare(stack, int)
implement(stack, int)

void f()
{
    stack(int) st(20);
    st.push(17);
    // ...
}
```

This example shows the four steps required to declare and implement the base `vector` class and to declare and implement the `stack` class.

See Also

Chapter 7

Chapter 3

vector(TYPE) class

vector(TYPE) class

vector(TYPE) class — Provides the (parameterized) data abstraction for a fixed-sized vector of objects of some given type.

Header

```
#include <vector.hxx>
```

Alternative Header

```
#include <vector.h>
```

Compile-Time Parameter

TYPE — The type of the objects in the vector. It must be an identifier.

Declaration

```
class vector(TYPE)
{
public:
```

```
                    // objection size_error
                    vector(TYPE)(int);
                    vector(TYPE)(vector(TYPE) &);
                    ~vector(TYPE)();
                    // objection copy_size_error
    vector(TYPE)    &operator=(vector(TYPE) &);
    TYPE            &elem(int);
                    // objection index_error
    TYPE            &operator[](int);
    int             size();
    void            set_size(int);

    static Objection size_error;
    static Objection copy_size_error;
    static Objection index_error;
};
```

Description

This class provides the (parameterized) data abstraction for a fixed-sized vector of objects of some given type.

Exception Handling

Exceptions are implemented with the Objection package. The initial action function for all objections prints an error message on `cerr` and calls `abort()`.

Constructors and Destructors

`vector(TYPE)(int new_size)`

Constructs a `vector` object with the integer argument representing the number of elements in the vector. If the number of elements is less than or equal to 0, the objection `vector(TYPE)::size_error` is raised.

`vector(TYPE)(vector(TYPE) &src)`

Constructs a `vector` object that takes initial values of the elements from another `vector` object of the same type and size.

`~vector(TYPE)()`

Deletes a `vector` object.

Member Data

The following objections are raised for the vector errors described.

`static Objection copy_size_error`

Attempted to assign a vector to another vector that has a different number of elements.

`static Objection index_error`

Attempted to reference a vector element with a subscript out of range.

static Objection size_error

Attempted to create a vector with less than one element in it.

Overloaded Operators

vector(TYPE) &operator = (vector(TYPE) &src)

Assigns a vector to another vector. If the sizes of the vectors are different, the objection `vector(TYPE)::copy_size_error` is raised.

TYPE &operator [] (int i)

Returns a reference to the *i*th element in the vector. The value of *i* has a range from 0 to `size()`–1. If the subscript is out of bounds, the objection

`vector(TYPE)::index_error`

is raised.

Other Member Functions

TYPE &elem(int i)

Behaves like `operator []` but without bounds checking.

void set_size(int new_size)

Changes the size of the vector.

int size()

Returns the number of elements in the vector.