Annex D (normative) Compatibility features [depr]

1 This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

2 These are deprecated features, where *deprecated* is defined as: Normative for the current edition of the Standard, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this section with the `deprecated` attribute (7.6.5).

D.1 Increment operator with `bool` operand [depr.incr.bool]

1 The use of an operand of type `bool` with the `++` operator is deprecated (see 5.3.2 and 5.2.6).

D.2 `register` keyword [depr.register]

1 The use of the `register` keyword as a *storage-class-specifier* (7.1.1) is deprecated.

D.3 Implicit declaration of copy functions [depr.impldec]

1 The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor (12.4, 12.8). In a future revision of this International Standard, these implicit definitions could become deleted (8.4).

D.4 Dynamic exception specifications [depr.except.spec]

1 The use of *dynamic-exception-specification*s is deprecated.

D.5 C standard library headers [depr.c.headers]

1 For compatibility with the C standard library and the C Unicode TR, the C++ standard library provides the 26 *C headers*, as shown in Table 155.

Table 155 — C headers

`<assert.h>`	`<inttypes.h>`	`<signal.h>`	`<stdio.h>`	`<wchar.h>`
`<complex.h>`	`<iso646.h>`	`<stdalign.h>`	`<stdlib.h>`	`<wctype.h>`
`<ctype.h>`	`<limits.h>`	`<stdarg.h>`	`<string.h>`	
`<errno.h>`	`<locale.h>`	`<stdbool.h>`	`<tgmath.h>`	
`<fenv.h>`	`<math.h>`	`<stddef.h>`	`<time.h>`	
`<float.h>`	`<setjmp.h>`	`<stdint.h>`	`<uchar.h>`	

2 Every C header, each of which has a name of the form `name.h`, behaves as if each name placed in the standard library namespace by the corresponding `cname` header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace `std` and are then injected into the global namespace scope by explicit *using-declaration*s (7.3.3).

3 [*Example:* The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — *end example*]

D.6 Old iostreams members [depr.ios.members]

1 The following member names are in addition to names specified in Clause 27:

```
namespace std {
  class ios_base {
  public:
    typedef T1 io_state;
    typedef T2 open_mode;
    typedef T3 seek_dir;
    typedef implementation-defined streamoff;
    typedef implementation-defined streampos;
    // remainder unchanged
  };
}
```

2 The type `io_state` is a synonym for an integer type (indicated here as `T1`) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

3 The type `open_mode` is a synonym for an integer type (indicated here as `T2`) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

4 The type `seek_dir` is a synonym for an integer type (indicated here as `T3`) that permits certain member functions to overload others on parameters of type `seekdir` and provide the same behavior.

5 The type `streamoff` is an implementation-defined type that satisfies the requirements of off_type in 27.2.2.

6 The type `streampos` is an implementation-defined type that satisfies the requirements of pos_type in 27.2.2.

7 An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.6.3.2.3):

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    void stossc();
    // remainder unchanged
  };
}
```

8 An implementation may provide the following member functions that overload signatures specified in Clause 27:

```
namespace std {
  template<class charT, class traits> class basic_ios {
  public:
    void clear(io_state state);
    void setstate(io_state state);
    void exceptions(io_state);
    // remainder unchanged
  };

  class ios_base {
  public:
    // remainder unchanged
  };

  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    pos_type pubseekoff(off_type off, ios_base::seek_dir way,
              ios_base::open_mode which = ios_base::in | ios_base::out);
```



```
    pos_type pubseekpos(pos_type sp,
              ios_base::open_mode which);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    basic_filebuf<charT,traits>* open
    (const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };
}
```

9 The effects of these functions is to call the corresponding member function specified in Clause 27.

D.7 `char*` streams [depr.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.7.1 Class `strstreambuf` [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);

    virtual ~strstreambuf();

    void  freeze(bool freezefl = true);
    char* str();
    int   pcount();
```



```
  protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);

  private:
    typedef T1 strstate;              // exposition only
    static const strstate allocated;  // exposition only
    static const strstate constant;   // exposition only
    static const strstate dynamic;    // exposition only
    static const strstate frozen;     // exposition only
    strstate strmode;                 // exposition only
    streamsize alsize;                // exposition only
    void* (*palloc)(size_t);          // exposition only
    void (*pfree)(void*);             // exposition only
  };
}
```

1 The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some *character* array type, whose elements store arbitrary values. The array object has several attributes.

2 [*Note:* For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:

— `allocated`, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;

— `constant`, set when the array object has `const` elements, so the output sequence cannot be written;

— `dynamic`, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

— `frozen`, set when the program has requested that the array object not be altered, reallocated, or freed.

— *end note*]

3 [*Note:* For the sake of exposition, the maintained data is presented here as:

— `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;

— `int alsize`, the suggested minimum size for a dynamic array object;

— `void* (*palloc)(size_t)`, points to the function to call to allocate a dynamic array object;

— `void (*pfree)(void*)`, points to the function to call to free a dynamic array object.

— *end note*]

4 Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.7.1.1 `strstreambuf` constructors [depr.strstreambuf.cons]

```
explicit strstreambuf(streamsize alsize_arg = 0);
```

1 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 156.

Table 156 — `strstreambuf(streamsize)` effects

Element	Value
`strmode`	`dynamic`
`alsize`	`alsize_arg`
`palloc`	a null pointer
`pfree`	a null pointer

```
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

2 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 157.

Table 157 — `strstreambuf(void* (*)(size_t), void (*)(void*))` effects

Element	Value
`strmode`	`dynamic`
`alsize`	an unspecified value
`palloc`	`palloc_arg`
`pfree`	`pfree_arg`

```
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
             signed char* pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
             unsigned char* pbeg_arg = 0);
```

3 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 158.

Table 158 — `strstreambuf(charT*, streamsize, charT*)` effects

Element	Value
`strmode`	0
`alsize`	an unspecified value
`palloc`	a null pointer
`pfree`	a null pointer

4 `gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:

— If `n > 0`, `N` is `n`.

— If `n == 0`, `N` is `std::strlen(gnext_arg)`.

— If `n < 0`, `N` is `INT_MAX`.[337]

5 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

6 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);
```

```
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

7 *Effects:* Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```
virtual ~strstreambuf();
```

8 *Effects:* Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if `strmode & allocated != 0` and `strmode & frozen == 0`. (D.7.1.3 describes how a dynamically allocated array object is freed.)

D.7.1.2 Member functions [depr.strstreambuf.members]

```
void freeze(bool freezefl = true);
```

1 *Effects:* If `strmode` & `dynamic` is non-zero, alters the freeze status of the dynamic array object as follows:

— If `freezefl` is `true`, the function sets `frozen` in `strmode`.

— Otherwise, it clears `frozen` in `strmode`.

```
char* str();
```

2 *Effects:* Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

3 *Remarks:* The return value can be a null pointer.

```
int pcount() const;
```

4 *Effects:* If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

337) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.8). The macro `INT_MAX` is defined in `<climits>` (18.3).

D.7.1.3 `strstreambuf` overridden virtual functions [depr.strstreambuf.virtuals]

```
int_type overflow(int_type c = EOF);
```

1 *Effects:* Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

— If `c != EOF` and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns `c` to `*pnext++`.

2 Returns `(unsigned char)c`.

— If `c == EOF`, there is no character to append.

3 Returns a value other than `EOF`.

4 Returns `EOF` to indicate failure.

5 *Remarks:* The function can alter the number of write positions available as a result of any call.

6 To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements `n` to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified.[338] If `palloc` is not a null pointer, the function calls `(*palloc)(n)` to allocate the new dynamic array object. Otherwise, it evaluates the expression `new charT[n]`. In either case, if the allocation fails, the function returns `EOF`. Otherwise, it sets `allocated` in `strmode`.

7 To free a previously existing dynamic array object whose first element address is `p`: If `pfree` is not a null pointer, the function calls `(*pfree)(p)`. Otherwise, it evaluates the expression `delete[]p`.

8 If `strmode & dynamic == 0`, or if `strmode & frozen != 0`, the function cannot extend the array (reallocate it with greater length) to make a write position available.

```
int_type pbackfail(int_type c = EOF);
```

9 Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

— If `c != EOF`, if the input sequence has a putback position available, and if `(char)c == gnext[-1]`, assigns `gnext - 1` to `gnext`.

10 Returns `c`.

— If `c != EOF`, if the input sequence has a putback position available, and if `strmode` & `constant` is zero, assigns `c` to `*--gnext`.

11 Returns `c`.

— If `c == EOF` and if the input sequence has a putback position available, assigns `gnext - 1` to `gnext`.

12 Returns a value other than `EOF`.

13 Returns `EOF` to indicate failure.

14 *Remarks:* If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type underflow();
```

15 *Effects:* Reads a character from the *input sequence*, if possible, without moving the stream position past it, as follows:

338) An implementation should consider `alsize` in making this decision.

— If the input sequence has a read position available, the function signals success by returning `(unsigned char)*gnext`.

— Otherwise, if the current write next pointer `pnext` is not a null pointer and is greater than the current read end pointer `gend`, makes a *read position* available by assigning to `gend` a value greater than `gnext` and no greater than `pnext`.

16 Returns `(unsigned char*)gnext`.

17 Returns `EOF` to indicate failure.

18 *Remarks:* The function can alter the number of read positions available as a result of any call.

```
pos_type seekoff(off_type off, seekdir way, openmode which = in | out);
```

19 *Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 159.

Table 159 — `seekoff` positioning

Conditions	Result
`(which & ios::in) != 0`	positions the input sequence
`(which & ios::out) != 0`	positions the output sequence
`(which & (ios::in | ios::out)) == (ios::in | ios::out))` and `way ==` either `ios::beg` or `ios::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

20 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 160.

Table 160 — `newoff` values

Condition	`newoff` Value
`way == ios::beg`	0
`way == ios::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios::end`	`seekhigh` minus the beginning pointer (`seekhigh - xbeg`).
If `(newoff + off) < (seeklow - xbeg)`, or `(seekhigh - xbeg) < (newoff + off)`	the positioning operation fails

21 Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

22 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.


```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out);
```

23 *Effects:* Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

- If `(which & ios::in) != 0`, positions the input sequence.
- If `(which & ios::out) != 0`, positions the output sequence.
- If the function positions neither sequence, the positioning operation fails.

24 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` from `sp.offset()`:

- If `newoff` is an invalid stream position, has a negative value, or has a value greater than (`seekhigh - seeklow`), the positioning operation fails
- Otherwise, the function adds `newoff` to the beginning pointer `xbeg` and stores the result in the next pointer `xnext`.

25 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
streambuf<char>* setbuf(char* s, streamsize n);
```

26 *Effects:* Implementation defined, except that `setbuf(0, 0)` has no effect.

D.7.2 Class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char* str();
  private:
    strstreambuf sb;  // exposition only
  };
}
```

1 The class `istrstream` supports the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

- `sb`, the `strstreambuf` object.

D.7.2.1 `istrstream` constructors [depr.istrstream.cons]

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

1 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,0))`. `s` shall designate the first element of an NTBS.


```
istrstream(const char* s, streamsize n);
```

2 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,n))`. `s` shall designate the first element of an array whose length is `n` elements, and `n` shall be greater than zero.

D.7.2.2 Member functions [depr.istrstream.members]

```
strstreambuf* rdbuf() const;
```

1 *Returns:* `const_cast<strstreambuf*>(&sb)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

D.7.3 Class `ostrstream` [depr.ostrstream]

```
namespace std {
  class ostrstream : public basic_ostream<char> {
  public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
    virtual ~ostrstream();

    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
  private:
    strstreambuf sb;  // exposition only
  };
}
```

1 The class `ostrstream` supports the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `strstreambuf` object.

D.7.3.1 `ostrstream` constructors [depr.ostrstream.cons]

```
ostrstream();
```

1 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)` and initializing `sb` with `strstreambuf())`.

```
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
```

2 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)`, and initializing `sb` with one of two constructors:

— If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s, n, s)`.

— If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s, n, s + std::strlen(s))`.[339]

339) The function signature `strlen(const char*)` is declared in `<cstring>` (21.8).

D.7.3.2 Member functions [depr.ostrstream.members]

```
strstreambuf* rdbuf() const;
```

1 *Returns:* `(strstreambuf*)&sb` .

```
void freeze(bool freezefl = true);
```

2 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

```
char* str();
```

3 *Returns:* `rdbuf()->str()`.

```
int pcount() const;
```

4 *Returns:* `rdbuf()->pcount()`.

D.7.4 Class `strstream` [depr.strstream]

```
namespace std {
  class strstream
    : public basic_iostream<char> {
  public:
    // Types
    typedef char                                  char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;

    // constructors/destructor
    strstream();
    strstream(char* s, int n,
              ios_base::openmode mode = ios_base::in|ios_base::out);
    virtual ~strstream();

    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();

  private:
  strstreambuf sb;  // exposition only
  };
}
```

1 The class `strstream` supports reading and writing from objects of classs `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as

— `sb`, the `strstreambuf` object.

D.7.4.1 `strstream` constructors [depr.strstream.cons]

```
strstream();
```

1 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)`.

```
strstream(char* s, int n,
          ios_base::openmode mode = ios_base::in|ios_base::out);
```

2 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)` and initializing `sb` with one of the two constructors:

— If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s,n,s)`.

— If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s,n,s + std::strlen(s))`.

D.7.4.2 `strstream` destructor [depr.strstream.dest]

```
virtual ~strstream()
```

1 *Effects:* Destroys an object of class `strstream`.

```
strstreambuf* rdbuf() const;
```

2 *Returns:* `&sb`.

D.7.4.3 `strstream` operations [depr.strstream.oper]

```
void freeze(bool freezefl = true);
```

1 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

```
int pcount() const;
```

3 *Returns:* `rdbuf()->pcount()`.

D.8 Function objects [depr.function.objects]

D.8.1 Base [depr.base]

1 The class templates `unary_function` and `binary_function` are deprecated. A program shall not declare specializations of these templates.

```
namespace std {
  template <class Arg, class Result>
  struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
  };
}
```



```
namespace std {
  template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  };
}
```

D.8.2 Function adaptors [depr.adaptors]

1 The adaptors ptr_fun, mem_fun, mem_fun_ref, and their corresponding return types are deprecated. [*Note:* The function template `bind` 20.9.9.1 provides a better solution. — *end note*]

D.8.2.1 Adaptors for pointers to functions [depr.function.pointer.adaptors]

1 To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
public:
  explicit pointer_to_unary_function(Result (*f)(Arg));
  Result operator()(Arg x) const;
};
```

2 `operator()` returns `f(x)`.

```
template <class Arg, class Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
```

3 *Returns:* `pointer_to_unary_function<Arg, Result>(f)`.

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function :
  public binary_function<Arg1,Arg2,Result> {
public:
  explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
  Result operator()(Arg1 x, Arg2 y) const;
};
```

4 `operator()` returns `f(x,y)`.

```
template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*f)(Arg1, Arg2));
```

5 *Returns:* `pointer_to_binary_function<Arg1,Arg2,Result>(f)`.

6 [*Example:*

```
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
  not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each `abc` with `def` in sequence `v`. — *end example*]

D.8.2.2 Adaptors for pointers to members [depr.member.pointer.adaptors]

1 The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in D.8.2.1.

```
template <class S, class T> class mem_fun_t
        : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
```

2 `mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <class S, class T, class A> class mem_fun1_t
      : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

3 `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<class S, class T> mem_fun_t<S,T>
   mem_fun(S (T::*f)());
template<class S, class T, class A> mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A));
```

4 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
```

5 `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <class S, class T, class A> class mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
```

6 `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.


```
template<class S, class T> mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
   mem_fun_ref(S (T::*f)(A));
```

7 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class const_mem_fun_t
      : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*p)() const);
  S operator()(const T* p) const;
};
```

8 `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <class S, class T, class A> class const_mem_fun1_t
      : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*p)(A) const);
  S operator()(const T* p, A x) const;
};
```

9 `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<class S, class T> const_mem_fun_t<S,T>
   mem_fun(S (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A) const);
```

10 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <class S, class T> class const_mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*p)() const);
  S operator()(const T& p) const;
};
```

11 `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <class S, class T, class A> class const_mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
  S operator()(const T& p, A x) const;
};
```


12 `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_ref_t<S,T,A>
    mem_fun_ref(S (T::*f)(A) const);
```

13 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

D.9 Binders [depr.lib.binders]

The binders `binder1st`, `bind1st`, `binder2nd`, and `bind2nd` are deprecated. [*Note:* The function template `bind` (20.9.9) provides a better solution. — *end note*]

D.9.1 Class template `binder1st` [depr.lib.binder.1st]

```
template <class Fn>
class binder1st
  : public unary_function<typename Fn::second_argument_type,
                          typename Fn::result_type> {
protected:
  Fn                      op;
  typename Fn::first_argument_type value;
public:
  binder1st(const Fn& x,
            const typename Fn::first_argument_type& y);
  typename Fn::result_type
    operator()(const typename Fn::second_argument_type& x) const;
  typename Fn::result_type
    operator()(typename Fn::second_argument_type& x) const;
};
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(value,x)`.

D.9.2 `bind1st` [depr.lib.bind.1st]

```
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn& fn, const T& x);
```

1 *Returns:* `binder1st<Fn>(fn, typename Fn::first_argument_type(x))`.

D.9.3 Class template `binder2nd` [depr.lib.binder.2nd]

```
template <class Fn>
class binder2nd
  : public unary_function<typename Fn::first_argument_type,
                          typename Fn::result_type> {
protected:
  Fn                       op;
  typename Fn::second_argument_type value;
public:
  binder2nd(const Fn& x,
            const typename Fn::second_argument_type& y);
```



```
  typename Fn::result_type
    operator()(const typename Fn::first_argument_type& x) const;
  typename Fn::result_type
    operator()(typename Fn::first_argument_type& x) const;
};
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(x,value)`.

D.9.4 `bind2nd` [depr.lib.bind.2nd]

```
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn& op, const T& x);
```

1 *Returns:* `binder2nd<Fn>(op, typename Fn::second_argument_type(x))`.

2 [*Example:*

```
find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

finds the first integer in vector `v` greater than 5;

```
find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

finds the first integer in `v` less than 5. — *end example*]

D.10 `auto_ptr` [depr.auto.ptr]

The class template `auto_ptr` is deprecated. [*Note:* The class template `unique_ptr` (20.8.1) provides a better solution. — *end note*]

D.10.1 Class template `auto_ptr` [auto.ptr]

1 The class template `auto_ptr` stores a pointer to an object obtained via `new` and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).

2 The class template `auto_ptr_ref` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a template with this name. The template holds a reference to an `auto_ptr`. It is used by the `auto_ptr` conversions to allow `auto_ptr` objects to be passed to and returned from functions.

```
namespace std {
  template <class Y> struct auto_ptr_ref;    // exposition only

  template <class X> class auto_ptr {
  public:
    typedef X element_type;

    // D.10.1.1 construct/copy/destroy:
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr_ref<X> r) throw();
   ~auto_ptr() throw();

    // D.10.1.2 members:
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
```



```
    X* release() throw();
    void reset(X* p =0) throw();

    // D.10.1.3 conversions:
    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref<Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
  };

  template <> class auto_ptr<void>
  {
  public:
    typedef void element_type;
  };
}
```

3 The class template `auto_ptr` provides a semantics of strict ownership. An `auto_ptr` owns the object it holds a pointer to. Copying an `auto_ptr` copies the pointer and transfers ownership to the destination. If more than one `auto_ptr` owns the same object at the same time the behavior of the program is undefined. [*Note:* The uses of `auto_ptr` include providing temporary exception-safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. Instances of `auto_ptr` meet the requirements of `MoveConstructible` and `MoveAssignable`, but do not meet the requirements of `CopyConstructible` and `CopyAssignable`. — *end note*]

D.10.1.1 `auto_ptr` constructors [auto.ptr.cons]

```
explicit auto_ptr(X* p =0) throw();
```

1 *Postconditions:* `*this` holds the pointer `p`.

```
auto_ptr(auto_ptr& a) throw();
```

2 *Effects:* Calls `a.release()`.

3 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

```
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

4 *Requires:* `Y*` can be implicitly converted to `X*`.

5 *Effects:* Calls `a.release()`.

6 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

```
auto_ptr& operator=(auto_ptr& a) throw();
```

7 *Requires:* The expression `delete get()` is well formed.

8 *Effects:* `reset(a.release())`.

9 *Returns:* `*this`.

```
template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
```

10 *Requires:* `Y*` can be implicitly converted to `X*`. The expression `delete get()` is well formed.

11 *Effects:* `reset(a.release())`.

12 *Returns:* `*this`.


```
~auto_ptr() throw();
```

13 *Requires:* The expression `delete get()` is well formed.

14 *Effects:* `delete get()`.

D.10.1.2 `auto_ptr` members [auto.ptr.members]

```
X& operator*() const throw();
```

1 *Requires:* `get() != 0`

2 *Returns:* `*get()`

```
X* operator->() const throw();
```

3 *Returns:* `get()`

```
X* get() const throw();
```

4 *Returns:* The pointer `*this` holds.

```
X* release() throw();
```

5 *Returns:* `get()`

6 *Postcondition:* `*this` holds the null pointer.

```
void reset(X* p=0) throw();
```

7 *Effects:* If `get() != p` then `delete get()`.

8 *Postconditions:* `*this` holds the pointer `p`.

D.10.1.3 `auto_ptr` conversions [auto.ptr.conv]

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

1 *Effects:* Calls `p.release()` for the `auto_ptr p` that `r` holds.

2 *Postconditions:* `*this` holds the pointer returned from `release()`.

```
template<class Y> operator auto_ptr_ref<Y>() throw();
```

3 *Returns:* An `auto_ptr_ref<Y>` that holds `*this`.

```
template<class Y> operator auto_ptr<Y>() throw();
```

4 *Effects:* Calls `release()`.

5 *Returns:* An `auto_ptr<Y>` that holds the pointer returned from `release()`.

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw()
```

6 *Effects:* Calls `reset(p.release())` for the `auto_ptr p` that `r` holds a reference to.

7 *Returns:* `*this`

D.11 Violating *exception-specification*s [exception.unexpected]

D.11.1 Type `unexpected_handler` [unexpected.handler]

```
typedef void (*unexpected_handler)();
```

1 The type of a *handler function* to be called by `unexpected()` when a function attempts to throw an exception not listed in its *dynamic-exception-specification*.

2 *Required behavior:* An `unexpected_handler` shall not return. See also 15.5.2.

3 *Default behavior:* The implementation's default `unexpected_handler` calls `std::terminate()`.

D.11.2 `set_unexpected` [set.unexpected]

```
unexpected_handler set_unexpected(unexpected_handler f) noexcept;
```

1 *Effects:* Establishes the function designated by `f` as the current `unexpected_handler`.

2 *Remark:* It is unspecified whether a null pointer value designates the default `unexpected_handler`.

3 *Returns:* The previous `unexpected_handler`.

D.11.3 `get_unexpected` [get.unexpected]

```
unexpected_handler get_unexpected() noexcept;
```

1 *Returns:* The current `unexpected_handler`. [*Note:* This may be a null pointer value. — *end note*]

D.11.4 `unexpected` [unexpected]

```
[[noreturn]] void unexpected();
```

1 *Remarks:* Called by the implementation when a function exits via an exception not allowed by its *exception-specification* (15.5.2), in effect after evaluating the throw-expression (D.11.1). May also be called directly by the program.

2 *Effects:* Calls the current `unexpected_handler` function. [*Note:* A default `unexpected_handler` is always considered a callable handler in this context. — *end note*]

D.12 Random shuffle [depr.alg.random.shuffle]

The function templates `random_shuffle` are deprecated.

```
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator&& rng);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The random number generating function object `rng` shall have a return type that is convertible to `iterator_traits<RandomAccessIterator>::difference_type`, and the call `rng(n)` shall return a randomly chosen value in the interval `[0,n)`, for `n > 0` of type `iterator_traits<RandomAccessIterator>::difference_type`.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4 *Remarks:* To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library.

In the second form of the function, the function object `rng` shall serve as the implementation's source of randomness.

