Minimal Runtime System

MIN

(Draft 1a)

Robert L. Walton

May 23, 2023

Table of Contents

1 Introduction

This document describes the internal design of MIN, the Minimal Runtime System. This document is written for readers who wish to add C++ code to a MIN implementation, or who wish to maintain an implementation.

MIN might better be described as a 'minimal complete' runtime system. It has a number of advanced features, such as support for objects that combine a vector with a hash table, support for an ephemeral real-time garbage collector, and advanced printing support.

A runtime system provides run time and compile time support for programming languages. A runtime system, such as MIN, is built on top of an underlying runtime system provided by the implementation language, in this case C++. MIN consists of a data store, of the single-thread execution system inherited from C++, and of MIN compatible threads. Other programming language specific runtime systems can be built on top of MIN.

A modest number of ***abbreviations*** are used throughout this document and the corresponding code:

`acc`	allocator/collector/compactor (5.6 [p34],C [p320])
`aux`	auxiliary (5.2 [p14])
`attr`	attribute (5.19.5 [p180])
`bra`	opening bracket (p229)
`DISP`	displacement (p85)
`gen`	general value (5.2 [p13])
`gtyped`	graph typed (5.19.8 [p224])
`insptr`	insertable pointer (read-write, push, pop pointer)
`int`	signed integer (p11)
`ket`	closing bracket (p229)
`lab`	label (5.11 [p79])
`min`	the '**`min`**' namespace (p6)
`MACC`	**`min::acc`** (p320)
`MIN_`	the `MIN_` macro preface (p6)
`MINT`	**`min::internal`** (p7)
`MOS`	**`min::os`** (p320)
`MUP`	**`min::unprotected`** (p6)
`num`	number (p69)
`obj`	object (p173)
`ptr`	pointer (often read-only, see updptr and insptr)
`ref`	reference (p28)
`sep`	separator (p229)
`str`	string (p71)
`uns`	unsigned integer (p11)
`updptr`	updatable pointer (i.e., read-write pointer)

var	variable (5.19.5 ^{p180})
vec	vector (5.19.5 ^{p180})

Also one of a small number of ***subscripts*** may be attached to a function name to indicate that the function is of a certain class of functions:

C	compact function (p14)
L	loose function (p14)
R	relocating function (p12)
S	resizing function (p177)
O	reorganizing function (p178)

2 Interfaces

MIN code and documentation is organized within the following directories:

.../**min/include**	***.h** files such as **min.h**
.../**min/unicode**	UNICODE data base files such as **unicode_data.h** (independent of the rest of MIN)
.../**min/src**	***.cc** files such as **min.cc**
.../**min/test**	test scripts such as **min_interface_test.cc**
.../**min/doc**	documentation files such as **min.tex**
.../**min/lib**	contains **libmin.a** library of MIN binary files

The C++ data and functions described in this document can be accessed by C++ code that contains the following inclusion:

```
#include  <min.h>
```

External MIN data and functions are placed in the **min** namespace. There are some macros that can be defined to control compilation, and these have names beginning with **MIN_**.

MIN has two interfaces: the ***protected interface***, which can be used by C++ code to access MIN while maintaining the integrity of MIN data, and the ***unprotected interface***, which provides more efficient access to MIN data but requires the user to follow certain protocols to be sure that data are not damaged.

From the syntactic point of view the only distinction between these interfaces is that the unprotected interface is in the with **min::unprotected** namespace, whereas the protected interface is in the **min** namespace. User code that accesses the unprotected interface typically abbreviates the long '**min::unprotected::**' prefix to '**MUP::**' by including the following definition:

```
#define  MUP  min::unprotected
```

In this document we will use the abbreviation '**MUP**' for '**min::unprotected**'. Note that namespace **#define**'s such as the above are not included in **min.h** or other MIN **.h** files; they must be included explicitly in **.cc** files.

Most **MUP** functions 'produce ***undefined results***' when their arguments are out of legal range. This means that when the arguments are out of range, function execution may lose control and crash, or may produce result values that are wrong or out of legal range. When documentation of a **MUP** function states that particular argument range checks are not performed, then the **MUP** function always produces undefined results when its arguments would not pass these checks, unless the documentation explicitly says otherwise.

The protection provided by the protected interface is obtained by the functions of that interface using the **MIN_ASSERT**, **MIN_REQUIRE**, or **MIN_ABORT** macros of Section 3 ^{p8}.

The following are the most commonly used compiler options that redefine **MIN_** macros:

Macro Name	Meaning	Page Reference
-DMIN_IS_COMPACT=1	make implementation compact	p18
-DMIN_NO_PROTECTION=1	suppress integrity checks	p8

The other compilation parameters involve technical details of memory management. For specifics see the file **min_parameters.h**.

Protected functions are in the **min** namespace and have names beginning with '**min::**', while unprotected MIN functions are in the **min::unprotected** namespace and have names beginning with '**min::unprotected::**'. The **min** and **min::unprotected** namespaces hold all the stable interfaces of the MIN implementation.

Code in **min.h** that is not meant to be accessed by users is in the following namespaces:

Namespace	Abbre-viation	Use
min::os	**MOS**	Operating system independent interface to operating system functions not covered by C++ standards. See Appendix B ^{p320}.
min::acc	**MACC**	The part of the interface to the Allocator/Collector/Compactor that can be changed when the acc is changed. See Appendix C ^{p320}.
min::internal	**MINT**	Interface to internal MIN code that can be changed without notice at any time. Not described in this document; see **.h** files.
min::unicode	**UNI**	Interface to UNICODE data base.

3 Assert Macros

Protected functions use the following macro to enforce protection:

`MIN_ASSERT(`e`,...)`	By default, defined to evaluate e, and if the value is false, call the **`min::assert_hook`** function which by default prints an error message that includes the character string generated by a call to **`printf(...)`** and then calls **`abort()`**. **`printf`** is not called if `...` is `NULL`.

`MIN_ASSERT` is actually defined to be one of the following:

`MIN_ASSERT_CALL_ON_FAIL(`e`,...)`	The default value of `MIN_ASSERT` if `MIN_NO_PROTECTION == 0`. Evaluates e, and if the value is false, calls the **`min::assert_hook`** function.
`MIN_ASSERT_CALL_ALWAYS(`e`,...)`	Just always calls the **`min::assert_hook`** function. By default, that function just returns if e evaluates to **`true`**.
`MIN_ASSERT_CALL_NEVER(`e`,...)`	Does nothing. Does not even evaluate e. The default value of `MIN_ASSERT` if `MIN_NO_PROTECTION == 1`.

Thus by default `MIN_ASSERT` is defined by

```
# define MIN_ASSERT MIN_ASSERT_CALL_ON_FAIL
```

and it can be disabled by

```
# define MIN_ASSERT MIN_ASSERT_CALL_NEVER
```

or set for debugging by

```
# define MIN_ASSERT MIN_ASSERT_CALL_ALWAYS
```

If not defined by a compiler option or a definition preceeding **`#include`** statements, the definition of `MIN_ASSERT` is controlled by:

`MIN_NO_PROTECTION`	1 to suppress integrity checks, such as `MIN_ASSERT`'s, `MIN_REQUIRES`'s, and code that initializes variables that should never be use. Defaults to 0.

Three alternatives to `MIN_ASSERT` are:

MIN_REQUIRE(*e*) Defined to always be the equivalent of **MIN_ASSERT(*e*,NULL)**. Used for system integrity checks.

MIN_CHECK(*e*) Defined to always be the equivalent of **MIN_ASSERT_CALL_ALWAYS(*e*,NULL)**, except that **MIN_CHECK** always passes a **NULL** **function_name** argument to **min::assert_hook** so printed messages will have less clutter. Used by diagnostic programs.

MIN_ABORT(...) Defined to always be the equivalent of **MIN_ASSERT_CALL_ALWAYS(false,...)**. Used when code transfers to a place it should never transfer to.

min::assert_hook is actually a pointer-to-function variable that can be modified to change the function called by the **MIN_ASSERT**, **MIN_REQUIRE**, and **MIN_ABORT** macros. The interface is:

```
void ( * min::assert_hook )
                  ( bool value,
                    const char * expression,
                    const char * file_name, unsigned line_number,
                    const char * function_name,
                    const char * message_format, ... )
```

The **value** argument to **min::assert_hook** is the value of the *e* argument to **MIN_ASSERT**. The **expression** argument is the expression *e* that is evaluated to produce the **value** argument, and is used for printing messages. It is **NULL** only on a call from **MIN_ABORT**, and is not used in this case. The **file_name** and **line_number** arguments identify the file and line containing the **MIN_ASSERT**, and are used in printing messages. The **function_name** argument identifies the function inside of which **MIN_ASSERT** was executed, and is used for printing messages, but may be **NULL** if the function name is not known. The **message_format** argument to **min::assert_hook**, and the arguments following it, are the values of the **...** arguments to **MIN_ASSERT**, and are used as **printf** arguments to print messages, unless **message_format** is **NULL**.

The default value of **min::assert_hook** and data used by this default are:

```
  void min::standard_assert
               ( bool value,
                 const char * expression,
                 const char * file_name, unsigned line_number,
                 const char * function_name,
                 const char * message_format, ... )
  bool min::assert_print = false
  bool min::assert_throw = false
  bool min::assert_abort = true
struct min::assert_exception { }
```

If **min::assert_print** is **true**, or if **value** is **false** and **min::assert_throw** is false, this function prints a message containing **file_name** and **line_number**, **function_name** if that is not **NULL**, **expression** and **value** if **expression** is not **NULL** or the word '**abort**' if **expression** is **NULL**, and **message_format** and subsequent arguments as per **printf** if **message_format** is not **NULL**.

Then if **value** is **false** this function throws a **min::assert_exception** if **min::assert_throw** is **true**, else calls **abort()** if **min::assert_abort** is **true**, else just returns.

On a call from **MIN_ABORT**, **value** is **false** and **expression** is **NULL**. On a call from any non-**ABORT** function, **expression** is not **NULL**.

4 Initialization

Because C++ compilers are inconsistent in the order in which they call constructors for **static** data, MIN initialization is not done until after **main()** begins execution. **main()** should begin with either

```
min::initialize();
```

or

min::interrupt^R();

Either will initialize MIN.

Generally MIN out-of-line functions cannot be called by constructors of **static** or **extern** data, because MIN is not initialized when these constructors execute. MIN constructors designed to be used by such data do not call MIN out-of-line functions or depend upon other MIN data being constructed. The following are some of the MIN constructors and **inline** functions that can be invoked during construction of **static** and **extern** data:

```
min::gen;
min::MISSING();
min::NONE();
min::locatable_gen;
min::locatable_var<T>;
min::ptr<T>;
min::packed_struct<S>;
min::packed_vector<E>;
min::packed_vector<E,H,L>;
```

After MIN initializes itself it calls initializers that are declared by including the following in a translation unit:

```
static void my_initialize ( void ) { ... }
static min::initializer my_initializer ( ::my_initialize );
```

Immediately after MIN performs its own initialization it calls **`my_initialize()`** and other functions specified by **`min::initializer`**'s. The order in which these are called is indeterminate. These functions can call any MIN function and can finish initialization of **`static`** or **`extern`** data.

5 Data

In defining MIN data the following number types are used to be sure the size of each number is clear:

`min::uns8`	unsigned 8-bit integer
`min::int8`	signed 8-bit integer
`min::uns16`	unsigned 16-bit integer
`min::int16`	signed 16-bit integer
`min::uns32`	unsigned 32-bit integer
`min::int32`	signed 32-bit integer
`min::float32`	32-bit IEEE floating point number
`min::uns64`	unsigned 64-bit integer
`min::int64`	signed 64-bit integer
`min::float64`	64-bit IEEE floating point number
`min::unsptr`	unsigned integer of same length as a pointer (32 or 64 bits)
`min::intptr`	signed integer of same length as a pointer (32 or 64 bits)
`min::unsgen`	unsigned integer of same length as a general value (32 or 64 bits, see 5.2 p13)
`min::Uchar`	unsigned 32-bit integer (used as UNICODE character)

Current implementations of MIN assume a compiler that has direct support for 64 bit integers. It is possible, but not recommended, to modify the implementations to use compilers without such support.

MIN depends upon certain undocumented C++ conventions.

First, it is assumed that classes will not have hidden padding that is not required to align member offsets. It is assumed that the compiler will allocate class members in order without any hidden padding if this will lead to an allocation in which each member of a number type above or of any pointer type has an offset in class instances that is a multiple of the member length.

Second, it is assumed that the contents of a base class are allocated is if the class contained an unnamed first member whose type is the base class.

Care is taken to use these first two assumptions when defining MIN data this may be input or output in binary form. Some checks on these assumptions are programmed into MIN initialization routines using C++ '**`assert`**' statements.

Third, it is assumed that a class containing just one element that is a number or pointer is just as efficient in optimized code as a number or pointer not in a class. In particular, such a one-element class used as an argument or return value should be passed in a register.

All these assumptions seem to be satisfied by the GNU C++ compiler.

5.1 Stubs and Bodies

MIN data memory consists of regions that contain stubs and regions that contain bodies. A region is a contiguous block of memory, typically consisting of an integral number of hardware pages.

Stubs are small fixed size units of memory that cannot be relocated: the usual stub size for MIN is 16 bytes. Each object has a stub, and the address of the stub is in effect the internal name of the object. Some atoms (e.g., numbers, strings) have stubs, and some (e.g., integers that are not large, and very short strings) do not.

A stub is divided into an 8 byte ***stub value*** and an 8 byte ***stub control***. The stub value can be a 64-bit IEEE floating point number, an 8 **`char`** string, or, as we will soon see, a pointer to a body. It is also possible for a stub value to hold any other 8 bytes of information.

The stub control holds a 1 byte ***type code*** and other information used, for example, by the allocator/collector/compactor (acc).

The type name of a stub is '**`min::stub`**', and a pointer to a stub has type '**`min::stub *`**'. Protected functions can return '**`const min::stub *`**' values, but only unprotected functions can return '**`min::stub *`**' values.

A ***body*** is a variable sized **`relocatable`** block of memory attached to a particular stub. A stub may have a body attached to it, in which case the stub value is a pointer to that body. When certain functions are called, any body can be moved and its corresponding stub value reset to point at the new location of the body. Functions with this property are called '***relocating functions***' and their names are marked by the superscript $^{\mathbf{R}}$ in this document. Included are functions that allocate objects. Obtaining a C++ pointer into a body is an unprotected operation, because the pointer must be updated if a relocating function is called.

Interrupts can relocate bodies. Therefore interrupts are only allowed at specific points in the code. The inline function

> `bool min::`**`interrupt`** ^{R} `( void )`

checks an interrupt flag, and if that is set, calls an out-of-line function to process the interrupt. The function returns **`true`** if and only if there was actually an interrupt (this should be used only for optimization).

A body may be ***deallocated*** by moving it to unimplemented memory. When this is done the stub ***type code*** is reset to the value **`min::DEALLOCATED`**, which indicates the body is deallocated. Deallocation is considered to be a variant of relocation. Relocating functions, those marked marked by ^{R} in documentation, may also deallocate objects.

Deallocation is done by the **`min::deallocate`** ^{R} function (p40) and testing to see if a a body has been deallocated is done by the **`min::is_deallocated`** function (p41).

Bodies are always some multiple of 8 bytes long, and are allocated on 8 byte boundaries.

Protected functions that take a stub pointer as argument use **`MIN_ASSERT`** macros (p8) to check the ***type code*** of the stub and various lengths. Unprotected functions contain no such checks.

Memory consisting of unrelocatable stubs pointing at relocatable bodies is called a '***stub/ body memory***'. Thus MIN has a stub/body memory. The main advantages of stub/body memory are that relocation of bodies can happen independently of other program activity, and bodies can be deallocated by program command at any time.[1]

5.2 General Values

A general value can store any of:

a direct atom value
a pointer to a stub
an auxiliary pointer
an index
a control code
a special value

General values are used as attribute names and values in objects and as function arguments and return values.

General values that represent numbers or character strings are called ***atoms***, because they have no subcomponents. There are two kinds of atoms: ***direct atoms*** that are stored completely in a general value, and ***indirect atoms*** that are stored in a stub or in a body pointed at by a stub, with the stub being pointed at by a general value.

Efficiency aside, it does not matter whether a general value stores a direct atom value or a pointer to a stub holding an indirect atom value, as atom values are immutable and cannot be changed. Of course not all atom values will fit into a general value, and those that do not must be stored in stubs or in bodies pointed at by stubs.

[1]Stub/body memories are certainly not new. For example, Kyoto COMMONLISP used a stub/body implementation of arrays, and the author has heard about implementations that go back to the late 1950's or early 60's: see p33 of the author's thesis, R-CODE, A Very Capable Virtual Computer.

An ***auxiliary pointer*** is an integer that is used by a general value stored inside a body to point at some part, called an ***auxiliary***, of the same body. There are several subtypes of auxiliary pointers storable in bodies. See p174.

An ***index*** is an integer that is used to give the index of a variable that is an element of an object body. See p225.

A ***control code*** is an integer that represents flags and codes stored in a general value. Control codes can have different interpretations in different contexts. For example, see 'Attribute Flags', p208.

A ***special value*** is a unique value that has some special meaning. **`min::MISSING()`**, for example, is a special value that may be input or output to indicate that data is missing, and **`min::NONE()`** is a special value used only to indicate that a function argument or result does not exist. See 5.3 p23.

The **`min::gen`** type is defined as a C++ class that consists of a single **`private`** element of type **`min::unsgen`**, which is an unsigned integer. From the point of view of C++ type checking, a **`min::gen`** value is a class, but from the computational point of view it is an unsigned integer.

Because **`min::gen`** is a class type, it is not possible to present **`min::gen`** constants to the compiler for use in optimized code in a completely straightforward way. To get the desired effect, **`min::gen`** constants are represented by **`inline`** functions, which allows the optimizing compiler to insert the constants directly into instructions. Thus we have the constant **`min::MISSING()`**, a function call with no arguments, instead of **`min::MISSING`**, a datum.

There are two kinds of MIN implementation: '***compact***' and '***loose***'. A compact implementation uses 32-bit general values, while a loose implementation uses 64-bit general values. An implementation cannot use both 32-bit and 64-bit general values; the implementation must use one or the other.

The 64-bit loose implementation formats a **`min::gen`** value as an IEEE floating point number, using the NaN (Not-a-Number) values to encode non-numeric **`min::gen`** values, such as pointers to stubs. Thus in the loose implementation **`min::float64`** values are stored verbatim in **`min::gen`** values.

Some functions and constants are defined only for compact implementations, and some only for loose implementations. Those defined only for compact implementations are called '***compact functions***' and their names are marked with the superscript $^{\mathbf{C}}$ in this document. Those defined only for loose implementations are similarly called '***loose functions***' and their names are marked with the superscript $^{\mathbf{L}}$.

The value of a compact implementation is that it uses less memory[2], but there may be a speed penalty. The value of a loose implementation is that it may run faster, but there is a memory penalty. It is not clear what the speed difference between the two implementations

[2]However, double precision floating point numbers only use less memory if each is replicated several times.

really is, so both implementations are offered in order to decide the issue by experiment.

Also compact implementations do not permit more than about 2^{32} objects to exist in memory at one time, whereas loose implementations permit up to about 2^{44} objects.

5.2.1 General Value Subtypes

A ***general value*** has type **`min::gen`** and is a 32 or 64 bit aligned value that can be of one of the following subtypes;

> a pointer to a stub
> a 64-bit IEEE floating point direct number atom ^{L}
> a 28-bit direct integer atom ^{C}
> a 0-5 **`char`** direct string atom ^{L}
> a 0-3 **`char`** direct string atom ^{C}
> a VSIZE-bit index
> a VSIZE-bit control code
> a VSIZE-bit special value
> a VSIZE-bit list auxiliary pointer
> a VSIZE-bit sublist auxiliary pointer
> a VSIZE-bit indirect auxiliary pointer
>
> > where ***VSIZE*** equals 24 for a compact implementation and 40 for a loose implementation

Here ^{C} subtypes are only implemented by compact implementations, and ^{L} subtypes are only implemented by loose implementations (p14).

Numbers and character strings stored inside a **`min::gen`** value are called '***direct atoms***'. Numbers and character strings stored in stubs or bodies which are pointed at by a **`min::gen`** value are called '***indirect atoms***'. An atom is always stored in only one way by an implementation. If a number will fit into a direct atom, it is stored as a direct atom, and never as an indirect atom. Similarly if a character string will fit into a direct atom, it is stored as a direct atom, and never as an indirect atom.

Analogously indirect atoms are stored in only one place in memory, so two indirect numbers are equal if and only the **`min::gen`** values pointing at them point at the same place, and similarly two indirect character strings are equal if and only the **`min::gen`** values pointing at them point at the same place.

Therefore two atoms are equal if and only if the **`min::gen`** values designating them are **`==`** as 32-bit or 64-bit values.

General values that point at stubs hold stub addresses in a packed format. 44 bits are available to store a stub address in a 64-bit general value, and 32 bits are available to store a stub address in a 32-bit general value, but in the latter case the packed address must be less than $2^{32} - 2^{29}$ (the 2^{29} other 32 bit values are used to store direct integers, direct strings, auxiliary pointers, indices, control codes, and special values). There are three ***stub address packing schemes***, and the fastest is selected for each type of general value according to the settings of the following '***stub address packing parameter***' macros:

`MIN_MAX_ABSOLUTE_STUB_ADDRESS`	The maximum absolute address of any stub as an unsigned integer constant. See p19 for defaults.
`MIN_MAX_RELATIVE_STUB_ADDRESS`	The maximum address of any stub relative to a constant stub base address, as an unsigned integer constant. See p18 for defaults.

The fastest scheme is the ***absolute stub address*** scheme, where the absolute stub address is stored. For 64 bit general values this is just a matter of inserting the stub address into the low order 44 bits of the value. For 32 bit general values this is just a matter of using the stub address as the general value. This scheme can be used if `MIN_MAX_ABSOLUTE_STUB_ADDRESS` can be stored in 44 bits for 64 bit general values, or is at most $2^{32} - 2^{29} - 1$ for 32 bit general values.

The second fastest scheme is the ***relative stub address*** scheme, where the stub address relative to a constant ***stub base*** address is stored. The relation between the absolute and relative stub addresses is:

$$\text{stub absolute address} = \text{stub base} + \text{stub relative address}$$

where 'stub base' is a constant determined when the program is initialized. This scheme can be used if `MIN_MAX_RELATIVE_STUB_ADDRESS` can be stored in 44 bits for 64 bit general values, or is at most $2^{32} - 2^{29} - 1$ for 32 bit general values.

The slowest scheme is the ***stub index*** scheme, where the relative stub address divided by the stub length is stored. The stub length is chosen to be a power of 2 so that this scheme will be efficient. Again, this scheme can be used if the relative address of the stub is not too large, but since the stub length is 16, the relative address can be 16 times larger for the stub index packing scheme than it is for the stub relative address packing scheme. Thus the index scheme can be used if `MIN_MAX_RELATIVE_STUB_ADDRESS/16` can be stored in 44 bits for 64 bit general values, or is at most $2^{32} - 2^{29} - 1$ for 32 bit general values.

Note that on machines that have 32 bit addresses (e.g., the IA32[3] machines), there is little to gain by either the relative stub address or stub index packing schemes, so only the absolute stub address scheme should be used on such machines.

A 32-bit general value direct integer consists of a high order 4-bit subtype code and a low order 28-bit signed integer stored in offset form, so the true integer can be derived from the

[3] Intel Architecture 32-bit, a.k.a, i386 machines.

min::gen value by subtracting the **min::gen** representation of zero. The range of such a direct integer is the same as the range of a 28-bit two's complement integer: -2^{27} to $2^{27}-1$, inclusive.

Other 32-bit general values consist of a high order 8-bit subtype code and a low order 24-bit datum. For most auxiliary pointer general values the datum is a 24-bit auxiliary pointer (see p174 for a definition of auxiliary pointers). For index and special values the datum is a 24-bit index. For control code general values the datum is 24-bits that contains flags and codes that have different interpretations in different contexts. And for direct string general values the datum holds 0 to 3 8-bit **char**'s.

For 64-bit general values that point at stubs, the high order 20 bits are used as a subtype code. For other 64-bit general values, the high order 24 bits are the subtype code and the low order 40 bits are the datum. The values chosen for these subtype codes make stub pointers, direct strings, auxiliary pointers, indices, and control codes – that is, all non-number **min::gen** values – into IEEE Nan (Not-a-Number) values that are never generated by hardware instructions.

For most auxiliary pointer general values the 40-bit datum is used to hold a 40-bit auxiliary pointer. For index and special values the datum holds a 40-bit index. For control code general values the datum is a 40-bit control code; and for direct string general values the datum holds 0-5 **char**'s.

The **min::gen** type has the alignment properties of either **min::uns32** or **min::uns64**, and **min::unsgen** is **typedef**'ed to the appropriate one of these two types.

Many **min::gen** values, which are 32 or 64 bits, are divided into a subtype, either 8 or 24 bits, and a datum, either 24 or 40 bits. In this context the datum is called the '***value***'.

The following definitions are provided in **min.h** to facilitate coding:

```
typedef min::uns32 min::unsgen^C
typedef min::uns64 min::unsgen^L
         min::gen MUP::new_gen ( min::unsgen value )
      min::unsgen MUP::value_of ( min::gen value )
    (constructor) min::gen ( void )
   const unsigned min::TSIZE
   const unsigned min::VSIZE
```

min::unsgen is typedef'ed to **min::uns32**^{C}or **min::uns64**^{L}. **MUP::new_gen** and **MUP::value_of** are unprotected converters between **min::unsgen** and **min::gen** values.

min::gen (void) is the default constructor that sets an unassigned **min::gen** value so it will more likely trigger a fault if read by mistake (there is an exception for **min::gen** elements of packed structures and packed vectors, which are set to **0** when the structures or vectors are created: see 5.13^{p84} and 5.14^{p91}). Note that the existence of this constructor makes it impossible to include **min::gen** values in C++ **union**'s, but **min::unsgen** values can be included instead and converted using the unprotected converters.

`TSIZE` is the subtype size in bits; equal to `8`^{C}or `24`^{L}. `VSIZE` is the value size in bits; equal to `24`^{C}or `40`^{L}.

5.2.2 General Value Compilation

The decisions on whether an implementation is compact or loose (p14) must be made before C++ code is compiled. Decisions must also be made determining the stub address packing parameters (p16). The following macros, which must be defined identically for all separately compiled parts of a single program, describe these decisions. These macros are in two groups, first the macros that are set by the programmer, and then the macros which by default are computed from the settings of the first group of macros. All these macros are defined in **`min_parameters.h`**.

The macros set by the programmer are:

`MIN_IS_COMPACT`	1 if compact implementation; 0 if loose; defaults to 0.
`MIN_MAX_EPHEMERAL_LEVELS`	Maximum number of garbage collector ephemeral levels allowed in any execution of the compiled binary; defaults to 2.

The macros which are normally set to default values computed from the above macro settings are:

`MIN_IS_LOOSE`
: 1 if loose implementation; 0 if compact; Must equal ‘`! MIN_IS_COMPACT`’.

`MIN_MAX_NUMBER_OF_STUBS`
: The maximum number of stubs that can exist in any execution of the compiled binary. Defaults:

if `MIN_IS_COMPACT` = 1:	$2^{28} - 2^{25}$
else pointers are 32 bits:	2^{28}
else:	2^{40-4e}

where
$e = max(0, \texttt{MIN\_MAX\_EPHEMERAL\_LEVELS} - 2)$.

`MIN_STUB_BASE`
: The value of the ‘stub base’ for relative addressing (see p16). If defined this must be a non-negative integer constant. Defaults to `0` if `MIN_MAX_NUMBER_OF_STUBS` is set by default, and otherwise is left undefined.

`MIN_MAX_RELATIVE_STUB_ADDRESS`
: The maximum address of any stub relative to the ‘stub base’ (address of the first stub), as an unsigned integer constant. Default:
`16*MIN_MAX_NUMBER_OF_STUBS - 1`.

MIN_MAX_ABSOLUTE_STUB_ADDRESS
The maximum absolute address of any stub. Defaults:
if **MIN_STUB_BASE** defined:
  MIN_STUB_BASE
  + MIN_MAX_RELATIVE_STUB_ADDRESS
else if **MIN_IS_COMPACT** = 1 and
  pointers are 32 bits:
  $2^{32} - 2^{29} - 1$
else if pointers are 32 bits:
  $2^{32} - 1$
else:
  $2^{48} - 1$

5.2.3 General Value Functions

General values can be assigned using the default **=** operator and compared by the usual equality operations:

```
bool operator ==
        ( min::gen g1, min::gen g2 )
bool operator !=
        ( min::gen g1, min::gen g2 )
```

The following functions return **true** if a **min::gen** datum is of the indicated subtype and **false** otherwise:

```
bool min::is_stub ( min::gen v )
bool min::is_direct_float^L ( min::gen v )
bool min::is_direct_int^C ( min::gen v )
bool min::is_direct_str ( min::gen v )
bool min::is_index ( min::gen v )
bool min::is_control_code ( min::gen v )
bool min::is_special ( min::gen v )
bool min::is_list_aux ( min::gen v )
bool min::is_sublist_aux ( min::gen v )
bool min::is_indirect_aux ( min::gen v )
bool min::is_aux ( min::gen v )
```

For a 32-bit general value **is_direct_float**^{L} is unimplemented. For a 64-bit general value **is_direct_int**^{C} is unimplemented. The **min::is_aux** function returns true if the general value is any auxiliary pointer (i.e., list, sublist, etc.).

The following protected functions return the value appropriate for a given subtype, after checking the subtype with a **MIN_ASSERT** statement:

```
      const min::stub * NULL_STUB
const min::stub * min::stub_of ( min::gen v )
     min::float64 min::direct_float_of^L ( min::gen v )
       min::int32 min::direct_int_of^C ( min::gen v )
       min::uns64 min::direct_str_of ( min::gen v )
      min::unsgen min::index_of ( min::gen v )
      min::unsgen min::control_code_of ( min::gen v )
      min::unsgen min::special_index_of ( min::gen v )
      min::unsgen min::list_aux_of ( min::gen v )
      min::unsgen min::sublist_aux_of ( min::gen v )
      min::unsgen min::indirect_aux_of ( min::gen v )
```

The **min::stub_of** function is unusual in that it can be applied to a **min::gen** value that does not contain a stub pointer, and will return **min::NULL_STUB** in this case. The latter is just a **NULL** value that has been type cast to the **const min::stub *** type. The other functions must be applied to **min::gen** values of the right kind else a **MIN_ASSERT** failure will occur.

Here the **min::uns64** value returned by **min::direct_str_of** should be overlaid by a union with a **char[]** buffer, as in the code:

```
union { min::uns64 u; char s[6]; } v;
min::gen g;
. . . set g to a direct string value . . .
v.u = min::direct_str_of ( g );
cout << v.s;
```

The **min::direct_str_of** function merely copies the 3 or 5 **char**'s of the **min::gen** direct string value and appends a **NUL** character. It does this by writing an appropriate value into **v.u**.

The **min::..._aux_of**, the **min::index_of** function, the **min::control_code_of** function, and the **min::special_index_of** function all return a 24-bit ^{C} or 40-bit ^{L} unsigned integer as a **min::unsgen** value.

The following unprotected functions return the value appropriate for a given subtype, without checking the subtype:

```
 min::stub * MUP::stub_of ( min::gen v )
min::float64 MUP::direct_float_of^L ( min::gen v )
  min::int32 MUP::direct_int_of^C ( min::gen v )
  min::uns64 MUP::direct_str_of ( min::gen v )
 min::unsgen MUP::index_of ( min::gen v )
 min::unsgen MUP::control_code_of ( min::gen v )
 min::unsgen MUP::special_index_of ( min::gen v )
 min::unsgen MUP::list_aux_of ( min::gen v )
 min::unsgen MUP::sublist_aux_of ( min::gen v )
 min::unsgen MUP::indirect_aux_of ( min::gen v )
 min::unsgen MUP::aux_of ( min::gen v )
```

Note that **MUP::stub_of** returns a '**min::stub ***' pointer whereas **min::stub_of** returns a '**const min::stub ***' pointer. Also, if the **min::gen** argument does not contain a stub pointer, the argument is illegal for **MUP::stub_of**, but it is legal for **min::stub_of** which will return **min::NULL_STUB**.

The **MUP::aux_of** function returns the auxiliary pointer of any **min::gen** value containing an auxiliary pointer (list, sublist, or indirect).

New **min::gen** values can be generated by the following protected functions:

```
min::gen min::new_stub_gen ( const min::stub * s )
min::gen min::new_direct_float_gen^L ( min::float64 v )
min::gen min::new_direct_int_gen^C ( int v )
min::gen min::new_direct_str_gen ( const char * p )
min::gen min::new_direct_str_gen ( const char * p, min::unsptr n )
min::gen min::new_index_gen ( min::unsgen i )
min::gen min::new_control_code_gen ( min::unsgen c )
min::gen min::new_special_gen ( min::unsgen i )
min::gen min::new_list_aux_gen ( min::unsgen p )
min::gen min::new_sublist_aux_gen ( min::unsgen p )
min::gen min::new_indirect_aux_gen ( min::unsgen p )
```

These protected functions check for argument range errors. Stubs are <u>not</u> allocated by these functions, so an **int** argument to **min::new_direct_int_gen**^{C} must fit in 28 bits, and the **const char *** strings must be short enough to fit into a direct string. For **min::new_direct_str_gen** with a second argument **n**, a string longer than **n** characters is shortened to **n** characters (analogously to **strncpy** and **strncmp**). The **min::unsgen** arguments used to make auxiliary pointers, indices, control codes, and special values must fit within **min::VSIZE** bits.

The subtype codes used for 64-bit **min::gen** direct string, stub pointer, auxiliary pointer, index, control code, and special values – that is, all the non-numeric **min::gen** values – are chosen to avoid being the same as the high order bits of any IEEE floating point number normally generated by the compiler, run-time system, or program execution. A **min::float64** input to **min::new_direct_float_gen** is assumed not to have these subtype codes, and no check is made for such, even by range-checking protected functions.

The following unprotected functions are analogous but do not check for range errors.

```
min::gen MUP::new_stub_gen ( min::stub * s )
min::gen MUP::new_direct_float_gen^L ( min::float64 v )
min::gen MUP::new_direct_int_gen^C ( int v )
min::gen MUP::new_direct_str_gen ( const char * p )
min::gen MUP::new_direct_str_gen ( const char * p, min::unsptr n )
min::gen MUP::new_index_gen ( min::unsgen i )
min::gen MUP::new_control_code_gen ( min::unsgen c )
min::gen MUP::new_special_gen ( min::unsgen i )
min::gen MUP::new_list_aux_gen ( min::unsgen p )
min::gen MUP::new_sublist_aux_gen ( min::unsgen p )
min::gen MUP::new_indirect_aux_gen ( min::unsgen p )
```

The following unprotected functions can be used to replace the data (non-subtype) part of a **min::gen** value that is <u>not</u> a pointer to a stub or a number (direct float or direct integer). The intended use is for changing the value of an auxiliary pointer in a general value, or the flags in a condition code.

```
min::gen MUP::renew_gen ( min::gen v, min::unsgen p )
```

The actual direct atom, stub pointer, auxiliary pointer, index, and control code **min::gen** subtype codes are implementation dependent. The following constants equal these subtype codes:

```
const unsigned min::GEN_STUB
const unsigned min::GEN_DIRECT_FLOAT^L
const unsigned min::GEN_DIRECT_INT^C
const unsigned min::GEN_DIRECT_STR
const unsigned min::GEN_LIST_AUX
const unsigned min::GEN_SUBLIST_AUX
const unsigned min::GEN_INDIRECT_AUX
const unsigned min::GEN_INDEX
const unsigned min::GEN_CONTROL_CODE
const unsigned min::GEN_SPECIAL
const unsigned min::GEN_ILLEGAL
```

min::GEN_ILLEGAL is actually illegal as a subtype code but may be returned by the following function which be used to retrieve the subtype code field:

```
unsigned min::gen_subtype_of ( min::gen v )
```

For 64-bit **min::gen** values, this retrieves the high order 24 bits of the value, and then zeros any low order bits that are not part of the subtype code (that is, the 64-bits are right shifted by 40 bits and then bits of the result that are not part of the subtype, such as bits in a number, are zeroed). For 32-bit **min::gen** values, this retrieves the high order 8 bits, and then zeros any low order bits that are not part of the subtype code (similarly the 32-bits are right shifted by 24 bits, etc.). **min::GEN_ILLEGAL** is returned by this function if the **min::gen** value is not a legal general value.

There are also unprotected functions to convert between **min::gen** and **min::unsgen** values: see p17.

5.3 Special Values

Special general values are **min::gen** values that are unique and not equal to any non-special **min::gen** value that can be generated during program execution. Special general values are C/C++ **const** values, and must only be used as specified by documentation. They are used as return or argument values by some functions.

For reasons given below, special values are defined by **inline** functions of no-arguments. Special values are divided into two groups, those that can only be used as function arguments and return values, and those whose use is unrestricted (and, in particular, can be used as object attribute values):

The special values that can only be used as function arguments and return values (and <u>cannot</u> be used as object attribute values) are:

const min::gen min::NONE()
: Denotes a non-existent function argument or result.

const min::gen min::ANY()
: An argument value used to specify that any value from a set of values may be used or returned.

const min::gen min::MULTI_VALUED()
: A value returned to indicate that a set of values has more than one element.

The special values whose use is unrestricted (and can be used as object attribute values) are:

const min::gen min::MISSING()
: Denotes a missing value or an empty set of values, and may be input or output and used as an attribute value.

const min::gen min::DISABLED()
: Specifies that a feature is disabled.

const min::gen min::ENABLED()
: Specifies that a feature is enabled.

const min::gen min::UNDEFINED()
: A value given to an object variable vector element (5.19.8 ^{p224}) that has no value and can be indirected.

`const min::gen min::UNUSED()`
A value given to an object variable vector element that is never accessed.

`const min::gen min::SUCCESS()`
A value returned to indicate a function has succeeded.

`const min::gen min::FAILURE()`
A value returned to indicate a function has failed.

`const min::gen min::ERROR()`
A value returned to indicate a function call has suffered an error. The function may print an error message in this case, or may leave an error message in **`min::error_message`** (p121).

`const min::gen min::LOGICAL_LINE()`
The value of the **`.initiator`** attribute of a logical line: see p239.

`const min::gen min::INDENTED_PARAGRAPH()`
The value of the **`.terminator`** attribute of an indented paragraph: see p239.

Special general values each have a unique ***index*** that identifies them relative to other special values. These indices are in the range 0 through $2^{24}-1$. The last 1024 indices of this range, $2^{24}-1024$ through $2^{24}-1$, are reserved for use by the MIN system. Other special values can be created by other systems, and indices near 0 are reserved for non-system users.

For efficiency reasons it is desirable for special values to be compile time constants. Because **`min::gen`** is a class type, C++ constants cannot be used to insert special values into optimized instructions, and **`inline`** functions are used instead. A special value **`S`** with index **`I`** should be defined by:

```
inline min::gen S ( void )
{
    return min::new_special_gen ( I );
}
```

where **`I`** is an integer constant in the range $0, \ldots, 2^{24}-1025$.

5.4 Stubs

General values may point at object stubs, which are 16 byte structures that are not relocated during execution. Some stubs have pointers to object bodies, which can be relocated during execution, either because the object is being expanded or contracted, or because memory is being compacted.

A stub contains an 8 byte stub value and an 8 byte stub control. The type of a stub is **`min::stub`**. Only pointers to stubs are used, and these come in two flavors: **`const min::stub *`** is used by protected functions and **`min::stub *`** without the '**`const`**' is used by unprotected functions.

If the stub control is viewed as a 64 bit integer, its high order byte is the type code. The high order bit of this type code, which is the high order bit of the 64 bit stub control integer, is off if the stub is managed by the allocator/collector/compactor (acc, of which the garbage collector is a part). In this case the stub is said to be '***collectible***'. In the other case, where the bit is on, the stub is said to be '***uncollectable***', and the stub is allocated and freed by explicit calls to the acc, but is not garbage collected or compacted by the acc.

If an stub has a body, its stub value is a pointer at that body. Any other pointer into the body is called a '***body pointer***', and will become invalid if the body is moved by a relocating function (see p12).

5.4.1 Stub Type Codes

The ***type code*** of a stub may be returned by

```
int min::type_of ( const min::stub * s )
int MUP::type_of ( const min::stub * s )
```

If the argument is `min::NULL_STUB`, the `min::type_of` function returns `0`, which is not a stub type code (it does not even equal `min::GEN_ILLEGAL`), whereas `MUP::type_of` suffers a memory fault (it tries to read a byte near address `0`).

The type code of the stub pointed at by a `min::gen` value can be obtained by

```
int min::type_of ( min::gen v )
```

This function will return `0`, which is not a legal stub type, if `v` does not contain a stub pointer.

A determination of whether or not a stub is collectible may be made by applying the function

```
bool min::is_collectible ( int type )
```

to the type code of the stub. Notice that type codes are signed integers, so that negative type codes are uncollectable and positive type codes are collectible.

A partial list of stub type codes is:

`const int min::NUMBER`
Stub value is an IEEE 64-bit floating point number.

`const int min::SHORT_STR`
Stub value is 0-8 `const char` string, `NUL` padded.

`const int min::LONG_STR`
Stub value is a pointer at a body of type `min::long_string` that contains a `const char` vector and its size.

`const int min::ACC_FREE`
Stub has just been allocated but not filled in with contents. Stubs with this

type will not be collected by the garbage collector even if they are marked by the garbage collector as dangling.

`const int min::DEALLOCATED`
Stub has a deallocated body.

`const int min::PREALLOCATED`
Stub is preallocated, and does not yet have a body or other contents. This can happen when an identifier referencing a stub via an **`min::id_map`** precedes the definition of the identified object. See 5.6.4 *p41*.

`const int min::FILLING`
Stub is preallocated that is being filled in. See 5.6.4.1 *p42*. Stubs with this type will not be collected by the garbage collector even if they are marked by the garbage collector as dangling.

A full list of stub type codes complete with page references is given on p247.

5.4.2 Stub Values

A stub contains a 64-bit ***stub value***. If the stub is collectible (as determined by its type), the type of this value is determined by the stub type code (5.4.1 *p25*). Otherwise the stub is typically attached to an object and the type of the stub value is determined by how it is attached; in this case the type of the stub value is most often just **`min::gen`**.

Many stubs are immutable and their stub values cannot be written after the stub has been created; nevertheless we describe unprotected functions below (5.7.4 *p46* and 5.7.3 *p45*) that write these values. Unprotected functions are also provided to obtain body pointers from stubs when these are the stub values of the stubs. This cannot be done by protected functions as these body pointers are relocatable and require special programming be sure they are up-to-date (5.7.6 *p51*).

5.4.3 Stub Control

A stub contains a 64-bit ***stub control***. The high order 8 bits of this is the stub type code, and the high order bit of this type code determines whether the stub is collectible (bit is off) or uncollectable (bit is on).

If the stub is collectible, the stub control is used exclusively by the acc, except for the type code, which is shared between the acc and the rest of the system. Such a control word is called an '***acc control***'. A typical (but not required) organization of an acc control is:

high order 8 bits:	type code
next 12-24 bits:	acc flags
low order 44-32 bits:	chain pointer

Here the chain pointer is a packed stub address (see p16) that is used to build lists of allocated stubs which the acc manages.

If a stub is uncollectable, its stub control, which is called a '***non-acc control***', can be organized in different ways according to the type code value. The standard way of organizing a non-acc stub control is:

high order 8 bits:	type code
next 8 bits:	subtype code
low order 48 bits:	chain pointer or unsigned integer value

Again the chain pointer is a packed stub address (see p16), but now it has enough bits to be packed with the fastest packing scheme.

A non-acc control may also be used outside a stub, say by the acc, and in this case it may be alternatively organized as:

high order 16 bits:	locator
low order 48 bits:	stub pointer

The main use of uncollectable stubs is as auxiliary stubs. An '***auxiliary stub***' is an uncollectable stub attached to an object. When the object is garbage collected, the auxiliary stub is freed. Auxiliary stubs are a means of adding memory to an object without relocating the object. For example, if the object stores 64-bit IEEE floating point numbers, a chain of auxiliary stubs can be used to add memory to the object for additional numbers. Note that the auxiliary stub itself does not contain information that tells the type of the value it stores; one has to trace the reference from the object pointing at the auxiliary stub to determine this type. Usually auxiliary stub values are **`min::gen`** values. See 5.7.4 *p*46 and 5.7.3 *p*45 for unprotected functions that can read and write auxiliary stubs.

5.5 Protected Body Pointers

Bodies are relocatable (p12) and pointers into bodies, called ***body pointers***, require special handling to ensure that they are up-to-date. There are two ways of managing this: the protected way and the unprotected way. In this section we will describe the protected way. The unprotected way is described, primarily for the benefit of those who want to implement new types of stubs, in 5.7.6 *p*51.

Note also that ***deallocation*** of a body is treated as reallocation of the body to inaccessible virtual memory.

IMPORTANT WARNING

If a protected body pointer points at a body, the body's stub must not be garbage collected, and the fact that it is being pointed at by a protected body pointer is <u>*not*</u>

sufficient to prevent this. Therefore another pointer to the body's stub must be stored elsewhere, such as in a **min::locatable_...** *variable (p35) or in another body that is protected from garbage collection, to keep the body's stub from being garbage collected.*

However, the body may be relocated without compromising the protected body pointer. Additionally, if the body is deallocated (without garbage collecting its stub, see p40), references using the protected body pointer will very likely cause a segment fault, and not an undefined result.

5.5.1 Type Specific Body Pointers

The protected way of handling body pointers uses special pointer data that is adapted to the type of datum being pointed at. For example, the following code can access any string:

```
min::gen x = . . .  // set x to some string
assert ( min::is_str ( x ) );
min::str_ptr xp ( x );
int length = min::strlen ( xp );
for ( int i = 0; i < length; ++ i )
{
    . . . xp[i] . . .
}
```

Here the **str_ptr** datum is a ***protected body pointer***. For strings that are long enough to have a body, the **str_ptr** stores a pointer to the stub of the string, and '**xp[i]**' is an inline function that expands to code that reads the body pointer from the stub and adds both an appropriate constant offset and then the index '**i**' to that pointer in order to get the address of the character.

It might be thought that this is inefficient as the body pointer is re-read from the stub for every different iteration of the '**for**' loop. However, if no out-of-line functions are called in the loop, an optimizing compiler will typically eliminate the excess reads and load the body pointer plus constant offset into a register before the loop begins. The key here is to avoid out-of-line function calls, as for each such call the optimizer must assume that the body pointer in the stub might change.

Specialized protected pointer types are provided for most types of object that have bodies.

5.5.2 Body References and Pointers

In addition to specialized pointer types for each type of object, there are a general body reference and pointer types that reference any type an element in any body associated with any stub.

5.5.2.1 Body References. The functions for creating and using `min::ref<T>` body reference types are:

```
min::ref<T> MUP::new_ref
                     ( const min::stub * s,
                       T const & location )
min::ref<T> MUP::new_ref<T>
                     ( const min::stub * s,
                       min::unsptr offset )
min::ref<T> min::new_ref
                     ( T & location )
                 where location is not relocatable

const min::stub * MUP::ZERO_STUB

const min::stub * const r.s
      min::unsptr const r.offset

min::ref<T> const & operator =
                        ( min::ref<T> const & r, T const & value )

min::ref<T> const & operator =
                        ( min::ref<T> const & r,
                          min::ref<T> const & r2 )

  T operator T ( min::ref<T> const & r )
  T operator -> ( min::ref<T> const & r )
T & operator ~ ( min::ref<T> const & r )     [unprotected]

bool operator == ( min::ref<T> const & r, T v )
bool operator == ( T v, min::ref<T> const & r )
bool operator != ( min::ref<T> const & r, T v )
bool operator != ( T v, min::ref<T> const & r )
```

In addition, the following are defined only if `T` is a packed structure or vector pointer, i.e., `min::packed_struct_xxxptr<S>` or `min::packed_vec_xxxptr<E,H,L>`:

```
bool operator == ( min::ref<T> const & r, const min::stub * s )
bool operator != ( min::ref<T> const & r, const min::stub * s )
```

A `min::ref<T>` type is similar to the '`T &`' type. Internally a `min::ref<T>` value `r` is a pointer to a location of type `T` that consists of a `const min::stub *` pointer `r.s` to a stub and an `min::unsptr` byte offset `r.offset` of a location in the body associated with the stub. Setting a reference type equal to a value of type `T` stores the value in the location. Setting a reference `r` equal to another reference `r2`, as in '`r = r2`', does <u>not</u> copy the internal pointers, but instead copies the value at the `r2` location to the `r` location.

A reference of type min::ref<T> can be explicitly converted to an relocation unprotected reference of type 'T&' by the unary '~' operator. In this case the value of the unary '~' is a reference not protected from relocation, so its only good use is to pass it to an expression or function that will not relocate anything. One must be careful that other arguments in the same statement do not perform relocation, as then with the wrong optimized order of argument evaluation '~' can be performed just before a relocation and its result used just afterwords.

A min::ref<T> value is implicitly convertible to the type T by reading the value referenced. As this implicit conversion is not activated by C++ in the case of an expression of the form 'r->...', 'operator ->' of a min::ref<T> argument is defined to invoke the conversion explicitly. Similarly for operator == and operator !=. In addition, these are defined to make the conversion when T is convertible to const min::stub * and the second argument is of this latter type, so that expressions such as

```
r == min::NULL_STUB
```

can be used.

The implicit conversion of a min::ref<T> value r to a T value is not activated by the C++ '.' operator, so if T is a structure type with member m, then r.m fails. One must use (~r).m or (&r)->m instead. The latter works because the '->' operator has been defined for min::ptr<T> values (see p32). Unfortunately C++ does not allow a similar definition for the '.' operator.

min::ref<T> values are returned by protected functions described elsewhere:

packed structures	see MIN_REF p40 and p89
packed vector headers	see MIN_REF p40 and p98
packed vector elements	see operator [] p96, push function p99
locatable variables	see operator min::ref<T> p37
object vector elements	see access functions p184, push functions p185

The unprotected MUP::new_ref function is used by these protected functions to construct a min::ref<T> value from a pointer s to a stub and either a const location within the body associated with that stub or an offset within the body (i.e., an address in bytes relative to the beginning of the body). Note that in either case, and in particular for a location of a 'T const' type, the resulting min::ref<T> reference value will allow the location to be written. This is done because locations inside data bodies that point at stubs are declared as 'const' locations so they cannot be written without using the MIN_REF macro to construct a min::ref<T> value: see p40. Note also that when an offset is given to MUP::new_ref<T>, the type T must be included in the function name, which is MUP::new_ref<T> and not just MUP::new_ref, as T cannot be deduced from the argument types.

The protected min::new_ref function may be used to construct a min::ref<T> value from a non-relocatable location, such as a global location or a location in the stack. This function constructs a **min::ref<T>** value whose stub is the special **MUP::ZERO_STUB** stub that has a body pointer equal to **0** so the offset of the **min::ref<T>** value may be the address of the non-relocatable location.

Actually there are two kinds of **min::ref<T>** types that differ according to whether or not the type **T** is '***locatable***' (**min::gen**, **const min::stub ***, and classes such as **min:packed_...ptr<...>** that encapsulate **const min::stub *** values are locatable types: see p34). If **T** is locatable, storing a value in a **min::ref<T>** location implicitly calls the **min::acc_write_update** functions of Section 5.7.1 p43 to update the stubs involved, unless the **min::ref<T>** value references **MUP::ZERO_STUB** or the value stored is **min::NULL_STUB**. If **T** is not locatable, or if the **min::ref<T>** value references **MUP::ZERO_STUB**, or if the value stored is **min::NULL_STUB**, then the **min::acc_write_update** functions are not implicitly called.

5.5.2.2 Body Pointers.

Associated with the **min::ref<T>** reference type is the companion **min::ptr<T>** pointer type:

```
min::ptr<T> MUP::new_ptr
                     ( const min::stub * s,
                       T * location )
min::ptr<T> MUP::new_ptr<T>
                     ( const min::stub * s,
                       min::unsptr offset )
min::ptr<T> min::new_ptr
                     ( T * location )
                 where location is not relocatable

(constructor) min::ptr<T>  p

const min::stub * const p.s
      const min::unsptr p.offset

min::ptr<T> & operator = ( min::ptr<T> & p, min::ptr<T> const & p2 )

            bool operator bool ( min::ptr<T> const & p )
min::ptr<T> min::null_ptr<T> ()

T * operator -> ( min::ptr<T> const & p )

min::ptr<T> operator & ( min::ref<T> const & r )
min::ref<T> operator * ( min::ptr<T> const & p )

min::ref<T> p[i]
 min::ptr<T> p+i

        T * operator ~ ( min::ptr<T> const & p )   [unprotected]
min::ptr<T> ptr<T> ( min::ptr<S> const & p )       [unprotected]
```

Internally a `min::ptr<T>` value is exactly like a `min::ref<T>` value; both point at a location defined by a stub and an offset within the body of the stub. A main difference is that `=` for reference values copies the values of the locations pointed at, while `=` for pointer values copies the pointers themselves; i.e., the pointer to the stub and the offset. Similarly `==` for reference values compares the values of the locations pointed at, whereas `==` for pointer values compares the pointers themselves.

Another difference is that if a `min::ref<T>` value `r` designates a location containing a `T` value `v`, then `r->...` is equivalent to `v->...`. But if a `min::ptr<T>` value `p` points at location `loc` containing a `T` value `v`, then `p->...` is equivalent to `(&loc)->...`. This is because the `->` operator has been defined for `min::ptr<T>` types to return `&loc`, the location designated by the pointer, but for `min::ref<T>` types it returns `v` (see p30), the location designated by the value in the location designated by the reference.

Creation of `min::ptr<T>` values by `...::new_ptr` functions is just like creation of `min::ref<T>` values by `...::new_ref` functions (see p30) with one exception. Unlike `MUP::new_ref`, for which a `const T &` location creates a `min::ref<T>` reference in which `T` is not `const`, for `MUP::new_ptr` a `T *` location creates a `min::ptr<T>` pointer in which the pointer `T` is `const` if and only if the location `T` is `const`.

The '`min::ptr<T>` p' constructor with no arguments creates `p` with `p.s = NULL_STUB` and `ps.offset = 0`, which is the ***null value*** of `min::ptr<T>`. The same value is returned by '`min::null_ptr<T>()`'. Converting a `min::ptr<T>` value to a `bool` returns `true` if and only if the pointer is not the null value.

A `min::ptr<T>` value can be created by applying the `&` operator to a `min::ref<T>` value or to a locatable variable (see `operator &` for locatable variables, p37). A `min::ref<T>` value can be recovered from a pointer value by applying the `*` operator to the `min::ptr<T>` value.

Given a `min::ptr<T>` value `p` and an index `i` of some suitable index type `I` (such as `int` or `min::uns32`), `p[i]` is a `min::ref<T>` reference value for the `i+1`'st element of the vector with elements of type `T` pointed at by `p`, and `p+i` is a `min::ptr<T>` pointer to that element.

Because `p[i]` has type `min::ref<T>` when `p` has type `min::ptr<T>`, the expression `p[i].m` fails when `T` is a structure type with member `m` (see p30). In this case it is necessary to use the alternative but usually equivalent expression `(&p[i])->m`, which is undeniably awkward.

This is an unfortunate trade off between making the the value of `p[i]` protected from relocation but harder to use when `T` is a structure type of a vector element, and the alternative of making the value of `p[i]` the type '`T &`', which would allow `p[i].m` to work properly, but which contains an unprotected relocatable pointer. Unfortunately C++ does not permit definitions of '`operator .`' in the manner that it permits definitions of '`operator ->`', or else this awkwardness would not arise.

A pointer of type `min::ptr<T>` can be explicitly converted to an relocation unprotected pointer of type '`T*`' by the unary '`~`' operator. This can be of use in code such as

```
ptr<char> s = ...;
```

```
int length = strlen ( ~ s );
char buffer[length+1];
strcpy ( buffer, ~ s );
```

In this case the value of the unary '**~**' is a pointer not protected from relocation, so its only good use is to pass it to a function such as **strlen** or **strcpy** which will not relocate anything. One must be careful that other arguments in the same statement do not perform relocation, as then with the wrong optimized order of argument evaluation '**~**' can be performed just before a relocation and its result used just afterwords.

If **p** is of type **min::ptr<S>**, then **min::ptr<T>(p)** is just **p** coerced to the type **min::ptr<T>** without any change to the actual value of **p**, just as **(T *)(q)** is a coercion of **q** if **q** has type '**S ***'. Such coercions are inherently unprotected, of course.

The following operators defined on **min::ptr<T>** values allow these values to be used to step through a vector:

```
          bool operator <
                   ( min::ptr<T> const & p1,
                     min::ptr<T> const & p2 )
min::ptr<T> operator ++                         [postfix ++]
                   ( min::ptr<T> & p, int )
min::ptr<T> operator --                         [prefix --]
                   ( min::ptr<T> & p )
```

These are just what is required to take the **min::ptr<T>** values of **min::begin_ptr_of** and **min::end_ptr_of** applied to various objects and step through the vector bracketed by the resulting pointers. See, for example, **min::begin_ptr_of** on p75 and **min::begin/end_ptr_of** on p79. To protect as much as possible from misuse, **<** is defined to be false if the two pointers point at different relocatable bodies. So one should use the pointers to access data only if the smaller pointer is still **<** than the larger pointer.

The following functions can be used to test if **min::ptr<T>** values are equal:

```
bool operator ==
         ( min::ptr<T> const & p1, min::ptr<T> const & p2 )
bool operator !=
         ( min::ptr<T> const & p1, min::ptr<T> const & p2 )
```

5.5.2.3 Pointer General Values. A **min::ptr<T>** value can be stored in a ***pointer general value*** and later retrieved from that general value. The retrieval is unprotected because **T** cannot be stored in the general value.

A pointer general value is a pointer to a stub of **min::PTR** type that holds a pointer to an auxilary stub of **min::PTR_AUX** type that contains the stub pointer and offset of a **min::ptr<T>** value. The offset is stored as a **min::unsgen** in the value of the auxilary stub, and the stub pointer is stored in the control of the auxilary stub. An auxilary stub is necessary because the stub of **min::PTR** type is garbage collectable, and the control of this stub is used by the garbage collector.

The functions for creating a pointer general value and retrieving the **min::ptr<T>** value it contains are:

```
    min::gen min::new_ptr_gen ( min::ptr<T> & p )
min::ptr<T> MUP::new_ptr<T> ( min::gen p )
```

If the **min::gen** argument to **MUP::new_ptr<T>** is not a pointer general value, **min::null_ptr<T>()** is returned.

5.5.3 Stack Temporaries of Relocatable Vectors

Sometimes it is useful to make a non-relocatable copy of a vector that is resident in a body that might be relocated. This can be done by the **MIN_STACK_COPY** macro which expands as follows:

```
MIN_STACK_COPY ( T, name, length, source )
```

expands to

```
T name [length];
memcpy ( name, ~ (source),
         sizeof ( T ) * (length) )
```

The vector at '**source**', which may be inside a body and therefore be relocatable, is copied to the stack vector '**name**', which is not relocatable. Thereafter '**name**' may be used to reference this copy. Here '**source**' should have type **min::ptr<T>**.

Because copying is fast, it can be more efficient to do this than it is to reference individual elements of a relocatable vector using **min::ptr<T>** values.

5.6 Allocator/Collector/Compactor Interface

The garbage collector needs to be able to locate all **min::gen** and **const min::stub *** values that are in use and that might contain stub pointers. Values of these types are called ***locatable values***. These types, and types that are just classes which encapsulate a **min::gen** or **const min::stub *** value (such as the **min::packed_...ptr<...>** types, 5.13^{p84} and 5.14^{p91}) are called ***locatable types***.

Values of a locatable type **T** stored in static memory or the stack need to be locatable by the garbage collector. This is done by storing them in **min::locatable_var<T>** type variables, as described in the first subsection of this section.

When you write a value of locatable type **T** into a stub or associated body, you need to call the **MUP::acc_write_update** function (5.7.1^{p43}) to update garbage collection flags associated with stubs. This is done automatically if a **min::ref<T>** reference value is used to reference the location being written. Such reference values are created by functions described in the second subsection of this section.

If you are implementing a new type of stub and maybe bodies associated with that stub type, or a new class of pointers to stubs, then you need to use the unprotected interfaces described in the third and fourth subsections of this section.

5.6.1 Locatable Variables

Values that might contain stub pointers must be locatable by the garbage collector (the collector part of the acc). Since in a compact implementation even double precision numbers are represented by **min::gen** values pointing at stubs, this means the just about all general values must be locatable. However, in order to benefit loose implementations, a distinction is made between **min::gen** values known to be numeric and those that might not be, so loose implementations can encounter less overhead when dealing with numeric **min::gen** values.

In addition to **min::gen** values that might point at stubs, **min::packed_...xxxptr<...>** values, which point at stubs (see 5.13 p84 and 5.14 p91), and **const min::stub *** values must be locatable.

Consequently locatable values of type **T** in static or stack memory must be stored in locations of type **min::locatable_var<T>**:

```
class min::locatable_var<T> :  public T
```
use **min::stub_ptr** for **T** instead of **const min::stub ***

```
typedef min::locatable_var<min::gen> min::locatable_gen
typedef min::locatable_var<min::stub_ptr> min::locatable_stub_ptr
```

In compact implementation:

```
typedef min::locatable_gen  min::locatable_num_gen
```

In loose implementation:

```
typedef min::gen  min::locatable_num_gen
```

```
(constructor) min::locatable_var<T>
                    var ( void )
(constructor) min::locatable_var<T>
                    var ( min::locatable_var<T> const & var )
(constructor) min::locatable_var<T>
                    var ( T const & value )
```

```
min::locatable_var<T> & operator =
                          ( min::locatable_var<T> & var,
                            min::locatable_var<T> const & var2 )
min::locatable_var<T> & operator =
                          ( min::locatable_var<T> & var,
                            T const & value )

min::ref<T const> operator min::ref<T const>
                    ( min::locatable_var<T> const & var )
      min::ref<T> operator min::ref<T>
                    ( min::locatable_var<T> & var )
min::ptr<T const> operator &
                    ( min::locatable_var<T> const & var )
      min::ptr<T> operator &
                    ( min::locatable_var<T> & var )
```

The name **min::locatable_gen** should be used instead of the equivalent **min::locatable_var<min::gen>**. If the implementation is loose (p14), **min::locatable_num_gen** is equivalent to **min::gen**. If the implementation is compact, it is equivalent to **min::locatable_gen**.

Normally **min::locatable_var<T>** is only used with **T = min::packed_...ptr<...>** (see 5.13[p84] and 5.14[p91]). It cannot be used with **T = const min::stub *** as this is not a class type and cannot be a base type for the locatable vector type. Instead use **T = min::stub_ptr**, which is a class whose only member is a **const min::stub *** value that can be read and written using standard operations:

```
(constructor) min::stub_ptr
                     var ( void )
(constructor) min::stub_ptr
                    var ( const min::stub * s )

min::locatable_var<T> & operator const min::stub *
                          ( min::stub_ptr const & var )
        min::stub_ptr & operator =
                          ( min::stub_ptr & var,
                            const min::stub * s )
```

The main operations that can be performed on a **min::locatable_var<T>** location are to copy a value of type **T** into it or out of it. Setting one **min::locatable_var<T>** location equal to another copies the value of the locations, and not the location structure itself. Setting a **min::locatable_var<T>** location to a value **v** of type **T** value writes the value to the location. A **min::locatable_var<T>** locatable variable has its **T** value as its base class, most other operations on the **T** value can be performed on the **min::locatable_var<T>** variable.

A **min::locatable_var<T>** locatable variable is also implicitly convertible to a reference of type **min::ref<T>** that references the value stored in the locatable variable. Note that if the locatable variable is **const**, the reference type will be **min::ref<const T>**, which does not allow the variable to be written.

Applying the **&** operator to a **min::locatable_var<T>** locatable variable returns a pointer of type **min::ptr<T>** that points at the variable value. Note that if the locatable variable is **const**, the pointer type will be **min::ptr<const T>**, which does not allow the value to be written.

An example using **min::locatable_gen** is:

```
static min::locatable_gen v;
. . . . . . . . . . .
void f ( ref<min::gen> const & r );
. . . . . . . . . . .
min::gen some_function ( min::gen x, min::gen y )
{
    min::locatable_gen q = x;
    min::ptr<min::gen> p = & q;
    min::ref<min::gen> r = q;
    . . .
    v = q;      // Copies the min::gen value of q to v.
    * p = y;    // Sets q to y.
    r = x;      // Sets q to x.
    . . .
    f ( v );    // Converts v to min::ref<min::gen>
    f ( q );    // Converts q to min::ref<min::gen>
    . . .
    return q;   // Returns the value of q.
}
```

An example using **min::locatable_var<T>** with **T = min::packed_vec_insptr<char>** (see 5.14 [p91]) is:

```
typedef min::packed_vec_insptr<char> bufvec;
min::locatable_var<bufvec> x;
void f ( min::ref<bufvec> p );
... some_function ( ... )
{
    min::locatable_var<bufvec> y = x;
    f ( y );  // Converts y to a min::ref<bufvec> reference.
}
```

Importantly a **min::gen** or **const min::stub *** value need <u>not</u> be stored in a locatable variable if it can be located by the garbage collector by some other means. For example, if it is stored in one locatable variable, it need not be stored in another. Or if a **min::gen** or **const min::stub *** value V is stored in an object pointed at by another **min::gen** or **const min::stub *** value P, and P is locatable, then V need not be stored in a locatable variable.

A very important rule is that when a function is called, the caller must be sure every **min::gen** or **const min::stub *** value passed to the called function can be located by the garbage collector <u>without</u> the called function needing to store the value in a locatable variable. For example, the caller can store the value in its own locatable variable. This is called the '***caller locating convention***'.

Also **min::gen** and **const min::stub *** values need only be locatable by the garbage collector when a relocating function (p12) is called. In between such calls **min::gen** and **const min::stub *** values can be stored in other places. In particular, there is no problem returning these values from a called function to its caller.

5.6.2 Locatable Member References

Suppose we have some kind of stub/body data type whose body is the structure **S** defined and used in the following:

```
struct S;
struct S_ptr
    // Pointer to a stub whose body is a datum of type S.
{
    const min::stub * s;

    S_ptr ( const min::stub * s ) : s ( s ) {}

    S * operator -> ( void )
    {
        return (S *) MUP::ptr_of ( s );
            // Return pointer to body.
    }

    operator const min::stub * ( void ) const
    {
        return s;
            // Return pointer to stub.
    }
};

struct S
```

```
{
    // Examples of different member types:
    //
    int x;
    min::gen g;
    S_ptr p;
};

void foo ( S_ptr q )
{
    // Examples of usage of members:
    //
    int x2 = q->x.
    q->x = ...;
    min::gen g2 = q->g;
    q->g = ...;
    MUP::acc_write_update ( q, q->g );
    S_ptr p2 = q->p;
    q->p = ...;
    MUP::acc_write_update ( q, q->p );
}
```

Because **`min::gen`** and **`S_ptr`** are locatable types, **`MUP::acc_write_update`** (5.7.1 ^{p43}) must be called when members of these types are written into a stub or associated body. There is a danger of unintentional omission of these calls.

An alternative is to replace some of the above code by the following:

```
struct S
{
    int x;
    const min::gen g;    // Added `const'.
    const S_ptr p;       // Added `const'.
};

inline min::ref<min::gen> g_ref ( S_ptr q )
{
    return MUP::new_ref ( q, q->g );
}

inline min::ref<S_ptr> p_ref ( S_ptr q )
{
    return MUP::new_ref ( q, q->p );
}
```

```
void foo ( S_ptr q )
{
    int x2 = q->x.
    x = ...;
    min::gen g2 = q->g;
    g_ref(q)= ...;        // q->g replaced by g_ref(q).
    S_ptr p2 = q->p;
    p_ref(q) = ...;       // q->p replaced by p_ref(q).
}
```

Here the calls to **MUP::acc_write_update** are inside the **..._ref(q) =** calls (p31). Writing **q->g = ...** and **q->p = ...** is prevented by making the **g** and **p** members '**const**', so accidentally updating locatable members without calling **MUP::acc_write_update** is prevented.

To make the above easier to code, the **MIN_REF** macro is provided which expands as follows:

```
MIN_REF ( type, name, ctype )
```

expands to

```
inline min::ref<type> name_ref ( ctype container )
{
    return MUP::new_ref ( container, container->name );
}
```

Here '**type**' is the member type and '**ctype**' is the '***container type***'.

This enables the **inline** function definitions above to be replaced by

```
MIN_REF ( min::gen, g, S_ptr )
MIN_REF ( S_ptr, p, S_ptr )
```

The **MIN_REF** macro is frequently used with packed structures (see 5.13 *p84*) and packed vectors (see 5.14 *p91*).

5.6.3 Deallocation

The operation of ***deallocating a body*** is considered to be a relocation of the body. The body pointer in the stub is pointed at a 'deallocated body' located in inaccessible virtual memory, and the type code in the stub is set to **min::DEALLOCATED** (p13).

The function that deallocates a body is:

```
void min::deallocate R ( const min::stub * s )
```

If the stub has a body, this function relocates the body to inaccessible memory and changes the type code of the stub to **min::DEALLOCATED**. Otherwise the function does nothing; and in particular, it does nothing to stubs with no body, and to stubs that have already been deallocated.

The function that tests whether a stub has **min::DEALLOCATED** type code (p13) is:

```
bool min::is_deallocated ( const min::stub * s )
```

The inaccessible memory to which a **min::DEALLOCATED** stub is pointed is called the '***deallocated body***' of the stub. This is large enough that any attempt to access the body of a deallocated stub will cause a memory fault.

5.6.4 Preallocated Stubs

A stub can be ***preallocated***, referenced, and even garbage collected, before its contents are known. Such stubs have the **PREALLOCATED** type. Functions for this type are:

```
  min::gen min::new_preallocated_gen ( min::uns32 id )
      bool min::is_preallocated ( min::gen g )
min::uns32 min::id_of_preallocated ( min::gen g )
min::uns32 min::count_of_preallocated ( min::gen g )
          void::increment_preallocated ( min::gen g )
```

The **min::new_preallocate_gen** function returns a **min::gen** value pointing at a new stub of **min::PREALLOCATED** type that stores the given **id**. The preallocated stub represents an object or non-identifier string value whose contents are yet unspecified. The preallocated stub can be referenced as is, and functions such as **min::new_obj_gen** (p175) and **min::copy** (p179) can be used to fill in the final contents of the stub.

Preallocated stubs store two **min::uns32** values: an identifier and a count. When a preallocated stub is created from an identifier input from a file, an entry is made in an identifier map (5.16 ^{p117}) associating the identifier with the preallocated stub, and the identifier is recorded in the stub. The count of the preallocated stub is initialized to **1** when the stub is created. The count can be incremented; this should be done when the same identifier is read again from the file. Then the count tells how many times the identifier has been read from the file.

It is possible to assign a non-object value to an identifier: see the **min::put** function in 5.16 ^{p117}. When this is done for an identifier associated with a preallocated object, the preallocated object ends up dangling. The identifier count can be used to tell if this is an error. Typically if the count is **1**, the only reference to the preallocated object in the input file is the one being used to set the identifier value, and there is no error, but if the count is greater than **1**, some previous reference in the input file will be left pointing at the now dangling preallocated object. However, this is <u>not</u> an Allocator-Collector-Compactor error; it is an application error.

If the object represented by the preallocated stub never has its contents filled in, the ID recorded in the stub can be used in debugging to find the references to the preallocated stub in the input file.

The functions above create preallocated stubs, test a **`min::gen`** value to see if it points at a preallocate stub, return the stub's identifier and count, and increment the count. If **`g`** does not point at a preallocated stub, `0` is returned as its identifier or count.

5.6.4.1 Filling Preallocated Stubs.

Filling a preallocated stub with contents requires care in order to interact correctly with the garbage collector. The preallocated stub is itself subject to garbage collection, and so must be protected against such collection.

The algorithm for filling a preallocated stub is as follows. Unless otherwise indicated, the garbage collector may <u>not</u> interrupt this algorithm.

1. Protect from garbage collection any stubs that will be pointed at by the new contents of the preallocated stub. For example, if the contents of an object X are being copied to the preallocated stub, protecting X with **`min::locatable_gen`** (p35) suffices.
2. Verify that the stub has **`min::PREALLOCATED`** type (and so has not been collected).
3. Change the stub type to **`min::FILLING`** with **`MUP::set_type_of`** (p46). This will keep the garbage collector from collecting the stub while it is being filled.
4. Fill the stub with contents. You may call **`MUP::new_body`** (p49) to allocate a body for the stub. The garbage collector may interrupt this step.
5. Execute **`MUP::acc_write_update`** (5.7.1 ^{p43}) for any pointer stored in the new contents (for objects see p187). The garbage collector may interrupt this step.
6. Change the stub's type to its final value with **`MUP::set_type_of`**.

Note that preallocated stubs may be garbage collected as soon as their type is changed from **`min::FILLING`** in Step 6, as preallocated stubs are usually visible to the garbage collector. If the preallocated stub is not already protected from garbage collection, then to protect it you will need to set some protected variable, such as a **`min::locatable_gen`** variable, to point at the preallocated stub between the last step of the algorithm and the time the garbage collector might next run.

It is also possible to move the contents of a second stub to a preallocated stub. Specifically, if the value of a datum with a stub and body is to be moved to a preallocated stub, the algorithm is as above with the following modifications:

Step 1 Do <u>not</u> protect the stubs that are pointed at by the body to be moved. Instead, inhibit garbage collector interrupts during the <u>entire</u> algorithm.

Step 2 Leave as is.

Step 3 Omit this step, as the garbage collector cannot run during this algorithm.

Step 4 Use **`MUP::move_body`** (p50) to move the body pointer from the second stub to the preallocated stub and set the body pointer of the second stub to point at a block of inaccessible memory.

Step 5 Leave as is, except do not permit the garbage collector to interrupt.

Step 6 Copy the type from the second stub to the preallocated stub, and then set the type of the second stub to **`min::DEALLOCATED`**.

5.7 Implementing New Stub Types

If you are implementing a new type of stub, and maybe bodies associated with that stub type, or a new class of pointers to stubs, then you will probably need to use the unprotected interfaces described in this section.

As described in the first subsection, you need to be able update stubs using the **`MUP::acc_write_update`** functions when locatable values are written into stubs or associated bodies. You need to be able to allocate and free stubs, as described in the second subsection. You need to be able to read and write stub values, as described in the third subsection. If you are using acc stubs you need to be able to write the type field in stub controls, and if you are using auxiliary stubs, you need to be able to read, write, and manipulate entire stub controls. This is described in the fourth subsection.

5.7.1 ACC Write Update Functions

In addition to needing to locate **`min::gen`** and **`min::stub *`** values, the acc must be notified whenever a pointer to a collectible stub **`s2`** is stored in the data of a collectible stub **`s1`** (i.e., in the stub **`s1`** or its body or auxiliary stubs attached to either). This is done by the following functions:

```
void MUP::acc_write_update
            ( const min::stub * s1,
              const min::stub * s2 )
void MUP::acc_write_update
            ( const min::stub * s1,
              min::gen g )
void MUP::acc_write_num_update
            ( const min::stub * s1,
              min::gen g )
```

```
void MUP::acc_write_update
             ( const min::stub * s1,
               const min::stub * const * p, min::unsptr n )
void MUP::acc_write_update
             ( const min::stub * s1,
               const min::gen * p, min::unsptr n )
void MUP::acc_write_num_update
             ( const min::stub * s1,
               const min::gen * p, min::unsptr n )
```

The first of the above functions updates the stubs **s1** and **s2** if **s2** is not **NULL**. The second updates the stub **s1** and the stub pointed at by **g** if **g** points at a stub. The third of the above functions equals the second function for a compact (p14) implementation, and is a no-operation for a loose implementation. The stubs updated must be collectable (i.e., they must not be auxiliary stubs).

The last three of the above functions use all the values **p[0]**, ..., **p[n-1]** as a second argument in a call to the corresponding function chosen from the first three of the above functions.

The **min::acc_write_update** function on p187 is similar to these last functions, but takes all the pointers stored in an object as second arguments.

The **min::acc_write_num_update** functions should only be used if all the **min::gen** values they reference are numeric, as in a loose implementation these functions do nothing.

These **min::acc_write_update** functions must not be called with any argument that points at a stub with **min::ACC_FREE** type, which can only happen when the stub has been recently returned by **min::new_acc_stub**. See 5.7.2 p44.

5.7.2 Stub Allocation Functions

The following functions are used to allocate stubs:

```
min::stub * MUP::new_acc_stub ( void )
min::stub * MUP::new_aux_stub ( void )
```

The **min::new_acc_stub** function returns a garbage collectible (acc) stub with its type set to **min::ACC_FREE**. A stub with this type must not be visible to the garbage collector; no pointer to it can be stored in an acc locatable variable. The **MUP::acc_write_update** function must not be called with any **min::gen** or **min::stub *** argument that points at a stub with **min::ACC_FREE** type. All **min::gen** or **min::stub *** values stored in the data of a **min::ACC_FREE** stub must also be stored in acc locatable variable, or must themselves point at **min::ACC_FREE** stubs. The **MUP::new_body** and **MUP::resize_body** functions (5.7.5 p49) may be called to fill a **min::ACC_FREE** stub, and these may invoke the garbage collector.

The actions of changing the type of a stub from **min::ACC_FREE** to another collectible type and storing a **min::gen** or **min::stub *** value pointing at the stub in an acc locatable variable must not be separated by any call to a relocating function.

It is permissible to allocate and build a graph of stubs some of which have **min::ACC_FREE** type. No stub locatable by the garbage collector may contain (in its stub or body or auxiliary stubs) a pointer to a stub with **min::ACC_FREE** type. Any stub with **min::ACC_FREE** type can contain only pointers to stubs of this type or stubs that are locatable by the garbage collector. After building the graph, the types of all stubs with **min::ACC_FREE** type should be changed to normal acc types, and pointers that permit the garbage collector to locate these stubs should be stored in locatable variables, all without making any calls to relocating functions.

The **min::new_aux_stub** function returns a non-acc (i.e., not garbage collectible) stub with its type set to **min::AUX_FREE**. This kind of stub is is not freed by the garbage collector. It may be freed only by calling:

```
void MUP::free_aux_stub ( min::stub * s )
```

A non-acc stub is most often attached to an acc stub in such a way that when the acc stub is garbage collected, the non-acc stub is freed. Such a non-acc stub is called an ***auxiliary stub***, and because most non-acc stubs are of this kind, functions dealing with non-acc stubs have names containing '**aux**' instead of '**non_acc**'.

5.7.3 Stub Value Read/Write Functions

The following unprotected functions read or write the stub value part of a stub:

```
  min::uns64 MUP::value_of ( const min::stub * s )
min::float64 MUP::float_of ( const min::stub * s )
    min::gen MUP::gen_of ( const min::stub * s )
      void * MUP::ptr_of ( const min::stub * s )
        void MUP::set_value_of ( min::stub * s, min::uns64 v )
        void MUP::set_float_of ( min::stub * s, min::float64 f )
        void MUP::set_gen_of ( min::stub * s, min::gen v )
        void MUP::set_ptr_of ( min::stub * s, void * p )
```

Thus the stub value can be taken to be of type **min::uns64**, **min::float64**, **min::gen**, or of some pointer type.

These functions do not check type codes, call **MUP::acc_write_update**, or check that values read or written are within legal range for a particular stub. For example, a stub value that is not a floating point number can be read by **MUP::float_of** with undefined results.

5.7.4 Stub Control Functions

If you are using auxiliary stubs, you need to be able to read, write, and manipulate the stub control part of the stub. If you are using acc stubs, you only need to read or write the type part of the control.

The following read or write the stub control:

```
min::uns64 MUP::control_of ( const min::stub * s )
       int min::type_of ( const min::stub * s )
      bool MUP::test_flags_of
                    ( const min::stub * s,
                      min::uns64 flags )
      void MUP::set_control_of ( min::stub * s, min::uns64 c )
      void MUP::set_type_of ( min::stub * s, int type )
      void MUP::set_flags_of ( min::stub * s, min::uns64 flags )
      void MUP::clear_flags_of ( min::stub * s, min::uns64 flags )
```

The **MUP::control_of** and **MUP::set_control_of** functions deal with the entire 64 bit stub control value of a stub. The other functions deal only with parts.

For acc stubs, only the type part of the control should be read or written. The other parts of the control are for use by the acc, and should not be accessed by non-acc code, in order to ensure that the acc is independent of other code.

The stub control in a stub is an example of a MIN ***control value***. A MIN control value holds a stub address or an unsigned integer in its low order bits. It may hold an 8 bit type code in its highest order bits. Any bits left over are flag bits, or if there is no 8 bit type code, the high order 16 bits may be a signed integer field called the '***locator***' which is used only by some acc code. Control values have type **min::uns64**. Control values are used as stub controls, and may be use in other places, e.g., by the acc to hold pointers from a block that holds a body back to the stub pointing at the body.

Thus control values contain an address/value low order field, and optional type code high order field, and flag bits. There are two kinds of control values: (ordinary) control values and acc control values:

Ordinary control value with type:

Bits	Contents
63-56	int8 type
55-48	8 flag bits
47-0	unsigned integer value or absolute stub address

Ordinary control value with locator:

Bits	Contents
63-48	int16 locator
47-0	unsigned integer value or absolute stub address

Acc control value:

Bits	Contents
63-56	int8 type
55-(56-G)	G flag bits
(55-G)-0	absolute or packed stub address

where $8 \leq G \leq 24$.

The larger G, the more acc flags, which may permit the garbage collector to be more efficient (e.g., to have more ephemeral levels).

The packed stub address is an absolute stub address, relative stub address, or stub index. The possible packing schemes used for these are the same as the packing schemes used for general values: see p16. However, the actual packing scheme used for acc control values may differ from the actual packing scheme used for general values, because the number of bits available for the packed stub address may differ in the two cases.

Addresses stored in a control value must be stub addresses, as only they can be packed into less than 64 bits.

The control values used as stub controls do have a type code field which can be read by **`min::type_of`** (which is protected) and written by **`MUP::set_type_of`**. For stub controls, acc control values are used with collectible types, and ordinary control values with uncollectable types.

The flag bits are set, cleared, and tested individually. They are defined by constants of type **`min::uns64`**, such as

`const min::uns64 MUP::STUB_ADDRESS`
: Indicates that the address/value field of an ordinary (non-acc) control holds a stub address. This flag is only used for uncollectable stubs whose control address/value field might be either a stub address or an unsigned integer.

These flag constants are defined by expressions of the form

`( min::uns64(1) << ` K ` )`

The above functions assume that any flag constants select bits in a control value that are not inside the address/value field or inside the type code field.

Ordinary (non-acc) control values that have a locator field cannot have any flag bits.

The flags in the stub control value of a stub can be tested, set, or cleared by some of the above functions. The **`MUP::test_flags_of`** function returns true if and only if the logical AND of the **flags** arguments and the flags in the stub's control is non-zero. The **MUP::set_flags_of** function sets one or more individual flags by logically ORing its argument into the stub's control, and the **MUP::clear_flags_of** function clears flags by logically ANDing the complement of its argument into the stub's control.

The above functions do not check type codes, nor do they check that values read or written are within legal range for a particular stub. Thus a stub control value can be written by **MUP::set_control_of** even if the written control datum is incompatible with the garbage collector implementation, and may produce undefined results when the garbage collector next executes.

The high order byte of any control written by **MUP::set_control_of** is the type code, and the high order bit is clear if the stub is collectible and set if the stub is uncollectable (5.4.3 *p26*). Changing a stub from collectible to uncollectable or vice versa requires removing or adding the stub to garbage collector lists that are threaded through the pointer field of the stub control. So one cannot simply change the type code field of a stub from collectible to uncollectable or vice versa.

Ordinary (non-acc) control values can be manipulated by the following functions:

```
min::uns64 MUP::new_control
                    ( int type_code, min::uns64 v,
                                     min::uns64 flags = 0 )
min::uns64 MUP::new_control_with_type
                    ( int type_code, const min::stub * s,
                                     min::uns64 flags = 0 )
min::uns64 MUP::new_control_with_locator
                    ( int locator, const min::stub * s )

min::uns64 MUP::renew_control_locator ( min::uns64 c, int locator )
min::uns64 MUP::renew_control_value ( min::uns64 c, min::uns64 v )
min::uns64 MUP::renew_control_stub
                    ( min::uns64 c, const min::stub * s )

        int MUP::locator_of_control ( min::uns64 c )
 min::uns64 MUP::value_of_control ( min::uns64 c )
min::stub * MUP::stub_of_control ( min::uns64 c )
```

The '**new**' functions compute a control value, the '**renew**' functions modify control values by inserting a new locator, value, or stub address, and the other functions return the parts of a control value. Here the 16-bit locator is represented as an **int** in the range from -2^{15} through $2^{15}-1$.

None of these functions check the ranges of their arguments.

Acc control values can be manipulated by the following similar functions (this is only done by acc code):

```
 min::uns64 MUP::new_acc_control
                    ( int type_code, const min::stub * s,
                                     min::uns64 flags = 0 )
 min::uns64 MUP::renew_acc_control_stub
                    ( min::uns64 c, const min::stub * s )
min::stub * MUP::stub_of_acc_control ( min::uns64 c )
```

Either ordinary or acc control values can be manipulated by the following functions:

```
min::uns64 MUP::renew_control_type ( min::uns64 c, int type )
       int MUP::type_of_control ( min::uns64 c )
```

Here the 8-bit type code is represented as an **int** in the range from −128 through 127.

5.7.5 Unprotected Body Functions

If you are implementing a new kind of stub, and your stubs have bodies, you need to be able to allocate, relocate, and deallocate the bodies. The following functions allocate and deallocate bodies:

```
void MUP::new_body ( min::stub * s, min::unsptr n )
void MUP::deallocate_body ( min::stub * s, min::unsptr n )
```

Here **n** is the size in bytes of the body to be allocated or deallocated. The allocator is not required to remember the size of a body, so when deallocating the body the caller must provide the same size as was used to allocate the body. The allocator will likely run a check that will likely catch a wrong size, but it may not be able to determine the right size.

Bodies are always aligned on 8 byte boundaries, but the size **n** need not be a multiple of 8.

When a body is deallocated, the stub type is set to **min::DEALLOCATED** and the stub pointer is set to point at a block of inaccessible memory. As a special case, if the body size **n** is zero, **MUP::deallocate_body** does nothing (see **MUP::body_size_of** below).

The function

```
min::unsptr MUP::body_size_of ( const min::stub * s )
```

returns the size of the body, that is, the same size as that passed to **MUP::new_body** when the body was allocated. This is necessary as the acc does not keep track of body sizes, and depends upon this **MUP::body_size_of** function to find body sizes. If the stub is deallocated (of type **min::DEALLOCATED**) or if the stub has no body, **0** is returned. This enables the code

```
MUP::deallocate_body ( s, MUP::body_size_of ( s ) )
```

The **min::body_size_of** function uses the type of the stub to select a type-appropriate algorithm to compute the body size. The code of this function must be modified if a new stub type is added.

The function

```
void * & MUP::ptr_ref_of ( min::stub * s )
```

returns a pointer to a pointer to the body. The first pointer points to a location P with a fixed address (P is the stub value); P holds a pointer to the body, which may be relocated by a call to a relocating function (p12). Whenever the body is relocated the value of P is changed to point to the new body location.

A body pointer can be moved from one stub to another as part of the algorithm for filling preallocated stubs described at the end of 5.6.4.1 *p42*. The function which does this is:

```
void MUP::move_body ( min::stub * s1, min::stub * s2 )
```

This function requires **s2** to have a body pointer, which it moves to **s1**, and then it sets the body pointer of **s2** to point at a block of inaccessible memory. Note that to move a body pointer between stubs also requires stub flags to be adjusted and pointers from the body back to its stub to be adjusted, all of which this function does. Also calls to **min::acc_write_update** must be made for **s1** and pointers stored in the body after this function is called and before the garbage collector can be allowed to interrupt. This function does not change the types of **s1** or **s2**. Setting the type of **s2** to **min::DEALLOCATED** after calling this function is recommended.

In order to change the size of a body, the following can be used:

```
(constructor) MUP::resize_body rb
                  ( min::stub * s,
                    min::unsptr new_size, min::unsptr old_size )
      void * & MUP::new_body_ptr_ref ( MUP::resize_body & rb )
           void MUP::abort_resize_body ( MUP::resize_body & rb )
           void MUP::retype_resize_body
                  ( MUP::resize_body & rb, int new_type )
```

When constructed the **MUP::resize_body** datum allocates a new body for a stub **s**, and when deconstructed the datum installs the new body in the stub **s** while deallocating the old body of **s**. Stub **s** is not altered until the **MUP::resize_body** datum is deconstructed, and the **MUP::abort_resize_body** function can be used to abort the body resizing and prevent stub **s** from ever being altered. The sizes of the old and new body of **s** must be passed to the **MUP::resize_body** constructor.

Given a **MUP::resize_body** datum **rb**,

```
MUP::new_body_ptr_ref ( rb )
```

returns a pointer to a pointer to the new body. The first pointer points to a location P with a fixed address that exists as long as the **MUP::resize_body** datum **rb** exists. P holds a pointer to the new body, which may be relocated by a call to a relocating function (p12) as all bodies can be, and will be pointed to by a changed value of P if it is relocated.

After the new body is obtained, information should be copied from the old body to the new body before the **MUP::resize_body** datum is deconstructed. The new body will not be touched by the garbage collector while the **MUP::resize_body** datum exists, but it may be relocated. The existing stub **s** must be protected from garbage collection and its body protected from reorganization by the user while the **MUP::resize_body** datum exists, but that body may also be relocated. **s** must NOT be deallocated while the **MUP::resize_body** datum exists, unless **MUP::abort_resize_body** has been called.

While the **MUP::resize_body** datum exists, the garbage collector will process the stub and old body normally. Any locatable values stored in the new body must also be stored in a separate garbage collector locatable place, as the garbage collector ignores the new body.

Normally the type of stub **s** is not changed, but if the **MUP::retype_resize_body** function is called with a new stub type, that type will be installed in stub **s** if and when the new body is installed in stub **s**.

5.7.6 Unprotected Body Pointers

Unprotected body pointers are C/C++ pointers that point directly into a body. Functions that obtain body unprotected pointers from stubs are unprotected (**MUP**) functions because the unprotected body pointers they return are invalidated if the body pointed at is relocated.

In order to track possible relocation, names of functions that might relocate bodies are marked with the superscript ^{R} in this document. Relocation can only happen inside such ***relocating functions*** (p12).

The following uses unprotected body pointers to point at a character string stored in a body:

```
min::gen x = . . .  // set x to some long string
assert ( min::is_stub ( x ) );
const min::stub * xstub = min::stub_of ( x );
assert ( min::type_of ( xstub ) == min::LONG_STR );
MUP::long_str * xstr = MUP::long_str_of ( xstub );
const char * xp = MUP::str_of ( xstr );
int length = MUP::length_of ( xstr );
for ( int i = 0; i < length; ++ i )
{
    // Relocating functions must NOT be called in this loop.
    . . . xp[i] . . .
}
```

Strings are in fact of three types, 'direct' which stores characters in the general value, 'short' which stores up to 8 characters in the stub value, and 'long' which stores more than 8 characters in the body. The above code only works for long strings (p78).

5.8 UNICODE Characters

Unicode characters are implemented by the following:

```
typedef min::uns32 min::Uchar
  const min::Uchar min::UNKNOWN_UCHAR
  const min::Uchar min::SOFTWARE_NL
  const min::Uchar min::NO_UCHAR = 0xFFFFFFFF
        min::Uchar min::utf8_to_unicode
                       ( const char * & s, const char * ends )
          unsigned min::unicode_to_utf8
                       ( char * & s, min::Uchar c )
```

Here UNICODE characters are represented as **min::Uchar** values that are 32-bit unsigned integers. '**char ***' strings are treated as 'Modified UTF-8' encodings of UNICODE character strings. In these the NUL (zero) UNICODE **min::Uchar** value is encoded as the 2 string bytes **"\xC0\x80"**, while the zero '**char**' is used to terminate the '**char ***' string. The term '***Modified UTF-8***' applies to a UTF-8 encoding in which the only permitted overlong encoding is this 2-byte encoding of NUL.

The **UNKNOWN_UCHAR** UNICODE character is returned by the **min::utf8_to_unicode** function when it finds an illegal UTF8 encoding. This is actually the '***UNICODE replacement character***', FFFD (hexadecimal), which is designated by the UNICODE standard as representing an input that is not encodable in UNICODE.

The **SOFTWARE_NL** UNICODE character is used to represent the end of line in a **min::file**, which is internally represented by a line-ending NUL character, when the end of line is displayed on a printer. See **min::DISPLAY_EOL** on p130. This is actually an arbitrary UNICODE private use character, and is not normally input or output itself.

The **min::uft8_to_unicode** function reads a UNICODE character encoded as a UTF-8 byte string. Here **s** points at the first byte of the encoding and is updated to point after the encoding, while **ends** points just after the last byte that can be part of the encoding.

The **min::uft8_to_unicode** function returns **min::NO_UCHAR** if **s >= ends** when the function is called. Otherwise in returns the first character encoded by the UTF-8 string pointed at by **s** and terminating just before **ends**, and updates **s** to point just after that character's encoding.

The **min::uft8_to_unicode** function will accept ***overlong*** UTF-8 encodings as legal, and also accept 7-byte encodings (first byte **0xFE**) so that $2^{32}-1$ can be encoded. If it encounters an illegal encoding it returns **min::UNKNOWN_UCHAR**. An illegal encoding will terminate if the next byte cannot be part of a legal encoding or if there is no next byte according to **ends**.

Applying the **min::uft8_to_unicode** function repeatedly until **s >= ends** will produce a sequence of UNICODE characters from any byte string, even one that is not legal UTF-8, though if the string is not legal some **min::UNKNOWN_UCHAR** characters will be returned.

The **min::unicode_to_utf8** function writes a UTF-8 character **c** into **s**, updating **s** to point after the character, and returns the number of bytes written. This function will output 7-byte encodings (first byte **0xFE**) for UNICODE character values **=> 0x80000000**, and will output the overlong encoding **"\xC0\x80"** for **NUL**. UTF-8 in which this is the only permitted overlong encoding is called 'Modified UTF-8' (p52). No other overlong encodings are output. At most 7 bytes will be output, and **s** must point to a byte string buffer with at least 7 bytes.

Strings of UNICODE characters can be converted to/from strings of UTF-8 encoded characters by the functions:

```
min::unsptr min::utf8_to_unicode
                     ( min::Uchar * & u, const min::Uchar * endu,
                       const char * & s, const char * ends )
min::unsptr min::unicode_to_utf8
                      ( char * & s, const char * ends,
                        const min::Uchar * & u,
                        const min::Uchar * endu )
```

The first of these repeatedly executes

```
* u ++ = min::utf8_to_unicode ( s, ends );
```

while **u < endu** and **s < ends**. The second repeatedly executes

```
min::unicode_to_utf8 ( s, * u ++ );
```

while **s < ends** and **u < endu** provided the next repetition will not end with **s > ends** (so the **s** string end will not be overrun). Both these functions increment both **s** and **u** and return the number of string elements, **min::Uchar**'s or **char**'s respectively, written.

5.8.1 Unicode Data Base

Various attributes of UNICODE characters are stored in the ***UNICODE Data Base***.

To keep this data base compact, UNICODE characters are grouped so all characters with the same attributes are assigned the same group. This is implemented by mapping each character to a ***UNICODE character index***, and then mapping the character index to various attributes. To enable a program to load only parts of the database, each attribute is in a separately loadable vector.

The first part of this mapping is implemented by

```
min::uns16 min::Uindex ( min::Uchar c )
```

which maps the UNICODE character **c** to its UNICODE character index **Uindex (c)**.

Given the index **i = min::Uindex (c)** of **c**, you can find out the name and picture of **c** by using:

```
min::ustring min::unicode::name[i]
min::ustring min::unicode::picture[i]
```

Almost all UNICODE characters have just one name, but a few have more than one. The following table describes the extra names:

```
struct       min::unicode::extra_name
{
     min::ustring name
     min::Uchar   c
}
min::unicode::extra_name min::unicode::extra_names
                         min::uns32 min::unicode::extra_names_number
```

Here each of the **`min::unicode::extra_names_number`** entries in **`min::unicode::extra_names`** givens an extra **`name`** for a character **`c`**. Extra names are useful when a character is input by name: see 5.8.6 p68.

You can find out the character flags of **`c`** as described below in 5.8.2 p55. So that ASCII and LATIN1 characters can each have their own flags, each has its own index. That is:

```
min::Uindex ( c ) == c      if c <= 0xFF
min::Uindex ( c )  > 0xFF   if c  > 0xFF
```

For indices **`i`** for which there is exactly one character that has index **`i`**, you can find out that character by using:

```
const min::Uchar min::unicode::character[i]
const min::uns16 min::unicode::index_limit
```

As no two characters share a name or picture, this last can be useful for finding the character that has a given name or picture. If **`i`** is associated with exactly one character, **`min::unicode::character[i]`** equals that character, and otherwise it equals **`min::NO_CHAR`**.

The size of all the above vectors is **`min::unicode::index_limit`**, which is a strict upper bound on the index **`i`** (i.e., **`0 <= i < index_limit`**).

Here '**`min::ustring`**' values encode UNICODE character strings in a format optimized for printing, or are **`NULL`** to indicate a missing value; see 5.8.5 p67 for details. The ***name*** is non-missing only for control characters; examples are the name **`CR`** for the ASCII carriage return character with (hex) code **`0D`**, **`SP`** for the ASCII single space character with (hex) code **`20`**, and **`NBSP`** for the LATIN1 non-breaking single space character with (hex) code **`A0`**. LATIN1 control characters (including horizontal spaces) and some other control characters have non-missing names.

The ***picture*** is non-missing for ASCII control characters, and for these is the UNICODE control picture character corresponding to the ASCII control character name. For example, the line feed picture is ␊, which is the single UNICODE character with code **`240A`**. As a special case, single space has the picture ␣ and the non-breaking single space (**NBSP**) has the picture ␢.

Also as special cases the **min::UNKNOWN_UCHAR** character has name **UUC** and picture ///, and the **min::SOFTWARE_NL** character has name **NL** and picture $^{N}{}_{L}$.

The **ustring** name and picture values contain only graphic characters (e.g., no spaces) and have non-zero numbers of columns encoded in their second bytes.

5.8.2 Unicode Character Flags

The most important attributes of a character are its ***flags***. Given the index of **c**:

```
i = min::Uindex ( c )
```

you can find out the flags of **c** by using:

```
const min::uns32 printer->print_format.char_flags[i]
  const min::uns32 * min::standard_char_flags
```

where usually

```
printer->print_format.char_flags == min::standard_char_flags
```

Character flags are obtained from the **char_flags** member of the **print_format** member of the **printer** being used to print the character, and this in turn is usually set to the **min::standard_char_flags** vector. Unlike other character attributes, some of these flags can be computed by user code in a flexible manner, so some of the standard character flags which we describe here are only a special case of what is possible.

As mentioned above (p54), each ASCII and LATIN1 character has its own **char_flags** element, so each can be assigned flags independently of the flags of any other character.

The standard character flags which must always be defined are

```
const min::uns32 min::IS_GRAPHIC
const min::uns32 min::IS_CONTROL
const min::uns32 min::IS_UNSUPPORTED

const min::uns32 min::IS_NON_GRAPHIC =
                       min::IS_CONTROL
                     + min::IS_UNSUPPORTED

const min::uns32 min::IS_HSPACE
const min::uns32 min::IS_VHSPACE

const min::uns32 min::IS_NON_SPACING
```

The specific assignment of character flags is generally done according to their UNICODE '***General Category***' with some exceptions. The following are the possible General Category values. These have the form `Xy` where `X` denotes the category and `y` denotes the subcategory. We use the notation `X` to represent the category `X` including all its subcategories, and we give here only some of the subcategories of each category. We also give the LATIN1 characters in each subcategory except where these are obvious (e.g, letters and digits).

`L`		letters
	`Ll`	lower case letters, includes `µ`
	`Lu`	upper case letters
	`Lo`	other letters, includes `º` and `ª`
`M`		combining marks
	`Mn`	non-spacing combining marks
	`Me`	enclosing combining marks
`N`		number
	`Nd`	decimal numbers, includes digits
	`No`	other numbers, includes `¹ ² ³ ¼ ½ ¾`
`P`		punctuation
	`Pc`	connector punctuation, includes `_`
	`Pd`	dash punctuation, includes `-`
	`Ps`	open punctuation, includes `{ [ (`
	`Pe`	close punctuation, includes `} ] )`
	`Pi`	initial punctuation, includes `«`
	`Pf`	final punctuation, includes `»`
	`Po`	other punctuation, includes `! " # % & ' * , . / : ; ? @ \ ¡ § ¶ · ¿`
`S`		symbol
	`Sm`	math symbol, includes `+ < = > | ~ ¬ ± × ÷`
	`Sc`	currency symbol, includes `$ ¢ £ ¤ ¥`
	`Sk`	modifier symbol, includes `^ ` ¨ ¯ ´ ¸`
	`So`	other symbol, includes `¦ © ® °`
`Z`		space characters
	`Zs`	non-zero width horizontal space, includes single space and `NBSP` (but not horizontal tab)
`C`		control characters
	`Cc`	LATIN1 control codes in ranges [`00`...`1F`] and [`7F`...`9F`], includes horizontal tab
	`Cf`	format control character, includes `SHY` (soft hyphen)

Each character gets exactly one of the following flags:

```
min::IS_GRAPHIC
min::IS_CONTROL
min::IS_UNSUPPORTED
```

The **min::IS_GRAPHIC** flag signifies that a character makes some visible mark.

The **min::IS_CONTROL** flag signifies that a character makes no visible mark.

The **min::IS_UNSUPPORTED** flag signifies that the character cannot be output (except as a numeric character code) and that no information about the character is available.

The assignment of these three flags in **min::standard_char_flags** is:

min::IS_GRAPHIC:	characters in categories **L**, **M**, **N**, **P**, and **S**
min::IS_CONTROL:	characters in categories **C** and **Z**
min::IS_UNSUPPORTED:	characters not given any category (or given a new category not mentioned above)

The support control part of a print format

```
printer->print_format.support_control
```

permits characters with particular flags to be marked unsupported in a give context; so for example, all non-ASCII or all non-LATIN1 characters can be marked unsupported so they will not be output as is (useful if the printer does not support them). See 5.8.3 p64 and p131.

Horizontal space characters get the **min::IS_HSPACE** flag. Included is the horizontal tab **HT**.

Vertical space characters that standardly appear in printed output get the **min::IS_VHSPACE** flag. There are four such characters: form feed **FF**, vertical tab **VT**, line feed **LF**, and carriage return **CR**. The last is included because it is associated with line feeds. Characters with the **IS_HSPACE** flag also get the **IS_VHSPACE** flag.

The assignment of these two flags in **min::standard_char_flags** is:

min::IS_HSPACE:	**HT**, plus characters in subcategory **Zs**
min::IS_VHSPACE:	**FF VT LF CR**, plus characters with the **IS_HSPACE** flag

The **min::IS_NON_SPACING** flag signifies that a character takes zero columns, instead of 1 column. The characters given this flag are the non-spacing combining mark characters and all control characters (including vertical spacing characters) other than horizontal spacing characters. Note that the horizontal tabs are handled specially by the printer as far as the number of columns they take, and this special handling ignores the character flags of the horizontal tab character.

The assignment of this flag in **min::standard_char_flags** is:

min::IS_NON_SPACING:	characters with the **IS_CONTROL** flag, but without the **IS_HSPACE** flag, plus characters in subcategories **Mn** and **Me**

Other character flags are used by the printer context data and can be adapted according to the application. The following flags are defined by **min::standard_char_flags**:

```
const min::uns32 min::IS_SP
const min::uns32 min::IS_BHSPACE

const min::uns32 min::CONDITIONAL_BREAK

const min::uns32 min::IS_LEADING
const min::uns32 min::IS_TRAILING

const min::uns32 min::IS_SEPARATOR
const min::uns32 min::IS_REPEATER
const min::uns32 min::NEEDS_QUOTES

const min::uns32 min::IS_DIGIT
const min::uns32 min::IS_NATURAL
const min::uns32 min::IS_LETTER
const min::uns32 min::IS_MARK

const min::uns32 min::IS_ASCII
const min::uns32 min::IS_LATIN1
```

The **min::IS_SP** is just for the single space character. The **min::IS_BHSPACE** flag is for 'breaking' horizontal space characters. These flags can be used in

printer->print_format.break_control.break_after

to mark characters after which automatic line breaks can be inserted (and then the spaces at the end of the broken line are deleted). See p131.

The assignment of these two flags in **min::standard_char_flags** is:

min::IS_SP:	SP, the single space character
min::IS_BHSPACE:	all characters with the IS_HSPACE flag, except NBSP (LATIN1 no-break space) and NNBSP (narrow no-break space, character code 0x202F)

The **min::CONDITIONAL_BREAK** flag is used in

printer->print_format.break_control.conditional_break

to mark graphic characters which when preceded by a few graphic characters and followed by a line break indicate that their containing lexeme is continued on the next line. See p132.

The assignment of this flag in **min::standard_char_flags** is:

`min::CONDITIONAL_BREAK`:	`/ # &` plus characters in categories `Pc` and `Pd` (includes `-` and `_`)

The `min::IS_LEADING` or `min::IS_TRAILING` flags are used to mark characters that can be leading separators prepended to other lexemes or trailing separators appended to other lexemes. The rough concept is that a string of characters with no spaces may be separated into lexemes by stripping leading separators from its beginning and trailing separators from its end, so '`[5;`' becomes the leading separator lexeme `[`, the middle lexeme `5`, and the trailing separator lexeme `;`, and therefore `[` is `IS_LEADING`, `5` is neither `IS_LEADING` nor `IS_TRAILING`, and `;` is `IS_TRAILING`. See `min::standard_str_classifier` (5.8.4[p64]) and Leading and Trailing Separators 5.17.3[p145] for details.

The assignment of these two flags in `min::standard_char_flags` is as follows:

`min::IS_LEADING`:	`‘ ¡ ¿ |` plus categories `Ps` and `Pi` which include `{ ( [ «`
`min::IS_TRAILING`:	`’ ! ? | . , ; :` plus categories `Pe` and `Pf` which include `} ) ] »`

Note that '`|`' is <u>both</u> `IS_LEADING` and `IS_TRAILING`.

The `min::IS_SEPARATOR` flag is used by `min::standard_str_classifier` to identify characters that cannot be combined with other characters in unquoted strings; e.g. `(`, `)`, and `|`. The `min::IS_REPEATER` flag is used by `min::standard_str_classifier` to identify characters that may be repeated when they appear in unquoted separators; e.g. `|`. The `min::NEEDS_QUOTES` flag is used by `min::standard_str_classifier` to identify graphic characters that require any string containing them to be quoted; e.g., `"`. See 5.8.4[p64] for details.

The assignment of these three flags in `min::standard_char_flags` is as follows:

`min::IS_SEPARATOR`:	`|` and categories `Ps`, `Pi`, `Pe`, `Pf`, which include `{ ( [ « » ] ) }`
`min::IS_REPEATER`:	`| ‘ ¡ ¿ ’ ! ? . :`
`min::NEEDS_QUOTES`:	`"`

The following flags serve to identify various types of lexemes. A lexeme that has a character with the `min::IS_DIGIT` flag before any character with the `min::IS_LETTER` flag is a '***number***. A number consisting of just characters with the `min::IS_NATURAL` flag is a '***natural number***', unless it begins with '`0`' and has more than one digit. A lexeme that has a character with the `min::IS_LETTER` flag before any character with the `min::IS_DIGIT` flag is a '***word***. A lexeme all of whose characters have the `min::IS_MARK` flag is a '***mark***'. See `min::standard_str_classifier` (5.8.4[p64]) and the `mark_classifier` of the object format, p236.

The assignment of these flags in `min::standard_char_flags` is as follows:

min::IS_DIGIT:	category **Nd** (digits)
min::IS_NATURAL:	the characters **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, and **9**
min::IS_LETTER:	category **L** (letters)
min::IS_MARK:	categories **P** and **S** (punctuation and symbols), but not characters with the **IS_SEPARATOR** or **NEEDS_QUOTES** flags

The last two flags are used in

printer->print_format.support_control.support_mask

to dynamically control which characters are supported. The **min::IS_ASCII** flag is on for ASCII characters, and The **min::IS_LATIN1** flag is on for LATIN1 non-ASCII characters. See 5.8.3 p64 for details.

The assignment of these two flags in **min::standard_char_flags** is:

min::IS_ASCII:	ASCII characters with codes in range [**00**...**7F**]
min::IS_LATIN1:	LATIN1, non-ASCII characters with codes in range [**80**...**FF**]

The question of which characters can be supported is determined by the file

.../**min/unicode/CompositeCharacters.txt**

which lists the multi-key sequences that can be used to input non-ASCII supportable characters, and also by the file

.../**min/unicode/SupportSets.txt**

which defines which sets of characters are marked with which support set character flags, such as **min::IS_ASCII/LATIN1**.

The non-control characters with the **min::IS_LATIN1** flag in **min::standard_char_flags** are listed, along with the multi-key sequences commonly used to input them, in Figures 1, 2, and 3. Printers can usually be configured to support these LATIN1 characters.

min::standard_char_flags is computed using data taken from .../**min/unicode/unicode_data.h** (which is included in the **min::unicode** namespace by the **min_unicode.h** file that is in turn included in **min.h**). Other character flag vectors can be computed using this data.

	Hex Code	Multi-Key Sequence	
	A0	<space><space>	No Break Space (**NBSP**)
¡	A1	!!	Inverted Exclamation
¢	A2	c|	Cent Sign
£	A3	L-	Pound Sign
¤	A4	ox	Currency Sign
¥	A5	Y=	Yen Sign
¦	A6	!^	Broken Bar
§	A7	S!	Section Sign
¨	A8	""	Diaeresis
©	A9	oc	Copyright Sign
ª	AA	^_a	Feminine Ordinal Indicator
«	AB	<<	Left-Pointing Double Angle Quotation Mark
¬	AC	-,	Not Sign
	AD	-<space>	Soft Hyphen (**SHY**)
®	AE	or	Registered Sign
¯	AF	^-	Macron
°	B0	oo	Degree Sign
±	B1	+-	Plus-Minus Sign
²	B2	^2	Superscript 2
³	B3	^3	Superscript 3
´	B4	''	Acute Accent
µ	B5	mu	Micro Sign
¶	B6	p!	Pilcrow Sign
·	B7	^.	Middle Dot
¸	B8	,,	Cedilla
¹	B9	^1	Superscript 1
º	BA	^_o	Masculine Ordinal Indicator
»	BB	>>	Right-Pointing Double Angle Quotation Mark
¼	BC	14	Vulgar Fraction One Quarter
½	BD	12	Vulgar Fraction One Half
¾	BE	34	Vulgar Fraction Three Quarters
¿	BF	??	Inverted Question Mark

Figure 1: LATIN1 Characters: Part I

	Hex Code	Multi-Key Sequence	
À	C0	A`	Latin Capital Letter A With Grave
Á	C1	A'	Latin Capital Letter A With Acute
Â	C2	A^	Latin Capital Letter A With Circumflex
Ã	C3	A~	Latin Capital Letter A With Tilde
Ä	C4	A"	Latin Capital Letter A With Diaeresis
Å	C5	AA	Latin Capital Letter A With Ring Above
Æ	C6	AE	Latin Capital Letter AE
Ç	C7	C,	Latin Capital Letter C With Cedilla
È	C8	E`	Latin Capital Letter E With Grave
É	C9	E'	Latin Capital Letter E With Acute
Ê	CA	E^	Latin Capital Letter E With Circumflex
Ë	CB	E"	Latin Capital Letter E With Diaeresis
Ì	CC	I`	Latin Capital Letter I With Grave
Í	CD	I'	Latin Capital Letter I With Acute
Î	CE	I^	Latin Capital Letter I With Circumflex
Ï	CF	I"	Latin Capital Letter I With Diaeresis
Ð	D0	DH	Latin Capital Letter ETH
Ñ	D1	N~	Latin Capital Letter N With Tilde
Ò	D2	O`	Latin Capital Letter O With Grave
Ó	D3	O'	Latin Capital Letter O With Acute
Ô	D4	O^	Latin Capital Letter O With Circumflex
Õ	D5	O~	Latin Capital Letter O With Tilde
Ö	D6	O"	Latin Capital Letter O With Diaeresis
×	D7	xx	Multiplication Sign
Ø	D8	O/	Latin Capital Letter O With Stroke
Ù	D9	U`	Latin Capital Letter U With Grave
Ú	DA	U'	Latin Capital Letter U With Acute
Û	DB	U^	Latin Capital Letter U With Circumflex
Ü	DC	U"	Latin Capital Letter U With Diaeresis
Ý	DD	Y'	Latin Capital Letter Y With Acute
Þ	DE	TH	Latin Capital Letter Thorn
ß	DF	ss	Latin Small Letter Sharp S

Figure 2: LATIN1 Characters: Part II

	Hex Code	Multi-Key Sequence	
à	E0	a`	Latin Small Letter A With Grave
á	E1	a'	Latin Small Letter A With Acute
â	E2	a^	Latin Small Letter A With Circumflex
ã	E3	a~	Latin Small Letter A With Tilde
ä	E4	a"	Latin Small Letter A With Diaeresis
å	E5	aa	Latin Small Letter A With Ring Above
æ	E6	ae	Latin Small Letter AE
ç	E7	c,	Latin Small Letter C With Cedilla
è	E8	e`	Latin Small Letter E With Grave
é	E9	e'	Latin Small Letter E With Acute
ê	EA	e^	Latin Small Letter E With Circumflex
ë	EB	e"	Latin Small Letter E With Diaeresis
ì	EC	i`	Latin Small Letter I With Grave
í	ED	i'	Latin Small Letter I With Acute
î	EE	i^	Latin Small Letter I With Circumflex
ï	EF	i"	Latin Small Letter I With Diaeresis
ð	F0	dh	Latin Small Letter ETH
ñ	F1	n~	Latin Small Letter N With Tilde
ò	F2	o`	Latin Small Letter O With Grave
ó	F3	o'	Latin Small Letter O With Acute
ô	F4	o^	Latin Small Letter O With Circumflex
õ	F5	o~	Latin Small Letter O With Tilde
ö	F6	o"	Latin Small Letter O With Diaeresis
÷	F7	-:	Division Sign
ø	F8	o/	Latin Small Letter O With Stroke
ù	F9	u`	Latin Small Letter U With Grave
ú	FA	u'	Latin Small Letter U With Acute
û	FB	u^	Latin Small Letter U With Circumflex
ü	FC	u"	Latin Small Letter U With Diaeresis
ý	FD	y'	Latin Small Letter Y With Acute
þ	FE	th	Latin Small Letter Thorn
ÿ	FF	y"	Latin Small Letter Y With Diaeresis

Figure 3: LATIN1 Characters: Part III

5.8.3 UNICODE Support Control

The inline function:

```
inline min::uns32 min::char_flags
                        ( const min::uns32 * char_flags,
                          min::support_control sc,
                          min::Uchar c )
```

returns the ***character flags*** **cflags** of the UNICODE character **c** according to the algorithm:

```
min::uns16 cindex = min::Uindex ( c );
min::uns32 cflags = char_flags[cindex];
if ( ( cflags & sc.support_mask ) == 0 )
    cflags = sc.unsupported_char_flags;
return cflags;
```

The function uses the **char_flags** vector (p55) and a support control structure **sc** with the format:

```
struct     min::  support_control
{
     min::uns32   support_mask
     min::uns32   unsupported_char_flags
}

const min::uns32 min::ALL_CHARS = 0xFFFFFFFF
const min::support_control min::ascii_support_control =
              { min::IS_ASCII, min::IS_UNSUPPORTED };
const min::support_control min::latin1_support_control =
              { min::IS_LATIN1 + min::IS_ASCII, min::IS_UNSUPPORTED };
const min::support_control min::support_all_support_control =
              { min::ALL_CHARS, min::IS_UNSUPPORTED };
```

Both **char_flags** and **sc** are typically elements of **printer->print_format** for the printer being used.

5.8.4 String Classifiers

A ***string classifier*** is a function that inputs a string and outputs a set of flags that can be used to determine whether the string must be quoted, if it is a separator and whether it is leading or trailing, and if the string is a mark that can be used as a type.

```
typedef min::uns32 ( * min::str_classifier )
                            ( const min::uns32 * char_flags,
                              min::support_control sc,
                              min::unsptr n,
                              min::ptr<const min::Uchar> p )
```

```
const min::uns32 min::IS_BREAKABLE
```

```
const min::str_classifier min::standard_str_classifier
const min::str_classifier min::quote_separator_str_classifier
const min::str_classifier min::quote_value_str_classifier
const min::str_classifier min::quote_all_str_classifier
const min::str_classifier min::null_str_classifier
```

The flags returned by the string classifier that can be used by the printer to make printing decisions as follows:

NEEDS_QUOTES The string needs to be quoted.

For **min::standard_str_classifier** this is returned if any of the following are true:

1. The string contains any character with the **NEEDS_QUOTES** flag.
2. The string contains a character that does not have the **IS_GRAPHIC** flag.
3. The string has a format acceptable to `strtod`. Specifically, the string has one of the following syntaxes:
   $\{\texttt{+}|\texttt{-}\}^?\ \mathit{digit}^+\ \{\texttt{.}\mathit{digit}^\star\}^?\ \{\ \{\texttt{e}|\texttt{E}\}\ \{\texttt{+}|\texttt{-}\}^?\ \mathit{digit}^+\ \}^?$
   $\{\texttt{+}|\texttt{-}\}^?\ \texttt{.}\mathit{digit}^+\ \{\ \{\texttt{e}|\texttt{E}\}\ \{\texttt{+}|\texttt{-}\}^?\ \mathit{digit}^+\ \}^?$
   $\{\texttt{+}|\texttt{-}\}^?\ \{\texttt{n}|\texttt{N}\}\ \{\texttt{a}|\texttt{a}\}\ \{\texttt{n}|\texttt{N}\}$
   $\{\texttt{+}|\texttt{-}\}^?\ \{\texttt{i}|\texttt{I}\}\ \{\texttt{n}|\texttt{N}\}\ \{\texttt{f}|\texttt{F}\}$
4. All of the following are true:
   (a) The first string character has the **IS_LEADING** flag, or the last string character has the **IS_TRAILING** flag, or any string character has the **IS_SEPARATOR** flag
   (b) there is more than one character in the string
   (c) not all characters in the string are identical or the string character does not have the **IS_REPEATER** flag.

IS_BREAKABLE The string can be broken into parts if it is too long to fit on one line, as per **min::print_breakable_unicode** (165). This flag is meaningful only if the **NEEDS_QUOTE** flag is not returned.

For **min::standard_str_classifier** this is returned if the all of the following are true:

1. All the string characters have the `IS_GRAPHIC` flag.
2. The string contains a character with either the `IS_LETTER` or `IS_DIGIT` flag, or all the characters of the string have the `IS_MARK` flag (i.e., the string is a word, number, or mark).
3. No string character has the `IS_SEPARATOR` flag.
4. No string character has the `NEEDS_QUOTES` flag.
5. The first string character does not have the `IS_LEADING` flag.
6. The last string character does not have the `IS_TRAILING` flag.

`IS_GRAPHIC` Unless the `NEEDS_QUOTES` flag is present, the string should be printed unquoted and may be immediately preceded by leading separators or immediately followed by trailing separators. Separators also have this flag.

For **`min::standard_str_classifier`** this is returned for strings all of whose characters have the `IS_GRAPHICS` flag.

`IS_SEPARATOR` The string is a strict separator.

For **`min::standard_str_classifier`** this is returned for strings all of whose characters have the `IS_SEPARATOR` flag.

`IS_LEADING` The string is a leading separator.

For **`min::standard_str_classifier`** this is returned for strings all of whose characters have the `IS_LEADING` flag.

`IS_TRAILING` The string is a trailing separator.

For **`min::standard_str_classifier`** this is returned for strings all of whose characters have the `IS_TRAILING` flag.

`IS_DIGIT` The string is a number.

For **`min::standard_str_classifier`** this is returned for strings that have a character with the `IS_DIGIT` flag before any character with the `IS_LETTER` flag.

`IS_NATURAL` The string is a natural number.

For **`min::standard_str_classifier`** this is returned for strings all of whose characters are have the `IS_NATURAL` flag (indicating they are ASCII digits) and which either do not begin with ‘`0`’ or consist of the single digit ‘`0`’.

IS_LETTER The string is a word.

For **min::standard_str_classifier** this is returned for strings that have a character with the **IS_LETTER** flag before any character with the **IS_DIGIT** flag.

IS_MARK The string is a mark, and as such can be used as the type of a bracketed expression that can be output in compact form.

For **min::standard_str_classifier** this is returned for strings all of whose characters have the **IS_MARK** flag.

All these flags are also used in **min::standard_char_flags** (p58) except **min::IS_BREAKABLE**.

The **min::quote_separator_str_classifier** is just like the **min::standard_str_classifier** except that it adds the **min::NEEDS_QUOTES** flag if the result has any of the **min::IS_SEPARATOR**, **min::IS_LEADING**, or **min::IS_TRAILING** flags.

The **min::quote_value_str_classifier** is just like the **min::standard_str_classifier** except that it adds the **min::NEEDS_QUOTES** flag if the result does not have either the **min::IS_LETTER** flag or the **min::IS_DIGIT** flag.

The **min::quote_all_str_classifier** always returns just the **min::NEEDS_QUOTES** flag.

The **min::null_str_classifier** is used when printing with a **min::str_format** that is **NULL** and returns 0 if all characters in the string have the **IS_VHSPACE** flag, and just **min::IS_GRAPHIC** otherwise.

The following function is used by the **min::standard_str_classifier**:

`bool min::`**`is_number`**` ( min::unsptr n, min::ptr<const min::Uchar> p )`

Returns true iff the UNICODE `n` character string pointed at by `p` is acceptable to `strtod`. This means that it has one of the following syntaxes:

$$\{\texttt{+}|\texttt{-}\}^?\ digit^+\ \{\texttt{.}digit^\star\}^?\ \{\ \{\texttt{e}|\texttt{E}\}\ \{\texttt{+}|\texttt{-}\}^?\ digit^+\ \}^?$$
$$\{\texttt{+}|\texttt{-}\}^?\ \texttt{.}digit^+\ \{\ \{\texttt{e}|\texttt{E}\}\ \{\texttt{+}|\texttt{-}\}^?\ digit^+\ \}^?$$
$$\{\texttt{+}|\texttt{-}\}^?\ \{\texttt{n}|\texttt{N}\}\ \{\texttt{a}|\texttt{a}\}\ \{\texttt{n}|\texttt{N}\}$$
$$\{\texttt{+}|\texttt{-}\}^?\ \{\texttt{i}|\texttt{I}\}\ \{\texttt{n}|\texttt{N}\}\ \{\texttt{f}|\texttt{F}\}$$

5.8.5 UNICODE Strings

Constant ***UNICODE strings*** can be represented as **min::ustring** values which are sequences of 8 bit bytes. The first two bytes, called the ***header***, hold the length of the string (not counting the header) in the first byte, and the number of print columns the string will take if the string is all graphics, in the second byte. The remainder of the bytes constitute a '**const char ***' value that is the Modified UTF-8 encoded, NUL-terminated, string itself.

Importantly **ustring** values are pointers into unrelocatable memory. It is intended that **ustrings** be used as parametric values in printing formats, as character names and pictures, etc.

The following are for programming with **ustring**'s:

```
typedef const min::uns8 * min::ustring
              min::uns32 min::ustring_length ( min::ustring s )
              min::uns32 min::ustring_columns ( min::ustring s )
            const char * min::ustring_chars ( min::ustring s )
```

The **min::ustring_length** function returns the number of **char**'s in the **const char *** string within the **ustring** (not including the terminating **NUL**); the maximum is 63. The **min::ustring_columns** function returns the number of print columns these take; the maximum is 63.

5.8.6 UNICODE Name Tables

A ***UNICODE name table*** can be used to look up a UNICODE character given one of its names. For most characters the name is that given by **min::unicode::name** (p54), but other names may be associated with characters by **min::unicode::extra_names** (p54) or by the `min::add` function immediately below.

UNICODE name tables have the type

(type) `min::unicode_name_table`

whose implementation is hidden. The tables may be created and read using the following functions:

```
min::unicode_name_table min::init R
    ( min::ref<min::unicode_name_table> table,
      const min::uns32 * char_flags =
                min::standard_char_flags,
      min::uns32 flags = min::ALL_CHARS,
      min::uns32 extras = 10 )

      void min::add R
                    ( min::unicode_name_table table,
                      const char * name,
                      min::Uchar c,
                      bool replace_allowed = false )
min::Uchar min::find
                    ( min::unicode_name_table table,
                      const char * name)
```

The **min::init** function creates a hash table if the **table** argument equals **min::NULL_STUB**, and sets this argument to the new hash table. If the table already exists, the function and does nothing.

If the table is created, the **min::unicode::name** and **min::unicode::extra_names** of a character is added to the table if the character's flags, as given by the **char_flags** argument, have a flag in common with the **flags** argument. Additionally extra space is reserved for later adding character names to the table, as long as the number of names added later does not exceed the value of the **extras** argument. The default arguments to **min::init** add all names in the **min::unicode::name** data base and reserve space for adding 10 additional names.

The **min::add** function adds **name** to the table, mapping it to the character **c**. It is a programming error if **name** is already mapped to a different character and the **replace_allowed** argument is **false**, or if more names are added than allowed by the **extras** argument to **min::init** (this latter error may not be detected and may result in inefficient lookup). If **replace_allowed** is **true** and **name** is already mapped to a different character, the table is changed so **name** maps to **c**.

The **min::find** function looks up **name** in the table and returns the character it maps to. **min::NO_UCHAR** is returned if the name is not in the table.

5.9 Numbers

A ***number stub*** is collectible, has **min::NUMBER** stub type code, and has an immutable **min::float64** stub value that can be read by

```
min::float64 min::float_of C ( const min::stub * s )
```

Number stubs exist only in compact implementations; in loose implementations number atoms are stored exclusively in direct number general values (5.2.1 ^{p15}).

General values that are numbers can be tested, created, and read by the following protected functions:

```
        bool min::is_num ( min::gen v )

    min::gen min::new_num_gen R ( int v )
    min::gen min::new_num_gen R ( min::unsptr v )
    min::gen min::new_num_gen R ( min::float64 v )

  min::int64 min::int_of ( min::gen v )
min::float64 min::float_of ( min::gen v )

  min::uns32 min::numhash ( min::gen v )
```

The **min::is_num** function for a loose 64-bit **min::gen** argument is just another name for the **min::is_direct_float** function of the same argument. For a compact 32-bit **min::gen** argument the function returns true if the argument is a direct integer or a stub pointer pointing at a number stub.

The **min::new_num_gen** function with **min::float64** argument and loose 64-bit **min::gen** value is just another name for **min::new_direct_gen**, which simply changes the type of its argument. The **min::new_num_gen** function with **min::float64** argument and compact 32-bit **min::gen** value creates a direct integer if the argument is an integer in the require range; otherwise the function returns a **min::gen** value that is a pointer to a number stub. If a pointer to a number stub is to be returned and a number stub containing the argument value already exists, a pointer to the existing stub is returned; otherwise a new number stub is created and a pointer to it returned. Therefore two 32-bit **min::gen** values that represent equal numbers are themselves **==**.

The **min::new_num_gen** function with an **int** or **min::unsptr** argument does the same thing as it would with its argument converted to a **min::float64** value, but is more efficient in the case where **min::gen** values are 32 bits and the argument is in the range of a direct integer general value.

The **min::float_of** function for a loose 64-bit **min::gen** argument is just another name for the **min::direct_float_of** function of the same argument, which after checking the subtype of the argument, simply changes the type of the argument. For a compact 32-bit **min::gen** argument the function returns any integer stored directly in the argument converted to a 64-bit IEEE floating point number, or returns the stub value for any number stub pointed at by the argument. In this last case the stub type code is checked by a **MIN_ASSERT** statement to be sure the stub is a number stub.

The **min::int_of** function does the same thing as the **min::float_of** followed by conversion to a **min::int64** value, except that **min::int_of** includes a check that the result is a pure integer, without any fractional part, and is within the range of the **min::int64** type, and **min::int_of** is more efficient when given a 32-bit direct integer **min::gen** argument.

Lastly, the **min::numhash** function returns the hash value of a **min::gen** value that is a number. This value is computed by considering the **min::float_of** value of the number to be a big endian string of 8 characters and using the algorithm on p74 to compute the hash value of this string.

To permit hash values of arbitrary floating point numbers to be computed, the following function is provided:

```
min::uns32 min::floathash ( min::float64 f )
```

The above **min::float_of**, **min::int_of**, and **min::numhash** functions of a **min::gen** argument apply **MIN_ASSERT** to check that their argument is a number. The following unprotected function assumes that its argument is a number without doing any **MIN_ASSERT** check:

```
min::float64 MUP::float_of ( min::gen v )
```

5.10 Strings

In MIN all ***strings*** are **NUL** terminated UTF-8 encoded UNICODE character strings. ***UTF-8*** encodes 32-bit UNICODE characters in 1 to 7 **char**'s.

All ASCII characters are encoded as themselves in the UTF-8 encoding. This implies that all ASCII character strings are UTF-8 encoded character strings with the same characters as their ASCII representation indicates.[4]

It is possible for a string to be miscoded UTF-8. Unless otherwise mentioned, the functions given below, including the protected functions, do not check for this.

A MIN string value cannot store the **NUL** character as legal UTF-8. But it is possible to use the '***modified UTF-8***' encoding instead of strict UTF-8. The difference is that strict UTF-8 encodes the NUL character in a single byte as an ASCII NUL, whereas modified UTF-8 encodes the **NUL** character as the 'overlong' 2-byte string '**0xC0,0x80**' (which is not legal in strict UTF-8 because it is not the shortest possible encoding of **NUL** in UTF-8).

There are two kinds of string stubs: short strings and long strings. In addition, a string of up to 3 bytes can be stored within a 32-bit **min::gen** value, and a string of up to 5 bytes can be stored within a 64-bit **min::gen** value, without using a stub (see 5.2.1 ^{p15} and 5.2.3 ^{p19}). Such strings are called ***direct strings***, while strings stored in stubs or bodies which are pointed at by **min::gen** values, the short and long strings, are called ***indirect strings***. A short string holds up to 8 bytes inside the string stub (there must be more bytes than a direct string will hold). A long string has an string body that holds the string bytes (there must be more than 8) along with the string length and hash value.

Sufficiently short strings are ***identifier strings***. The variable:

`min::unsptr min::`**`max_id_strlen`** [default **32**]

is the maximum length of an identifier string in bytes when encoded in UTF-8 (not the same as the length in UNICODE characters). Although this parameter variable is writable, it should not be reset after any string of length between the new and old value has been created, and it should never be set less than **8**.

All three kinds of strings, direct, short, and long, are immutable. All identifier strings have hash values (p83). An identifier string can be compared for equality with any other string by using just the **==** or **!=** operators, but two non-identifier strings cannot be so compared. Identifier strings are names (5.12 ^{p83}) and can be components of labels (5.11 ^{p79}); non-identifier strings are not names can cannot be components of labels.

[4] ASCII character codes range from 0 through 127. UTF-8 extends this by assigning meaning to codes from 128 to 255.

5.10.1 Protected String Functions

There are protected functions accessing general values that denote strings of any kind without distinction, and these are described next. Unprotected functions that apply only to particular types of string are described later in Section 5.10.3 p78.

The following functions create new general string values:

```
min::gen min::new_str_gen R ( const char * p )
min::gen min::new_str_gen R ( const char * p, min::unsptr n )
min::gen min::new_str_gen R ( min::ptr<const char> p )
min::gen min::new_str_gen R ( min::ptr<const char> p, min::unsptr n )
min::gen min::new_str_gen R ( min::ptr<char> p )
min::gen min::new_str_gen R ( min::ptr<char> p, min::unsptr n )
```

The **min::new_str_gen** functions copy the input string after the manner of **strcpy** and **strncpy**, respectively. That is, they copy from **p** until a **NUL** is copied or **n** bytes have been copied, whichever comes first. The forms with a **min::ptr<const char>** or **min::ptr<char>** argument are for use when the input string is part of a relocatable body. Because **min::ptr<char>** is not implicitly cast-able to **min::ptr<const char>**, both types are argument are provided for.

When a string general value is created, if the input string is short enough for the general value to hold the string itself, a direct string general value is created. Otherwise if the string has at most **min::max_id_strlen** bytes (see p71), **min::new_str_gen** searches to see if any equal string exists. If such a string exists, a pointer to the stub of the existing string is returned in the new **min::gen** value, and no new stub is created. Otherwise, if the input string is 8 or fewer **char**'s, a short string stub is created, and if the input string has more than 8 **char**'s, a long string stub and body are created.

Thus if two identifier string **min::gen** values are equal as strings, they have **== min::gen** values.

In order to do the search, a hash table of indirect identifier strings is maintained.

There are also functions to create new general string values from vectors of UNICODE characters:

```
min::gen min::new_str_gen R
                  ( const min::Uchar * p, min::unsptr n )
min::gen min::new_str_gen R
                  ( min::ptr<const min::Uchar> p, min::unsptr n )
min::gen min::new_str_gen R
                  ( min::ptr<min::Uchar> p, min::unsptr n )
```

The following functions test a **min::gen** value to see if it is a string and obtain information from a string **min::gen** value.

```
          bool min::is_str ( min::gen v )
          bool min::is_id_str ( min::gen v )
          bool min::is_non_id_str ( min::gen v )

   min::unsptr min::strlen ( min::gen v )
    min::uns32 min::strhash ( min::gen v )

        char * min::strcpy ( char * p, min::gen v )
        char * min::strncpy ( char * p, min::gen v, min::unsptr n )

           int min::strcmp ( const char * p, min::gen v )
           int min::strncmp
                        ( const char * p, min::gen v, min::unsptr n )
    min::uns64 min::strhead ( min::gen v )
```

Five of these functions correspond to the standard C/C++ **strlen**, **strcpy**, **strncpy**, **strcmp**, and **strncmp** functions, and differ from these only in that instead of taking a **char *** source string argument, these functions take a **min::gen** source argument.

The **is_str** function simply returns true if and only if its argument is a string. The **is_id_str** function returns true if and only if its argument is an identifier string. The **is_non_id_str** function returns true if and only if its argument is a string but is not an identifier string.

The **strhash** function returns the hash value of the string, as per the string hashing algorithm on p74. It returns **0** if its **min::gen** argument is not an identifier string.

The **strhead** function is an optimized function that returns the first 8 bytes of the string in a **min::uns64** value. This is intended to be used by overloading it with **min::uns8[8]**, as in the code:

```
union { min::uns64 u; min::uns8 s[8]; } v;
min::gen g;
. . . set g to a string value . . .
v.u = min::strhead ( g );
. . . first 8 bytes of string are now in v.s[0], ..., v.s[7] . . .
```

If a string has fewer than 8 bytes, it is padded with zero bytes at its end so that 8 bytes may always be returned. If the **min::gen** argument to **strhead** is not a string, 8 zero bytes are returned (as if the argument had been an empty string).

strhead is useful for testing to see what kind of lexeme the string represents; for example, if the first byte is a letter, or the first byte is **'** and the second byte is a letter, then the string may represent a word, but if the first byte is **'** and the second is zero, the string may represent a separator. Note that the bytes are UTF-8 encoded, and may represent non-ASCII UNICODE characters. Note that UTF-8 can encode a UNICODE character in as many as 7 bytes, so if the string contains non-ASCII characters, the last of the 8 bytes may belong to a truncated non-ASCII UNICODE character.

To permit hash values of arbitrary strings to be computed, without creating **min::gen** values from them first, the following functions are provided:

```
min::uns32 min::strhash ( const char * p )
min::uns32 min::strnhash ( const char * p, min::unsptr n )
```

Both **strhash** and **strnhash** accept **NUL** terminated strings, but **strnhash** stops reading the string after the first **n** bytes if none of these bytes is NUL. These functions work for strings of arbitrary length, and not just for strings of identifier string length.

A string hash value is computed according to the following machine independent algorithm:

```
hash = 0
n = length of string
for i = 1 through n:
    c = i'th char of string as unsigned 8 bit integer
    hash = ( hash * 65599 ) + c
if hash = 0, then hash = 2**32 - 1
```

where the final result is truncated to 32 bits. The constant is a prime such that multiplication by it may be turned into shifts and adds by compilers: $65599 = 2^{16} + 2^{6} - 1$. A hash value is never zero (so zero can be used to denote a missing hash value).

The low order bits of the hash value are random, so it can be truncated to provide a random hash.

The following functions can be used to convert **min::gen** string values to numbers:

```
bool min::strto
              ( min::int32 & value, min::gen g, int base = 0 )
bool min::strto
              ( min::int64 & value, min::gen g, int base = 0 )
bool min::strto
              ( min::uns32 & value, min::gen g, int base = 0 )
bool min::strto
              ( min::uns64 & value, min::gen g, int base = 0 )
bool min::strto ( min::float32 & value, min::gen g )
bool min::strto ( min::float64 & value, min::gen g )
```

If the **min::gen** argument is a string consisting of an ASCII-whitespace surrounded number legal to the UNIX **strtol**, **strtoll**, **strtoul**, or **strtoull** functions with given **base** argument, or to the UNIX **strtof**, or **strtod** functions, respectively, **true** is returned and the number is returned in the **value** argument. Otherwise **false** is returned and the **value** argument is left untouched. This last includes the case where the **min::gen** argument is not a string, the case where the **base** argument does not have a legal value, and the case where the number is 'empty' (the string is just whitespace). For unsigned integers, it includes the case where the string begins with a minus sign.

If the resulting number is outside the range of **value**, an approximation to that number is stored. For integer **value**'s this is the maximum or minimum integer that can be stored. For floating point it is plus or minus infinity for large numbers such as **1e+500**, and **0** for tiny numbers such as **1e-500**.

For floating point **value**'s, the **min::gen** argument may be a string such as **"-Inf"** or **"NaN"**, with letter case ignored.

5.10.2 Protected String Pointers

A read-only pointer to the bytes of a string **min::gen** value can be obtained using the following functions to create and use a ***string pointer***:

```
(constructor) min::str_ptr sp ( min::gen v )
(constructor) min::str_ptr sp ( const min::stub * s )
(constructor) min::str_ptr sp ( void )

min::str_ptr & operator =
                  ( min::str_ptr & sp, min::gen v )
min::str_ptr & operator =
                  ( min::str_ptr & sp1, min::str_ptr const & sp2 )
min::str_ptr & operator =
                  ( min::str_ptr & sp1, const min::stub * s )

operator bool ( min::str_ptr const & sp )

char sp[i] — for min::unsptr i

min::ptr<const char> min::begin_ptr_of ( min::str_ptr const & sp )

min::unsptr min::strlen ( min::str_ptr const & sp )
 min::uns32 min::strhash ( min::str_ptr const & sp )
     char * min::strcpy ( char * p, min::str_ptr const & sp )
     char * min::strncpy
                  ( char * p,
                    min::str_ptr const & sp, min::unsptr n )
        int min::strcmp ( const char * p, min::str_ptr const & sp )
        int min::strncmp
                  ( const char * p,
                    min::str_ptr const & sp, min::unsptr n )
```

The constructors create a string pointer pointing to the **char**'s of the string specified by the **min::gen** or '**min::stub ***' argument. A **min::gen** argument should be a direct string or a pointer to a short or long string stub, and a stub pointer should be a pointer to a short or long string stub. If the arguments are not, then the string pointer is set to the **NULL** state in which using it to access the string will almost certainly cause a memory fault. The constructor with no argument also sets the string pointer to the **NULL** state. The operator to convert a string pointer to a **bool** returns **true** if the string pointer is not in the **NULL** state, and **false** if the string pointer is in the **NULL** state.

The **=** operator can reset the string pointer to point at a different **min::gen** value, or at a stub, or at the value pointed at by another string pointer. Like the constructor, the string pointer is set to the **NULL** state if the **min::gen** value or stub pointer is not a string, or the string pointer being assigned from is itself in the **NULL** state.

The **strlen**, **strhash**, **strcpy**, **strncpy**, **strcmp**, and **strncmp** functions retrieve the same information about the string pointed at by a string pointer as they retrieve about the string value the pointer points at.

The **min::begin_ptr_of** function returns a **min::ptr<const char>** pointer to the **NUL**-terminated vector of **char**'s that is the string. Its value can be converted to a '**const char ***' by the unitary '**~**' operator, and that value may be passed to the **strcpy**, **strcmp**, ... C library functions. See 5.5.2.2[p31].

sp[i] can be used to access the **i+1**'st byte of the **NUL**-terminated string. There is no protection against the operator index being longer than the string length. Note that the return type is **char** and not '**char &**', so **sp]i]** cannot be used to write the **i+1**'st byte.

For direct and short strings the string pointer, when it is created, copies the string bytes into a buffer internal to the string pointer, in order to save the bytes in a direct string value, or to add a missing **NUL** to the end of the short string **char** vector. In long string cases no copying is done, and the string pointer is essentially just a pointer to the string stub, which in turn points at the string proper inside a relocatable string body.

Numbers can be read from a string pointed at by a string pointer by the following functions:

```
bool min::strto
               ( min::int32 & value,
                 min::str_ptr const & sp, min::unsptr & i,
                 int base = 0 )
bool min::strto
               ( min::int64 & value,
                 min::str_ptr const & sp, min::unsptr & i,
                 int base = 0 )
```

```
bool min::strto
            ( min::uns32 & value,
              min::str_ptr const & sp, min::unsptr & i,
              int base = 0 )
bool min::strto
            ( min::uns64 & value,
              min::str_ptr & const sp, min::unsptr & i,
              int base = 0 )

bool min::strto
            ( min::float32 & value,
              min::str_ptr & const sp, min::unsptr & i )
bool min::strto
            ( min::float64 & value,
              min::str_ptr & const sp, min::unsptr & i )
```

If the string pointer is not in the **NULL** state and the part of the string beginning with **sp[i]** is ASCII whitespace followed by a number legal to the UNIX **strtol**, **strtoll**, **strtoul**, or **strtoull** functions with the given **base** argument, or to the UNIX **strtof**, or **strtod** functions, respectively, **true** is returned, the number is returned in the **value** argument, and the **i** argument is updated to point just after the number. Otherwise **false** is returned and **value** and **i** are left untouched. This last includes the case where the string pointer is in the **NULL** state, the case where the **base** argument does not have a legal value, and the case where the number is 'empty' (the string is just whitespace). For unsigned integers, it includes the case where the string begins with a minus sign.

If the resulting number is outside the range of **value**, an approximation to that number is stored. For integer **value**'s this is the maximum or minimum integer that can be stored. For floating point it is plus or minus infinity for large numbers such as **1e+500**, and **0** for tiny numbers such as **1e-500**.

For floating point **value**'s, the string representing the number may be **"-Inf"**, **"NaN"**, or similar, with letter case ignored.

The following function makes a copy of a non-identifier string pointed at by a string pointer in a preallocated stub:

```
min::gen min::copy
                ( min::gen preallocated,
                  min::str_ptr & const sp )
```

Only non-identifier strings may be so copied, because they are the only strings that are not in the string hash table.

5.10.3 Unprotected String Functions

The following unprotected function is equivalent to **min::begin_ptr_of()** with its result cast to '**const char ***':

```
const char * MUP::str_of ( min::str_ptr const & sp )
```

However, note that the pointer returned is relocatable, so this function is unprotected.

The following unprotected functions may be used to access the internals of short and long strings.

A ***short string stub*** is collectible, has **min::SHORT_STR** stub type code, and has an immutable **min::uns64** stub value that holds a **NUL** padded 8 **char** vector and can be read by

```
min::uns64 MUP::short_str_of ( const min::stub * s )
```

This function does not check the type of the stub **s**. The **min::uns64** value returned by **MUP::short_str_of** should be overlaid by a union with a **char[ ]** buffer, as in

```
union { min::uns64 str; char buf[9]; } u;
min::stub * s1;
. . . set s1 to point at a short string stub . . .
u.str = MUP::short_str_of ( s1 );
u.buf[8] = 0; // Be sure result is NUL terminated.
cout << u.buf;
```

Short string values are **NUL** (zero) padded 0 to 8 **char** strings. To be sure any value read is **NUL** terminated, a **NUL** (zero) must be stored after the value read, as is done by **u.buf[8] = 0** in the example. Of course, short string values have more bytes than can be stored in a direct string (p72).

A ***long string stub*** is collectible, has **min::LONG_STR** stub type code, and has a value that is a pointer to a **MUP::long_str** type body which holds an arbitrary length **NUL** terminated **char** string.

The long string body consists of the 32-bit length and 32-bit hash value of the string, followed by a **char** vector containing the string proper with the terminating NUL. The **char** vector is padded to a multiple of 8 bytes with NUL bytes, but the terminating **NUL** and the padding are not included in the length. The length is larger than 8.

The following are unprotected functions to return a relocatable pointer to the long string body, a relocatable pointer to the string itself, and the length and hash of the string.

```
MUP::long_str * MUP::long_str_of ( const min::stub * s )
   const char * MUP::str_of ( MUP::long_str * str )
    min::unsptr MUP::length_of ( MUP::long_str * str )
     min::uns32 MUP::hash_of ( MUP::long_str * str )
```

These functions are unprotected because **long_str *** pointers are relocatable.

5.11 Labels

Object attribute labels (see Section 5.19.7 p204) are often ***atoms***, i.e., single numbers or identifier strings. But they may be sequences of atoms. Such sequences are represented by ***labels***.[5] Labels may also be elements of other labels.

A label is just a vector of name components, where a name component is an atom or a label. Labels are immutable and have the property that no two distinct label stubs can have equal vectors of name components.

Note that a label of one element is distinct from the element itself and has a different hash code. Also note that labels can be elements of labels. A programming language may wish to require that the elements of labels be numbers or strings, that numbers and strings be treated as labels of one element, and that proper labels with zero or one element not be created. But the **min.h** code does not do this.

A ***label stub*** is collectible, has **min::LABEL** stub type code, and has an immutable value. The label value may be read by using **min::lab_ptr**'s:

```
(constructor) min::lab_ptr labp ( min::gen v )
(constructor) min::lab_ptr labp ( const min::stub * s )
(constructor) min::lab_ptr labp ( void )

                operator const min::stub *
                    ( min::lab_ptr const & labp )
min::lab_ptr & operator =
                    ( min::lab_ptr & labp, min::gen v )
min::lab_ptr & operator =
                    ( min::lab_ptr & labp, const min::stub * s )

min::gen labp[i] — for min::uns32 i

min::ptr<const min::gen> min::begin_ptr_of
                              ( min::lab_ptr & labp )
min::ptr<const min::gen> min::end_ptr_of
                              ( min::lab_ptr & labp )

min::uns32 min::lablen ( min::lab_ptr & labp )
min::uns32 min::labhash ( min::lab_ptr & labp )
```

Here if **v** or **s** do not point at the stub of a label the new label pointer is set to **min::NULL_STUB**, but this is not in an of itself an error. The resulting **min::lab_ptr** can be tested to see

[5]Labels could also be represented by sublists stored inside objects (p189), but each label tends to be reused by many objects, and storing it inside each using object would be inefficient. In addition labels are useful as function arguments.

if it is **==** to **min::NULL_STUB**. Using such a pointer to access parts of a label, however, gives undefined results, but usually a memory fault. The no-argument label pointer constructor also creates a label pointer equal to **min::NULL_STUB**, and the **=** operators set a label pointer just as a constructor with the **=** right side as the constructor argument would.

Given a label pointer **labp** pointing at a real label, **labp[i]** is the **i+1**'st element of the label, for **0<i<min::lablen(labp)**, where the **min::lablen** function returns the number of elements in the label. The **min::labhash** function returns the hash of the label.

The **min::begin_ptr_of** and **min::end_ptr_of** functions return **min::ptr<const min::gen>** values that point at the first label element and the location just after the last label element, respectively. These functions return **min::ptr** pointers that remain valid even if the label body is relocated.

The length of a label is the number of elements (general values) in the label. The length of a label may be read from a label pointer, or may be read directly by the following functions:

```
min::uns32 min::lablen ( const min::stub * s )
min::uns32 min::lablen ( min::gen v )
```

The hash value of a label may be computed from the label pointer, or directly by the following functions:

```
min::uns32 min::labhash ( const min::stub * s )
min::uns32 min::labhash ( min::gen v )
min::uns32 min::labhash ( const min::gen * p, min::uns32 n )
```

The last function computes the hash value for a label that could be created from the given vector **p** of **n** general values, where each general value is a name component.

The hash of a label is computed from the hash of each of its elements using:

```
const min::uns32 min::labhash_initial = 1009
const min::uns32 min::labhash_factor = 65599**10 (mod 2**32)
      min::uns32 min::labhash ( min::uns32 hash, min::uns32 h )
```

in the following machine independent algorithm:

```
hash = min::labhash_initial;
n = length of label
for i = 1 through n:
    h = hash of i'th element of label
    hash = min::labhash ( hash, h )
```

where **min::labhash (hash, h)** is defined as:

```
min::labhash ( hash, h ):
    // All arithmetic is mod 2**32
    hash = hash * min::labhash_factor + h
    if ( hash == 0 ) hash = -1
```

```
    return hash
```

Comparing this with the hash algorithm for strings on p74, one sees that as long as label elements are numbers and strings of fewer than 10 bytes, the hash of a label is equivalent to the hash of the string made by concatenating a prefix and then the label elements, where each string of fewer than 10 bytes is padded to 10 bytes by prefacing it with **NUL** bytes. Note that numbers are treated as 8 byte strings; see p70. The prefix is any 10 byte string with hash value **min::labhash_initial = 1009**.

The two argument **labhash (hash, h)** function performs the incremental step in the computation of a label hash, and can be useful in some lookup situations.

Although label values are generally read using lab pointers, a label value may be read by the protected functions:

```
min::uns32 min::labncpy
                    ( min::gen * p,
                      const min::stub * s, min::uns32 n )
min::uns32 min::labncpy
                    ( min::gen * p,
                      min::gen v, min::uns32 n )
```

These read an initial segment of the label vector into the location addressed by **p**. If the label vector has **n** or more elements, the first **n** elements are read. Otherwise, as many elements as the label vector has are read. The number of elements read is returned in any case. The label can be denoted by either its stub address or by a general value pointing at its stub address.

A label may be created by the following protected functions:

```
min::gen min::new_lab_gen
                  ( const min::gen * p,
                    min::uns32 n )
min::gen min::new_lab_gen
                  ( min::ptr<const min::gen> p,
                    min::uns32 n )
min::gen min::new_lab_gen
                  ( min::ptr<min::gen> p,
                    min::uns32 n )
```

Here **p** must point to a vector of **n min::gen** values that becomes the value vector of the label. Each **min::gen** value must be a name component. This function returns any existing label with elements equal to those given by the function arguments, in preference to creating a new label. Thus two **min::gen** label values with **==** elements are **==**.

For convenience in generating frequently used labels, 2, 3, 4, and 5 element labels whose elements are strings may be created by

```
min::gen min::new_lab_gen
                  ( const char * s1,
                    const char * s2 )
min::gen min::new_lab_gen
                  ( const char * s1,
                    const char * s2,
                    const char * s3 )
min::gen min::new_lab_gen
                  ( const char * s1,
                    const char * s2,
                    const char * s3,
                    const char * s4 )
min::gen min::new_lab_gen
                  ( const char * s1,
                    const char * s2,
                    const char * s3,
                    const char * s4,
                    const char * s5 )
```

The following function returns true if and only if its argument is a label:

```
bool min::is_lab ( min::gen v )
```

The following unprotected constructors and functions operating on the resulting unprotected label points are just like their protected versions except that they assume any **min::gen** value **v** or **min::stub *** value **s** is a label and furthermore do not check subscript ranges:

(constructor) `MUP::lab_ptr labp ( min::gen v )`
(constructor) `MUP::lab_ptr labp ( min::stub * s )`
(constructor) `MUP::lab_ptr labp ( void )`

```
               operator const min::stub *
                   ( MUP::lab_ptr const & labp )
MUP::lab_ptr & operator =
                   ( MUP::lab_ptr & labp, min::gen v )
MUP::lab_ptr & operator =
                   ( MUP::lab_ptr & labp, const min::stub * s )
  min::gen labp[i]

min::ptr<const min::gen> min::begin_ptr_of
                                  ( MUP::lab_ptr & labp )
min::ptr<const min::gen> min::end_ptr_of
                                  ( MUP::lab_ptr & labp )

min::uns32 min::lablen ( MUP::lab_ptr & labp )
min::uns32 min::labhash ( MUP::lab_ptr & labp )
```

5.12 Names

A ***name*** is a number (5.9 [p69]), an identifier string (5.10 [p71]), or a label (5.11 [p79]). A name can also be viewed as a sequence of ***name components***, each of which is a number, identifier string, or label. Names and name components are all immutable values which have an associated hash value.

A number or identifier string is used to represent a 1-component name whose only component is the number or string. Other names are represented by labels whose elements are the components of the names. A label whose only component is a number or identifer string is not used to represent a name, in order to ensure that each name has a unique representation.

The following functions concern names in general:

```
        bool min::is_name ( min::gen v )
  min::uns32 min::hash ( min::gen v )
         int min::compare ( min::gen v1, min::gen v2 )
  min::int32 min::is_subsequence ( min::gen v1, min::gen v2 )
    min::gen min::new_name_gen ( const char * s )
    min::gen min::new_name_gen ( min::ptr<const char> s )
```

The **min::is_name** function returns true if and only if its argument is a name (number, identifier string, or label).

The **min::hash** function returns a non-zero ***hash value*** of its argument, which must be a name.

The **min::compare** function returns an integer < 0, $= 0$, or > 0 according to whether its **v1** argument is less than, equal to, or greater than its **v2** argument. In this ordering numbers are before strings, strings are before labels, and labels are before any non-name values. Numbers are ordered numerically, strings are ordered lexicographically as per the C language **strcmp** function, and labels are ordered lexicographically using the **min::compare** function recursively to compare label elements. Non-name values are ordered by converting their **min::gen** values to bit strings and comparing the bit strings. This means that **min::gen** values that point at stubs are ordered according to their stub addresses.

The **min::is_subsequence** function returns the index of the first occurrence of its first argument **v1** as a subsequence of its second argument **v2**, or returns **-1** if there is no such occurrence. For this function, a non-label must be an atom, and is equivalent to a length **1** label whose sole element is the atom.

The **min::new_name_gen** functions convert a UTF-8 character string to a name. The character string is parsed into components that contain no (horizontal or vertical) space characters and these are made into a name that is returned. If there is more than one component, a label is returned with each component being a MIN string element. If there is exactly one component, a MIN string equal to that component is returned. If there are no components, a label with zero elements is returned.

The following names are defined for general use:

```
min::gen min::TRUE            = min::new_str_gen ( "TRUE" )
min::gen min::FALSE           = min::new_str_gen ( "FALSE" )
min::gen min::empty_str       = min::new_str_gen ( "" )
min::gen min::empty_lab       = min::new_lab_gen ( NULL, 0 )
min::gen min::doublequote     = min::new_str_gen ( "\"" )
min::gen min::line_feed       = min::new_str_gen ( "\n" )
min::gen min::colon           = min::new_str_gen ( ":" )
min::gen min::semicolon       = min::new_str_gen ( ";" )
min::gen min::dot_initiator   = min::new_str_gen ( ".initiator" )
min::gen min::dot_separator   = min::new_str_gen ( ".separator" )
min::gen min::dot_terminator  = min::new_str_gen ( ".terminator" )
min::gen min::dot_type        = min::new_str_gen ( ".type" )
min::gen min::dot_position    = min::new_str_gen ( ".position" )
```

5.13 Packed Structures

A ***packed structure*** is a class datum stored in a body associated with a stub that has a `min::PACKED_STRUCT` stub type code.

The class type must be similar to a C-language `struct` in that:

1. Construction of a datum of the type by a no-argument constructor does not have to set any datum byte to a non-zero value. Note that bytes need not be set to any value, but zero must be an acceptable value for all bytes of a newly constructed datum.

   When a packed structure is created it is zeroed and then one special header member described below, the `control` member, is given a value. No constructor is called.

2. Destruction of a datum of the type does nothing.

   When a packed structure is destroyed, no destructor is called.

3. Assignment of a datum of the type from another datum of the same type is equivalent to a `memcpy` operation.

   When a packed structure is moved by the copying part of the ACC, it is moved by `memcpy`.

Numeric, `min::gen`, and `const min::stub *` values are permitted as packed structure class members, as are `min::packed_...ptr<...>` values (see below) and C++ pointers. Note that all these are initialized to `0` (equal to `NULL` or `min::NULL_STUB` for pointers) when a packed structure is created.

The type of a packed structure must have as its first member (that is, the member at displacement `0`);

```
const min::uns32 control;
```

This first member holds subtype information and flags.

The packed structure type may have a public base structure, as long as that also follows the above rules. In this case the '**control**' member must be the first member of the base structure. This rule can be applied recursively to get, for example, a '**struct**' based on a public '**struct**' based on a public '**struct**' whose first member is '**control**'.

There are two types of pointers that can be used to access members of packed structures:

```
min::packed_struct_ptr<S>
min::packed_struct_updptr<S>
```

where **S** is the class type of the structure. The first ***read-only pointer*** type permits read-only access to packed structure members, while the second ***updatable pointer*** type permits read-write access. These pointers are like the type '**const min::stub ***' but with extra clothes. Pointers of these types can also be included as members in packed structures and in packed vector headers and elements (see 5.14 [p91]).

The type of a packed structure is described at run-time by a **min::packed_struct<S>** C++ static object, where **S** is the type of the structure. The **new_gen** and **new_stub** member functions of this C++ static object can be used to create new packed structures of the described type.

More explicitly, to create a new type of ***packed structure*** named **pstype** use

```
(constructor) min::packed_struct<S> pstype
                  ( const char * name,
                    const min::uns32 * gen_disp = NULL,
                    const min::uns32 * stub_disp = NULL )
(constructor) min::packed_struct_with_base<S,B> pstype
                  ( const char * name,
                    const min::uns32 * gen_disp = NULL,
                    const min::uns32 * stub_disp = NULL )
min::uns32 min::DISP ( & S::m )
min::uns32 min::DISP_END

                min::gen pstype.new_gen ( void )
       const min::stub * pstype.new_stub ( void )
             min::uns32 pstype.subtype
     const char * const pstype.name
const min::uns32 * const pstype.gen_disp
const min::uns32 * const pstype.stub_disp
```

where

S The type of the body of packed structures of the type being declared. This type must be a class whose first member is

```
const min::uns32 control;
```

This member holds a code that effectively points at **pstype**, and also holds some flags. This member is initialized by **new_gen** or **new_stub** and must not be changed by the user. **S** must meet the requirements given on p84.

B **S** may have a single base type **B** as in

```
struct S : public struct B { ... }
```

where **B** is a packed structure type meeting the requirements given on p84. In this case **S** does <u>not</u> have a '**control**' member, as the '**control**' member of **B** serves as the '**control**' member of **S**.

packed_struct_with_base<S,B> enables conversion of **min::packed_struct_xxxptr<S>** pointers to **min::packed_struct_xxxptr<B>** pointers. In fact, given

packed_struct_with_base<S,B1> s
packed_struct_with_base<B1,B2> b1
....
packed_struct_with_base<BN,B> bn
packed_struct<B> b

then conversion of **min::packed_struct_xxxptr<S>** pointers to **min::packed_struct_xxxptr<B>** pointers is enabled.[6]

subtype A small integer automatically assigned to uniquely identify **pstype**. This is stored in the '**control**' member of all packed structures that are created by **pstype.new_gen** or **pstype.new_stub**, and serves as a pointer from any of these packed structures to **pstype**.

name A name unique to the **pstype** datum, typically the fully qualified name of this datum. This character string may be output to identify the packed structure type when a packed structure is output.

gen_disp This C vector is a list of the displacements (in bytes) of all the **min::gen** members of **S**, terminated by the value **min::DISP_END**. The displacement of member **m** in structure type **S** should be computed by **min::DISP(&S::m)**.

[6]What actually happens is the **min::packed_struct_xxxptr<S>** pointer is implicitly converted to a **const min::stub *** pointer and the latter is converted to a **min::packed_struct_xxxptr<B>** pointer. As part of this last conversion, the **control** is read and checked, and the check finds that the object subtype is **s.subtype** which is a super-subtype of **b1.subtype**, which is a super-subtype of . . . , which is a super-subtype of **bn.subtype**, which is a super-subtype of **b.subtype**, and therefore the check passes and the conversion is legal.

This displacements vector should not be given (its address should be **NULL**) if there are no **min::gen** values in **S**. But if there are **min::gen** members, this displacements vector must be given for garbage collection purposes.

const min::gen members are treated like **min::gen** members for these purposes.

stub_disp Ditto but for **const min::stub *** members of **S** instead of **min::gen** members. The following member types are treated as **const min::stub *** members for these purposes:

```
const min::stub * const
min::packed_struct_xxxptr<S>
min::packed_vec_xxxptr<E,H,L>
```

The **pstype.new_gen** and **pstype.new_stub** functions create a new packed structure datum of the type described by **pstype** and return a **min::gen** or **const min::stub *** value pointing at it. The new packed structure is set to all zeros, except for its '**control**' member.

The function call **min::DISP (& S::m)** will return the displacement in bytes of the member **m** of a structure of **struct** type **S**. This should be used instead of assuming that **struct** layouts are tightly packed, as often they are not. The type of the **S::m** member must be one of the types mentioned above: **min::gen**, **const min::stub ***, **min::packed_...ptr<...>**, etc.

pstype.subtype, **pstype.name**, **pstype.gen_disp**, and **pstype.stub_disp** can be used to retrieve the subtype, name, and displacement information associated with **pstype**.

The subtype of a packed structure (or packed vector) and the name of that subtype can be retrieved by

```
   min::uns32 min::packed_subtype_of ( min::gen v )
   min::uns32 min::packed_subtype_of ( const min::stub * s )
   min::uns32 MUP::packed_subtype_of ( const min::stub * s )
const char * min::name_of_packed_subtype ( min::uns32 subtype )
```

These functions work in the same way for both packed structures and packed vectors (5.14 ^{p91}). No packed structure has the same subtype as any packed vector, and vice versa.

The **min::packed_subtype_of** functions return **0**, which is not a legal packed structure or packed vector subtype, if the argument does not reference a packed structure or vector. This includes the case where the argument is **min::NULL_STUB**. The **MUP::packed_subtype_of** function gives undefined result unless its argument references a packed structure or packed vector. When the argument is a packed structure or packed vector, the subtype is computed by looking in the '**control**' member of the structure or vector.

The **min::name_of_packed_subtype** function returns the **pstype.name** (p86) of the **pstype** associated with the subtype, which is useful for tracing and debugging.

Note that there can be several different **`min::packed_struct<S>`** values with different subtypes but the same structure type **`S`**. Packed structure pointers such as those of type **`min::packed_struct_ptr<S>`** described below may point at a packed structure with any of these subtypes; the pointer type is not specific to the subtype, but is rather specific to **`S`**.

The two kinds of pointers that can be used to access a packed structure. The read-only **`min::packed_struct_ptr<S>`** pointers have the usage:

```
(constructor) min::packed_struct_ptr<S> psp ( min::gen v )
(constructor) min::packed_struct_ptr<S> psp ( const min::stub * s )
(constructor) min::packed_struct_ptr<S> psp ( void )

   min::packed_struct_ptr<S> & operator =
                    ( min::packed_struct_ptr<S> & psp,
                      min::gen v )
   min::packed_struct_ptr<S> & operator =
                    ( min::packed_struct_ptr<S> & psp,
                      const min::stub * s )

                    operator const min::stub *
                       (min::packed_struct_ptr<S> const & psp )
   min::ptr<S const> operator ->
                       (min::packed_struct_ptr<S> const & psp )
   min::ref<S const> operator *
                       (min::packed_struct_ptr<S> const & psp )
```

A **`min::packed_struct_ptr<S>`** value is just a '**`const min::stub *`**' value in fancy clothes, and may be converted to/from a '**`const min::stub *`**' value.

For the constructors and the **`=`** assignment operator, if the argument **`v`** or **`s`** does not point at a packed structure of type **`S`**, the **`min::packed_struct_ptr<S>`** result is set to **`min::NULL_STUB`**, which is just the standard C language **`NULL`** cast to the '**`const min::stub *`**' type. Note that **`NULL`** cannot itself be used as an argument to the constructors or **`=`** operator, as it is convertible to either a '**`min::stub *`**' or **`min::gen`** type, and is thus creates ambiguity. But **`min::NULL_STUB`** can be used. If a constructor for **`min::packed_struct_ptr<S>`** is not given an argument, the pointer is also set to **`min::NULL_STUB`**.

A pointer equal to **`min::NULL_STUB`** gives undefined results if used to access a structure, though almost always the result will be a memory fault. The **`==`** and **`!=`** operators can be used to test whether or not a **`min::packed_struct_ptr<S>`** pointer equals **`min::NULL_STUB`**.

A **`min::packed_struct_ptr<S>`** pointer is also internally a '**`S const **`**' value that is made to behave externally like a '**`min::ptr<S const>`**' value with respect to the **`->`** and unary **`*`** operators. Thus looking ahead at the example below, if **`upv`** is a **`min::packed_struct_`**

ptr<S> value and **S** has a member **m**, then **upv->m** is the **m** member of the **S** struct pointed at by **upv**.

The read-write **min::packed_struct_updptr<S>** pointer type has as its **public** base class the **min::packed_struct_ptr<S>** pointer type, so that it can be implicitly converted to a read-only pointer. The new code defined for this read-write pointer type is:

```
(constructor) min::packed_struct_updptr<S> psup ( min::gen v )
(constructor) min::packed_struct_updptr<S> psup
                     ( const min::stub * s )
(constructor) min::packed_struct_updptr<S> psup ( void )

min::packed_struct_updptr<S> & operator =
                   ( min::packed_struct_updptr<S> & psup,
                     min::gen v )
min::packed_struct_updptr<S> & operator =
                   ( min::packed_struct_updptr<S> & psup,
                     const min::stub * s )

min::ptr<S> operator ->
                 ( min::packed_struct_updptr<S> const & psup )
min::ref<S> operator *
                 ( min::packed_struct_updptr<S> const & psup )
```

The **->** and ***** operators are redefined (i.e., <u>not</u> inherited) so that they return '**min::ptr<S>**' and '**min::ref<S>**' values instead of '**min::ptr<S const>**' and '**min::ref<S const>**' values.

min::packed_struct_updptr<S> pointers can be converted implicitly to **const min::stub *** values because the **min::packed_struct_ptr<S>** pointer type is a **public** base class of the **min::packed_struct_updptr<S>** pointer type.

min::gen, **const min::stub ***, and **min::packed_...ptr<...>** elements of packed structures must be locatable. The **MIN_REF** macro described on p40 should be used with a **min::packed_struct_updptr<S>** container type for locatable elements of a packed structure.

An example use of a packed structure is:

```
struct ps;
typedef min::packed_struct_ptr<ps> psptr;
typedef min::packed_struct_updptr<ps> psupdptr;
    // Note: pointers can be defined before struct is defined.

struct ps {
    min::uns32 control;
    min::uns32 i;
    min::uns32 j;
```

```
    const min::gen g;
    const psptr pv;
    const min::stub * const s;
};

MIN_REF ( min::gen, g, psupdptr )
MIN_REF ( psptr, pv, psupdptr )
MIN_REF ( const min::stub *, s, psupdptr )

static min::uns32 ps_gen_disp[2] =
    { min::DISP ( & ps::g ), min::DISP_END };
static min::uns32 ps_stub_disp[3] =
    { min::DISP ( & ps::s ), min::DISP ( & ps::pv ), min::DISP_END };

static min::packed_struct<ps> pstype
    ( "pstype", ps_gen_disp, ps_stub_disp );

main ( ... )
{
    . . . . . . .
    min::gen v1 = pstype.new_gen();
        // min::packed_subtype_of ( v1 ) == pstype.subtype
    psupdptr upv ( v1 );
        // min::packed_subtype_of ( upv ) == pstype.subtype
    upv->i = 55;
    upv->j = 99;
    g_ref(upv) = min::new_str_gen ( "Hello" );

    psptr pv = upv;
    // Upv is converted to a `const min::stub *' value
    // that is used to set pv.

    pv_ref(upv) = pv;
    upv = pstype.new_gen();
        // new.stub() could be used here in place of new_gen()

    // Now upv->i == 0, pv->i == 55, pv->pv->i == 55,
    //     upv->pv == min::NULL_STUB

    . . . . . . .
}
```

Packed structure pointers do no caching and there is no need for a refresh function analogous to the **`min::..._refresh`** functions for list pointers (p193).

5.14 Packed Vectors

A ***packed vector*** is like a packed structure but with an added vector that follows the structure, and with a **`min::PACKED_VEC`** stub type code instead of a **`min::PACKED_STRUCT`** type code.

The structure at the beginning of a packed vector is called the packed vector ***header*** and the vector elements that follow the header are called the packed vector ***elements***. The types of both the vector header and the vector elements must follow the three rules stated at the beginning of the Packed Structures section (5.13 ^{p84}), with two exceptions.

The type of a packed vector header, like that of a packed structure, must be a class whose first member (that is, the member at displacement **`0`**) is

```
const min::uns32 control;
```

The first exception is that the header must also have the following two other members, at no particular displacement:

```
const L length;
const L max_length;
```

where **`L`** is an unsigned integer type, and defaults to **`min::uns32`**. Here **`length`** is the current number of elements in the vector and **`max_length`** is the maximum number of elements that can be in the vector before the vector needs to be resized to increase **`max_length`** (resizing is automatic but moves the vector in memory).

The second exception is that the type of the vector elements may be a class type, but without any of the members required for the header, or may be any type appropriate to an element of such a class. So, for example, **`min::gen`** may be either the vector element type or the type of a member of the vector element type.

There are also three pointer-to-packed-vector types:

```
min::packed_vec_ptr<E,H,L>
min::packed_vec_updptr<E,H,L>
min::packed_vec_insptr<E,H,L>
```

where **`E`** is the type of the packed vector element, **`H`** is the type of the packed vector header struct, and **`L`** is the unsigned integer type of the **`length`** and **`max_length`** members of the header, and also of the subscripts used to access vector elements. The first ***read-only pointer*** type permits read-only access to packed vector members and elements, the second ***updatable pointer*** type permits read-write access, while the the third ***insertable pointer*** type permits read-write access and also permits pushing and popping vector elements and resizing the vector. These pointers are like the type '**`const min::stub *`**' but with extra clothes. These pointer types can also be used as the types of packed vector elements or as members of classes that are the types of packed vector elements or headers or of packed structures.

A packed vector type is described at run-time by a **min::packed_vec<E,H,L>** C++ static object, where **E**, **H**, and **L** are as above. The **new_gen** and **new_stub** member functions of this C++ static object can be used to create new packed vectors of the described type.

More explicitly, to create a new type of ***packed vector*** named **pvtype** use

```
struct min::packed_vec_header<L>
                {
                  const min::uns32 control;
                  const L length;
                  const L max_length;
                };
```

```
(constructor) min::packed_vec
                     <E,H=min::packed_vec_header<min::uns32>,
                        L=min::uns32>
                     pvtype
                  ( const char * name,
                    const min::uns32 * element_gen_disp = NULL,
                    const min::uns32 * element_stub_disp
                                             = NULL,
                    const min::uns32 * header_gen_disp = NULL,
                    const min::uns32 * header_stub_disp = NULL )
```

```
(constructor) min::packed_vec_with_base<E,H,B,L=min::uns32> pvtype
                  ( const char * name,
                    const min::uns32 * element_gen_disp = NULL,
                    const min::uns32 * element_stub_disp
                                             = NULL,
                    const min::uns32 * header_gen_disp = NULL,
                    const min::uns32 * header_stub_disp = NULL )
```

```
         min::gen pvtype.new_gen ( void )
const min::stub * pvtype.new_stub ( void )
         min::gen pvtype.new_gen
                            ( L max_length,
                              L length = 0,
                              E const * vp = NULL )
const min::stub * pvtype.new_stub
                            ( L max_length,
                              L length = 0,
                              E const * vp = NULL )
```

```
      const char * const pvtype.subtype
      const char * const pvtype.name
const min::uns32 * const pvtype.header_gen_disp
const min::uns32 * const pvtype.header_stub_disp
const min::uns32 * const pvtype.element_gen_disp
const min::uns32 * const pvtype.element_stub_disp
            min::uns32 pvtype.initial_max_length
          min::float64 pvtype.increment_ratio
            min::uns32 pvtype.max_increment
```

where

E The type of the elements in the vector part of the bodies of packed vectors of the type being declared. This must meet the requirements given on p91.

Also, **E** must not be a **const** type.

H The type of the header at the beginning of the bodies of packed vectors of the type being declared. This must meet the requirements given on p91.

This must be a class (or **struct**) type whose first member is

```
const min::uns32 control;
```

and which must also have the two members

```
const L length;
const L max_length;
```

where **L** is an unsigned integer type. These members are initialized by **new_gen** or **new_stub** and must not be changed by the user.

H defaults to **min::packed_vec_header<min::uns32>** which has just the require header members: **control**, **length**, and **max_length** with **L=min::uns32**. Because of this default, **H** is the second template parameter, even though is precedes the vector elements in memory.

L The unsigned integer type of the vector length and vector element subscripts of the vector. Defaults to **min::uns32**. Other sensible values for **L** are **min::uns16** and **min::unsptr**.

B **H** may have a single base type **B** as in

```
struct H : public struct B { ... }
```

where **B** is a packed structure type (see 5.13 p84). In this case **H** does not have a '**control**' member, as the '**control**' member of **B** serves as the '**control**' member of **H**. The '**length**' and '**max_length**' members may be in either **H** or **B**. If **packed_vec_with_base<E,H,B,L>** is used to define the packed vector type, conversion of

min::packed_vec_xxxptr<E,H,L> pointers to **min::packed_struct_xxxptr<B>** pointers is enabled.

subtype A small integer automatically assigned to uniquely identify **pvtype**. This is stored in the '**control**' member of all packed vectors that are created by **pvtype.new_gen** or **pvtype.new_stub**, and serves as a pointer from any of these packed vectors to **pvtype**.

name A name unique to the **pvtype** datum, typically the fully qualified name of this datum. This character string may be output to identify the packed vector type when a packed vector is output.

header_gen_disp This vector is a list of the displacements (in bytes) of all the **min::gen** members of **H**, terminated by the value **min::DISP_END**. This displacements vector should not be given (its address should be **NULL**) if there are no **min::gen** values in **H**. But if there are **min::gen** members, this displacements vector must be given for garbage collection purposes.

header_stub_disp Ditto but for **min::stub *** members of **H**. See p87 for more details.

element_stub_disp Ditto but for **min::gen** members of **E**.

element_stub_disp Ditto but for **min::stub *** members of **E**. See p87 for more details.

pvtype.name, **pvtype.header_gen_disp**, **pvtype.header_stub_disp**, **pvtype.element_gen_disp**, and **pvtype.element_stub_disp** can be used to retrieve the name and displacement information associated with **pvtype**.

In addition the parameters **pvtype.initial_max_length**, **pvtype.increment_ratio**, and **pvtype.max_increment** may be set by the user:

initial_max_length The maximum length of a packed vector newly created by calling the **pvtype.new_gen** or **pvtype.new_stub** function with no arguments. Defaults to **128**.

increment_ratio The **min::reserve** function multiplies the old maximum length by this ratio to get the maximum length increment: see formula on p101. Defaults to **0.5**.

max_increment The maximum increment of the maximum length computable by the **min::reserve** function: see formula on p101. Defaults to **4096**.

The **pvtype.new_gen** and **pvtype.new_stub** functions create a new packed vector datum of the type described by **pvtype** and return a **min::gen** or **const min::stub *** value pointing at it. A packed vector has a ***length*** which is the number of elements currently in the packed vector. It also has a ***maximum length*** which is the maximum length allowed before the packed vector must be resized. The **pvtype.new_gen** or **pvtype.new_stub** function may be given the maximum length and length and a vector of length type **E** elements that is copied to the initial value of the packed vector. If the vector **vp** of initial elements is given as **NULL**, the initial elements will be zeros. If no parameters are given to **pvtype.new_gen** or **pvtype.new_stub**, the length defaults to **0** and the maximum length defaults to **pvtype.initial_max_length**.

A newly created packed vector is set to all zeros, except for its '**control**', '**length**', and '**max_length**' members.

The subtype of a packed vector (or packed structure) and the name of that subtype can be retrieved by

```
    min::uns32 min::packed_subtype_of ( min::gen v )
    min::uns32 min::packed_subtype_of ( const min::stub * s )
    min::uns32 MUP::packed_subtype_of ( const min::stub * s )
const char * min::name_of_packed_subtype ( min::uns32 subtype )
```

These functions work for packed vectors in the same was as for packed structures; see p87 for details.

The following are defined for general use:

```
        min::packed_vec<char> min::char_packed_vec_type
  min::packed_vec<min::uns32> min::uns32_packed_vec_type
min::packed_vec<const char *> min::const_char_ptr_packed_vec_type
    min::packed_vec<min::gen> min::gen_packed_vec_type
```

Above we listed three types of pointers that can be used to access a packed vector. The first is the read-only **min::packed_vec_ptr<E,H,L>** packed vector pointer type which has the usage:

```
(constructor) min::packed_vec_ptr
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvp
                        ( min::gen v )
(constructor) min::packed_vec_ptr
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvp
                        ( min::stub * s )
(constructor) min::packed_vec_ptr
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvp
                        ( void )

min::packed_vec_ptr<E,H,L> & operator =
                  ( min::packed_vec_ptr<E,H,L> & pvp,
                    min::gen v )
min::packed_vec_ptr<E,H,L> & operator =
                  ( min::packed_vec_ptr<E,H,L> & pvp,
                    const min::stub * s )

                  operator const min::stub *
                      ( min::packed_vec_ptr<E,H,L> const & pvp )
min::ptr<H const> operator ->
                      ( min::packed_vec_ptr<E,H,L> const & pvp )
min::ref<H const> operator *
                      ( min::packed_vec_ptr<E,H,L> const & pvp )

             const L pvp->length
             const L pvp->max_length
   min::ref<E const> pvp[i]
   min::ptr<E const> pvp+i
min::ptr<E const>  min::begin_ptr_of
                            ( min::packed_vec_ptr<E,H,L> pvp )
 min::ptr<E const> min::end_ptr_of
                            ( min::packed_vec_ptr<E,H,L> pvp )
```

As for packed structures, read-only packed vector pointers used as the left operand of **->** are converted to **min::ptr<H const>** pointers. This can be used to access the '**length**' and **max_length** members of the header.

Similarly, as for packed structures the ***** operator converts read only packed vector pointers to **min::ref<H const>** references.

In addition subscripting read-only packed vector pointers gives read-only access to the elements of the vector. The subscripting operator **[]** checks the index **i** against the current length of the vector to ensure the access is legal (i.e., **pvp[i]** executes a **MIN_ASSERT** check that **0<=i<pvp->length**). The vector elements in a packed vector are organized as a C language vector so that

(~ & pvp[0])[i] references the same element as **pvp[i]**

But the vector elements are stored in a body whose address can change if a relocating function is called, so C pointers to packed vector elements are relocatable. However, temporary use within a single statement is all right if the statement does not call a relocating function, so, for example, '**~ & pvp[i]**' may be used as an argument to **memcpy**.

pvp+i can be used to compute a **min::ptr<E const>** pointer that points at the **pvp[i]** vector element. The **MIN_ASSERT** check **0<=i<pvp->length** is performed when **pvp+i** is computed.

min::begin_ptr(pvp) is the same as '**pvp + 0**' except the latter would fail if **pvp->length == 0**. **min::end_ptr(pvp)** is the same as '**pvp + pvp->length**' except the latter will always fail because the index is not less than **pvp->length**.

For example, if **memcpy** is used to copy from the vector, use **min::begin_ptr(pvp)**:

```
memcpy ( ..., ~ min::begin_ptr(pvp), sizeof ( E ) * pvp->length );
```

Using **~ & pvp[0]** instead of **~ min::begin_ptr(pvp)** will fail with a **MIN_ASSERT** fault if **pvp->length == 0**.

The read-write **min::packed_vec_updptr<E,H,L>** pointer type has as its **public** base class the **min::packed_vec_ptr<E,H,L>** pointer type, so that read-write pointers can be implicitly converted to read-only pointers. The new code defined for the read-write pointer type is:

```
(constructor) min::packed_vec_updptr
                      <E,H=min::packed_vec_header<min::uns32>,
                         L=min::uns32>
                      pvup
                  ( min::gen v )
(constructor) min::packed_vec_updptr
                      <E,H=min::packed_vec_header<min::uns32>,
                         L=min::uns32>
                      pvup
                  ( const min::stub * s )
(constructor) min::packed_vec_updptr
                      <E,H=min::packed_vec_header<min::uns32>,
                         L=min::uns32>
                      pvup
                  ( void )
```

```
min::packed_vec_updptr<E,H,L> & operator =
                ( min::packed_vec_updptr<E,H,L> & pvup,
                  min::gen v )
min::packed_vec_updptr<E,H,L> & operator =
                ( min::packed_vec_updptr<E,H,L> & pvup,
                  const min::stub * s )

min::ptr<H> operator ->
              ( min::packed_vec_updptr<E,H,L> const & pvup )
min::ref<H> operator *
              ( min::packed_vec_updptr<E,H,L> const & pvup )

  min::ref<E> pvup[i]
  min::ptr<E> pvup+i
min::ptr<E>  min::begin_ptr_of
                  ( min::packed_vec_updptr<E,H,L> pvup )
 min::ptr<E> min::end_ptr_of
                  ( min::packed_vec_updptr<E,H,L> pvup )
```

The **->**, *****, **[]**, and **+** operators and **min::begin_ptr_of()** and **min::end_ptr_of()** functions are redefined (i.e., not inherited) so that they return '**min::ptr<H>**' or '**min::ref<E>**' values instead of '**min::ptr<H const>**' or '**min::ref<E const>**' values.

The **MIN_REF** macro described in Section 5.6.2 ^{p38} should be used with a **min::packed_vec_updptr<S>** container type for locatable elements of a packed vector header.

Special considerations are required for writing vector elements that are structures containing locatable members. See the example at the end of this section.

The insertable **min::packed_vec_insptr<E,H,L>** pointer type, which permits elements to be added to a packed vector, has the **min::packed_vec_updptr<E,H,L>** pointer type as its **public** base class, so that insertable pointers can be implicitly converted to read-write or read-only pointers. The new code defined for the insertable pointer type is:

(constructor) `min::`**`packed_vec_insptr`**
```
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvip
                       ( min::gen v )
```
(constructor) `min::`**`packed_vec_insptr`**
```
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvip
                       ( const min::stub * s )
```
(constructor) `min::`**`packed_vec_insptr`**
```
                              <E,H=min::packed_vec_header<min::uns32>,
                                 L=min::uns32>
                              pvip
                       ( void )
```

```
min::packed_vec_insptr<E,H,L> & operator =
                  ( min::packed_vec_insptr<E,H,L> & pvip,
                    min::gen v )
min::packed_vec_insptr<E,H,L> & operator =
                  ( min::packed_vec_insptr<E,H,L> & pvip,
                    const min::stub * s )
```

`min::ref<E> min::`**`push`**^{S} `( packed_vec_insptr<E,H,L> pvip )`

`void min::`**`push`**^{S}
```
                     ( packed_vec_insptr<E,H,L> pvip,
                       min::uns32 n, E const * vp = NULL )
```
`void min::`**`push`**^{S}
```
                     ( packed_vec_insptr<E,H,L> pvip,
                       min::uns32 n, min::ptr<const E> vp )
```
`void min::`**`push`**^{S}
```
                     ( packed_vec_insptr<E,H,L> pvip,
                       min::uns32 n, min::ptr<E> vp )
```

```
   E min::pop
              ( packed_vec_insptr<E,H,L> pvip )
void min::pop
              ( packed_vec_insptr<E,H,L> pvip,
                min::uns32 n, E * vp = NULL )
void min::pop
              ( packed_vec_insptr<E,H,L> pvip,
                min::uns32 n, min::ptr<E> vp )
```

```
void min::resize^S
            ( packed_vec_insptr<E,H,L> pvip,
              min::uns32 max_length )
void min::reserve^S
            ( packed_vec_insptr<E,H,L> pvip,
              min::uns32 reserve_length )
```

Insertable packed vector pointers permit the current length and maximum length of the vector to be changed. The current vector length is **pvip->length** and the maximum length is **pvip->max_length** (see description of **H** on p93). The current length can be changed by **min::push**^{S} and **min::pop** functions and the maximum length can be changed by **min::resize**^{S} and **min::reserve**^{S} functions. If **min::push**^{S} needs more space it automatically calls **min::reserve**^{S}.

The **min::push** function adds elements to the end of the packed vector by incrementing the current vector length, and if this would exceed the maximum length, first calls the **min::reserve** function with the number of elements to be pushed.

The single argument **min::push**^{S} adds one element to the vector, zeros that element, and returns a reference to the element. The intended use is

```
push(pvip) = v;
```

where **v** is the value of the element to be added to the vector. If **v** is a structure type containing locatable elements, the situation is tricky: see the below example.

The two or three argument **min::push**^{S} adds **n** elements to the vector, and fills the new elements from **vp** if that is not **NULL**, or with zeros otherwise. If **vp** is not **NULL**, new elements are placed within the packed vector in the same memory order as they appear in the **vp** vector. Here **vp** cannot be an unprotected body pointer, as the data it points at might be relocated during the execution of **push**. It can either be a non-body pointer, a **min:ptr<const E>** pointer, or a **min:ptr<E>** pointer. In any case **min::acc_write_update** (5.7.1^{p43}) is called by **min::push** if **E** is locatable. Again the situation is tricky if **v** is a structure type containing locatable elements: see the below example.

The **min::pop** function removes elements from the end of the packed vector, decrementing the current vector length (which is checked to be sure it is large enough). A single element may be removed and returned, or **n** elements may be removed and returned in the **vp** vector. In the latter case removed elements are placed in **vp** in the same memory order as they appeared in the packed vector. Also, if **vp** is **NULL**, the removed elements are simply discarded instead of being copied.

The **min::resize** function resets the maximum length. If the current length would be larger than the new maximum length, the current length is reset to the new maximum length.

The **min::reserve** function checks that the current length plus the **reserve_length** is at most the maximum length, and if this check fails, resets the maximum length according to the formula:

```
new_maximum_length = max ( length + reserve_length,
                           old_maximum_length
                           +
                           min ( pvtype.max_increment,
                                   pvtype.increment_ratio
                                 * old_maximum_length ) )
```

An example use of a packed vector is:

```
struct pvh;
struct pve;
typedef min::packed_vec_ptr<pve,pvh> pvptr;
typedef min::packed_vec_insptr<pve,pvh> pvinsptr;
    // Note: pointer types may be defined before
    //       header and element types are defined.

struct pvh {
    const min::uns32 control;
          min::uns32 i;
    const min::uns32 length;
    const min::uns32 max_length;
};

struct pve {
    // Note: pve is not a locatable type but contains
    // locatable type elements which means that
    // MUP::acc_write_update must be called explicitly
    // when pve elements are written into a packed vector.

    min::gen g;
    const min::stub * s;
    min::uns8 j;
};

static min::uns32 pve_gen_disp[2] =
    { min::DISP ( & pve::g ), min::DISP_END };
static min::uns32 pve_stub_disp[2] =
    { min::DISP ( & pve::s ), min::DISP_END };

static min::packed_vec<pve,pvh> pvtype
    ( "pvtype", pve_gen_disp, pve_stub_disp );

main ( ... )
```

```
{
    min::gen v = pvtype.new_gen ( 5 );
        // min::packed_subtype_of ( v ) == pvtype.subtype
    pvinsptr pvip ( v );
        // pvip->max_length == 5
        // pvip->length == 0
        // min::packed_subtype_of ( pvip ) == pvtype.subtype
    pve e1 = { min::MISSING(), NULL, 88 };
    min::push(pvip) = e1;
       // Need not call MUP::acc_write_update as neither
       // e1.g or e1.s point at a stub.
    pvptr pvp ( v );
        // pvp->length == 1
        // pvp[0].j == 88

    pve e2[3] = { { min::MISSING(), NULL, 11 },
                  { min::MISSING(), NULL, 22 },
                  { min::MISSING(), NULL, 33 } };
    min::push ( pvip, 3, e2 );
        // Again no need to call MUP::acc_write_update because
        // e2 does not point at stubs.
        //
        // pvp[1].j == 11
        // pvp[2].j == 22
        // pvp[3].j == 33

    min::locatable_gen name1, name2;
    name1 = min::new_str_gen ( "my-name-1" );
    name2 = min::new_str_gen ( "my-name-2" );

    pve e3 = { name1, min::stub_of ( name2 ), 44 };
    min::push(pvip) = e3
        // Here it is necessary that the right size of =
        // not call relocating min::new_... functions
        // because min::push returns a `pve &' value and
        // not a min::ref<pve> value.
    MUP::acc_write_update ( pvip, e3.g );
    MUP::acc_write_update ( pvip, e3.s );
        // We must call acc_write_update ourselves as
        // pve is not itself a locatable type.

        // pvp->length == 5
```

```
        min::resize ( pvip, 10 );
            // pvp->max_length == 10

        pve e4;
        locatable_gen g2;
        locatable_var<const min::stub *> s2;
        e4 = min::pop ( pvip );
            // e4.j == 44
            // pvp->length == 4
        g2 = e4.g;
        s2 = e4.s;
            // We must make e4.g and e4.s locatable before
            // we call any more relocating functions, where we
            // are assuming they might no longer be related to
            // name1 and name2 above.

        pve e5[3];
        min::pop ( pvip, 2, e5 );
            // e5[0].j == 22
            // e5[1].j == 33
        e4 = min::pop ( pvip );
            // e4.j == 11
            // pvp[0].j == 88
            // pvp->length == 1
    }
```

Packed vector pointers do no caching and there is no need for a refresh function analogous to the **`min::..._refresh`** functions for list pointers (p193).

5.15 Files

A MIN ***file*** is a sequence of UTF-8 encoded **`NUL`**-terminated lines plus descriptive information. A **`min::file`** value is a pointer to a packed structure that contains the descriptive information and also contains a pointer to a packed **`char`** vector, the ***file buffer***, that contains the lines.

The main purpose of the **`min::file`** type is to allow lines previously read to be retrieved so they can be printed in error messages. To this end, some or all read file lines are saved in the file buffer, and an index to these lines is maintained in a ***line index***. There are two main options for managing this: one is to retain all previously read lines, and the other is to maintain only the last N lines, where N is the **`spool_lines`** parameter of the file.

The data types and members of a file are:

```
typedef min::packed_struct_updptr<min::file_struct>
                    min::file

      min::packed_vec_insprt<char> file->buffer
                        min::uns32 file->buffer->length
                        min::uns32 file->end_offset
                        min::uns32 file->end_count
                        min::uns32 file->file_lines
                        min::uns32 file->next_line_number
                        min::uns32 file->next_offset
min::packed_vec_insptr<min::uns32> file->line_index

              min::uns32 file->spool_lines
const min::line_format * file->line_format

std::istream * file->istream
     min::file file->ifile
std::ostream * file->ostream
  min::printer file->printer
     min::file file->ofile
      min::gen file->file_name
```

The members of a file may be read but not written, unless noted below. They are:

buffer The vector of **char**'s that contains the lines of the file. Each line is encoded using modified UTF-8[7] and **NUL** terminated. A non-**NUL**-terminated, modified UTF-8 encoded, incomplete last line, called a ***partial line***, may appear at the end of the buffer.

Note that in general functions that load data into files do not check for illegal UTF-8 encodings, and should not be used to load **NUL** characters into files. Line terminating **NUL** characters should be loaded by the **min::end_line** function.

Initialized to an empty buffer when the file is created. The **min::init_input...** functions arrange for filling the buffer. Some of these functions fill the buffer with the complete file. Others arrange for the buffer to be filled as needed from the file **istream** or **ifile** members. The 1-argument **min::init_input** function creates an empty buffer or empties an existing buffer and expects the user to fill the buffer using **min::push**, **min::load_...**, and **min::end_line** functions (see p110).

[7] Modified UTF-8 is UTF-8 modified by using the overlong 2-byte encoding of **NUL**, so that strings may both contain the **NUL** character (in the overlong encoding) and be **NUL** terminated.

`buffer->length` The number of `char` elements in the file buffer (as per the buffer being a packed vector: see 5.14[p91]). This is `0` when the buffer is empty.

`end_offset` The offset of the `buffer` element just after the last line-terminating `NUL` character in the `buffer`. Or `0` if there are no line-terminating `NUL` characters. This is `0` when the buffer is empty. Initialized to `0` when the file is created and set appropriately by `min::init_input...` and `min::load_...` functions.

`end_count` The number of line-terminating `NUL` characters that have ever been placed into the `buffer` for the file. Initialized to `0` when the file is created or initialized for input by a `min::init_input...` function. Incremented by `min::load_...` and `min::end_line`.

When spooling is used, lines may be deleted from the beginning of the `buffer`, but `end_count` will not be changed, so `end_count` may not be the number of `NUL` characters currently in the buffer.

`file_lines` If all the `char`'s in the file have been appended to the file `buffer`, this is the number of complete lines in the file, which equals `end_count`. Otherwise this equals `min::NO_LINE`.

Initialized to `min::NO_LINE` when the file is created and by `min::init_input...` functions that do not load the entire contents of the file into the file `buffer`. Set to the number of complete lines in the file by `min::init_input...` functions that load the entire contents of the file into the file `buffer`, or by `min::next_line` when that function reads an end of file from `istream` or `ifile`. Set to `end_count` by the `complete_file` function that marks a file has being complete.

`next_line_number` The number of the next line to be returned by the `min::next_line` function that is used by programs to read lines from a `min::file`. Or the total number of lines in the file if there is no next line (there may still be a remaining portion of a partial line). The first line number is `0`. Set to `0` by `min::init_input...` functions. Reset appropriately when the file is rewound.

`next_offset` The offset in the `buffer` of the first character of the next line to be returned by the `min::next_line` function, when that function returns a line (i.e., does not return `NO_LINE`, i.e., when `next_offset < end_offset`). Specifically, `buffer[next_offset]` is the first character of the next line.

Or when `end_offset <= next_offset`, so that `NO_LINE` is returned by `min::next_line`, the offset in the `buffer` of the first character of the partial line at the end of the buffer that has not been previously skipped by the **min::skip_remaining** function (see p113). This partial line may be empty, and it is not **NUL** terminated: see **min::remaining_length** on p113.

Set to `0` by **min::init_input...** functions. Reset appropriately when the file is rewound.

line_index If **spool_lines != 0** (see below), then **line_index[m]** is the offset in the buffer of the first character of line number **n** (or of the line-terminating **NUL** if the line is empty), where

```
m = line_index->length - ( next_line_number - n )
```

provided

```
0 < ( next_line_number - n ) <= line_index->length
```

or equivalently,

```
next_line_number - line_index->length
            <= n <
      next_line_number
```

If the file has not been rewound since it was initialized by a **min::init_input...** function with **spool_lines != 0**, it is guaranteed that

```
      line_index->length
              >=
min ( spool_lines, next_line_number)
```

so at least **min(spool_lines, next_line_number)** lines can be located using the line index.

If **spool_lines == min::ALL_LINES**, which is just the largest possible **min::uns32** integer, then **spool_lines >= next_line_number**, **line_index->length == next_line_number**, **m == n**, and all the lines in the file before the next line can be located using the line index.

Otherwise, if **spool_lines > 0**, **line_index->length** is determined by the past history of calls to **min::flush_spool** and **min::rewind** (see p114).

Created when **spool_lines** is set to a non-`0` value and set to **min::NULL_STUB** when **spool_lines** is set to a `0` value. Truncated when a file is rewound and downsized when **min::flush_spool** is called.

spool_lines

If **0**, **line_index == min::NULL_STUB** and there is no spooling. Otherwise when **min::flush_spool(n)** is called with **n <= next_line_number**, then if there are more than **spool_lines** before line number **n** in the **line_index**, lines before line number **n - spool_lines** are deleted.

Set to **0** when the file is created. Set to an argument that defaults to **min::ALL_LINES**, the largest **min::uns32** number, by **min::init_input...** functions.

line_format

Information used by the **min::print_line** function (p151) to print lines of the file. Included are the flags:

```
min::DISABLE_LINE_BREAKS
min::EXPAND_HT
min::DISPLAY_EOL
min::DISPLAY_PICTURE
min::DISPLAY_NON_GRAPHIC
```

which override the corresponding flags in **print_format.op_flags** (see p129) when printing file lines using **min::print_line**.

These flags also determine the column position of each character representative in the line, and are used to this end by the by the **min::print_line_column** function (p155).

For example, the backspace return prints as follows with the given flags:

min::DISPLAY_PICTURE	$^{B}{}_{S}$	1 column
no **min::DISPLAY_PICTURE**	**<BS>**	4 columns

line_format is set to **NULL** when the file is created, and set to an argument that defaults to **NULL** by **min::init_input...** functions. If a file's **line_format** is **NULL**, the **min::print_line** function uses **printer->print_format.line_format** (which must <u>not</u> be **NULL**) in its place.

istream

If not **NULL**, the **min::next_line** function reads lines from this **std::istream** when it finds there is no line to return to its caller. Set to **NULL** when the file is created.

ifile

If not **min::NULL_STUB**, the **min::next_line** function reads lines from this **min::file** when it finds there is no line to return to its caller. Set to **min::NULL_STUB** when the file is created.

Note that it is a programming error if **istream != NULL** and **ifile != NULL_STUB**.

ostream If not **NULL**, then the **min::flush_file** function outputs buffer **char** elements to this **std::ostream** when it finds there are elements not yet output. Line-terminating **NUL** elements are translated into calls to **std::endl**. Set to **NULL** when the file is created.

printer If not **min::NULL_STUB**, then the **min::flush_file** function outputs buffer **char** elements to this **min::printer** when it finds there are elements not yet output. Non-NUL elements in a line are output as per the **min::verbatim** printer operation (p138), and the line-terminating NUL elements are translated into sending **min::eol** to the printer. Note that **line_format** is not used when buffer elements are flushed to **printer**.

Set to **min::NULL_STUB** when the file is created.

ofile If not **min::NULL_STUB**, then the **min::flush_file** function outputs buffer **char** elements to this **min::file** when it finds there are elements not yet output. Non-NUL elements are simply appended to the end of the **ofile**, and line-terminating **NUL** elements are translated into calls to **min::line_end (ofile)**. Set to **min::NULL_STUB** when the file is created.

file_name If not **min::MISSING()**, this is the name of this file used for printing error messages concerning file lines, in particular by the **min::pline_numbers** constructor (p152). Set to **min::MISSING()** when the file is created.

Creation, initialization, and parameterization of files is accomplished by the following:

```
void min::init^S ( min::ref<min::file> file )

void min::init_line_format^R
              ( min::ref<min::file> file,
                const min::line_format * line_format )
void min::init_file_name^R
              ( min::ref<min::file> file,
                min::gen file_name )
```

`void min::`**`init_ostream`**^{R}
```
            ( min::ref<min::file> file,
              std::ostream & ostream )
```
`void min::`**`init_ofile`**^{R}
```
            ( min::ref<min::file> file,
              min::file ofile )
```
`void min::`**`init_printer`**^{R}
```
            ( min::ref<min::file> file,
              min::printer printer )
```

`const min::uns32 min::`**`ALL_LINES =`** maximum **`min::uns32`** value

`void min::`**`init_input`**^{S}
```
            ( min::ref<min::file> file,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
```

`void min::`**`init_input_stream`**^{S}
```
            ( min::ref<min::file> file,
              std::istream & istream,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
```

`void min::`**`init_input_file`**^{S}
```
            ( min::ref<min::file> file,
              min::file ifile,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
```

`void min::`**`init_input_named_file`**^{S}
```
            ( min::ref<min::file> file,
              min::gen file_name,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
```

```
void min::init_input_string^S
            ( min::ref<min::file> file,
              min::ptr<const char> string,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
void min::init_input_string^S
            ( min::ref<min::file> file,
              min::ptr<char> string,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
void min::init_input_string^S
            ( min::ref<min::file> file,
              const char * string,
              const min::line_format * line_format = NULL,
              min::uns32 spool_lines = min::ALL_LINES )
```

All these **min::init...** functions create a **file** and set their first argument if that argument initially has the value **min::NULL_STUB**.

The non-**init_input...** functions just do this and set a file member. For example, **min::init_line_format** just creates the file if necessary and sets **line_format**, while **min::init_ostream** just creates the file if necessary and sets **ostream**. Used on an existing file, these functions do nothing but change a file parameter.

The **min::init_input...** functions reinitialize all members except **ostream**, **ofile**, and **printer**. Like all file **init** functions, these create the file if necessary. All these functions take arguments that initialize **line_format** and **spool_lines**.

The **min::init_input** function assumes input to the file will come from some outside source. It empties the **buffer** and sets **istream** to **NULL**, **ifile** to **min::NULL_STUB**, and **file_name** to **min::MISSING()**. For example,

```
min::locatable_ptr<min::file> file;
init_input ( file );
...
const char * my_string = ...;
int length = ::strlen ( my_string );
min::push ( file->buffer, length, my_string );
min::end_line ( file );
...
```

Here **min::push** is just the packed vector push function (p99) applied to push character representatives to the end of the file **buffer**, and the function:

```
void min::end_line^S ( min::file file )
```

pushes a line terminating **NUL** character to the end of **buffer**, updates **end_offset** to equal the new buffer length, and increments **end_count**.

When a file initialized by **min::init_input** is complete, you should call the function:

```
void min::complete_file ( min::file file )
```

to mark the file as being complete by copying **end_count** to **file_lines**. After doing this, you should not push any more characters into the file's buffer or call **min::end_line** for the file.

You can also append to a file initialized by **min::init_input** by making it the **ofile** of another file and flushing this other file (see **min::flush_file**, p115). For example,

```
min::locatable_ptr<min::file> file1, file2;
init_input ( file1 );
init_input_... ( file2, ... );
init_ofile ( file2, file1 );
. . . . .
min::flush_file ( file2 );
```

Here **min::flush_file** copies any portion of **file2** not previously flushed to the end of **file1**. Also, it calls **complete_file** for **file1** if **file2** is complete (this last behavior can be suppressed by giving an extra argument to **min::flush_file**).

You cannot push characters to a file initialized by any **min::init_input...** function other than **min::init_input**.

You can append the contents of a string to a file initialized with **min::init_input** by using one of the functions:

```
void min::load_string^S
            ( min::file file,
              min::ptr<const char> string )

void min::load_string^S
            ( min::file file,
              min::ptr<char> string )

void min::load_string^S
            ( min::file file,
              const char * string )
```

These copy the **string** to the end of the file **buffer** and replace any '**\n**' line feeds in the copy with line terminating **NUL**'s. When these functions are called, **file** must not be complete. Any **const char * string** argument must not be a pointer to a relocatable string.

Similarly the contents of an operating system named file can be copied to the end of a file **buffer** by the function:

```
bool min::load_named_file^S
              ( min::file file,
                min::gen file_name )
```

In this case, both '**\n**' line feeds and **NUL** characters in the copy of the file contents are replaced by line terminating **NUL**'s. Also, if there is any error reading the file, an error message is written into **min::error_message** (p121) and **false** is returned, whereas if there is no read error **true** is returned. The file must have a well defined size; it cannot be a named stream. When this function is called, **file** must not be complete.

The **min::init_input_string** function initializes the file for input, loads that string into the file **buffer**, and marks the file complete. Any **const char * string** argument must not be a pointer to a relocatable string.

The **min::init_input_named_file** function initializes the file for input, sets the **file_name** member, loads the contents of the operating system file this names into the file **buffer**, and marks the file complete. The file must have a well defined size; it cannot be a named stream. If no error occurs reading the file, **true** is returned. Otherwise an error message is written into into **min::error_message** (p121) and **false** is returned.

The **min::init_input_stream** function does not load data into the file **buffer**, but instead initializes the file as per **min::init_input** with an empty **buffer** and sets the file **istream** member which causes the **min::next_line** function (see below) to load lines to the end of the file **buffer** from **istream** as new lines are required. Similarly the **min::init_input_file** function sets **ifile** which causes **min::next_line** to load lines from **ifile** as new lines are required. In both cases, **file_lines** is initialized to **min::NO_LINE**, and not changed until an end of file is read from **istream** or **ifile**, at which time **file_lines** is set to the number of lines in the file thereby marking the file as complete.

Initializing a file with a **min::init_input...** function sets **next_line_number** and **next_line_offset** to **0**, thereby setting up the **min::next_line** function to sequence through file lines, beginning with the first line of the file:

```
const min::uns32 min::NO_LINE
      min::uns32 min::next_line^S ( min::file file )
```

The **min::next_line** function returns an **offset** such that the file **buffer[offset]** element is the first character of the next **NUL**-terminated line. If there is no such line, because we are at the end of the **buffer**, **min::NO_LINE** is returned instead. In this case there may still be a non-**NUL**-terminated partial line at the end of the file (see p104).

A file is complete, in the sense the no more characters will be appended to its **buffer**, if **file_lines** contains the number of complete lines in the file, and is not equal to **min::NO_LINE**. This can be tested by

```
bool min::file_is_complete ( min::file file )
```

If **istream** is not **NULL**, then when **min::next_line** encounters the end of **buffer** and the file is not complete, **min::next_line** reads from **istream** and pushes the characters read to the end of **buffer**, until a linefeed, **NUL**, or end of file is read. Both linefeeds and **NUL**'s invoke **min::end_line** to terminate the line. If instead an end of file is read, the file is completed by setting **file_lines** to **next_line_number**, and no further use of **istream** is made on this or subsequent calls to **min::next_line**. In this case a partial line may have been produced at the end of **buffer**.

If **min::next_line** returns **min::NO_LINE**, the file may or may not be complete. Certain files have the '***completeness property***', which means that **min::next_line** returns **min::NO_LINE** only if the file is complete. A file with a non-**NULL** **istream** has the completeness property.

If the **ifile** member of the file is not **min::NULL_STUB**, instead of the **istream** member being non-**NULL**, **min::next_line** gets more characters by calling **min::next_line(ifile)** instead of by using **istream**. Characters in partial lines of **ifile** are also gotten when they become available, and the file is completed when **ifile** becomes complete. The file has the completeness property only if **ifile** has this property.

The partial line (p104) that may end the file when **min::next_line** returns **min::NO_LINE** can be returned by the functions:

```
min::uns32 min::remaining_offset ( min::file file )
min::uns32 min::remaining_length ( min::file file )
```

The **min::remaining_length** function returns **buffer->length - next_offset**, the total number of partial line bytes, and the **min::remaining_offset** function returns **next_offset**, the offset in the file buffer of the first of these bytes. Note that these bytes do <u>not</u> end with **NUL**. After processing these bytes, the function:

```
min::uns32 min::skip_remaining ( min::file file )
```

can be used to set **next_offset** to be equal to **buffer->length** in order to prevent reprocessing the bytes.

The **min::skip_remaining** function can have an undesirable interaction with the use of the two argument **min::end_line** function (see below) to make an initial segment of a partial line into a complete line. It also affects the next call to **min::next_line**; specifically, if remaining bytes are skipped, then added to and made into a complete line, the next call to **min::next_line** will return that line as if it consisted of only of the non-skipped characters that were added after the call to **min::skip_remaining**.

At any time the complete partial line at the end of the file, including bytes skipped over by **min::skip_remaining**, can be accessed by the functions:

```
min::uns32 min::partial_length ( min::file file )
min::uns32 min::partial_offset ( min::file file )
```

The **min::partial_length** function returns **buffer->length - end_offset**, the total number of partial line bytes, and the **min::partial_offset** function returns **end_offset**, the offset in the file buffer of the first of these bytes. Note that these bytes do not end with **NUL**.

If the end of the **buffer** contains a partial line, the following function may be used to split this into a complete line possibly followed by a shorter partial line:

```
void min::end_line^S
            ( min::file file,
              min::uns32 offset )
```

This function sets **buffer[offset]** to NUL, overwriting any previous **char** value of this buffer element, and making this element the line-terminating **NUL** for an initial segment of the partial line at the end of the buffer. The remainder of the partial line, if there is any, becomes the new partial line at the end of the buffer.

Note that a partial line cannot be split using the last function if **min::skip_remaining** (p113) has been used to skip past the **offset** point where the **NUL** would be written.

A main reason why the **min::file** type exists is to allow lines previously returned by **min::next_line** to be retrieved so they can be printed in error messages. The following function does this:

```
min::uns32 min::line
                 ( min::file file,
                   min::uns32 line_number )
```

Here **line_number** is **0** for the first line of the file, **line_index** must exist (i.e., not be **NULL_STUB**), and

```
       file->next_line_number - file->line_index->length
    <= line_number
     < file->next_line_number
```

is required. If these conditions are not met, **min::line** returns **min::NO_LINE**.

Whether the required conditions are met depends upon the setting of **spool_lines** and any use of the **min::flush_spool** or **min::rewind** functions which are described below.

If **spool_lines** equals **0**, **line_index** does not exist, and **min::line** always returns **min::NO_LINE**. If **spool_lines** equals min::**ALL_LINES**, the default argument setting for all **min::init_input...** functions, then all lines previously returned by **min::next_line** are always available, **line_index->length** always equals **next_line_number**, and the above requirement reduces to

```
0 <= line_number < file->next_line_number
```

For other values of **spool_lines**, which are useful if a very long input file is being read and some ability to look back a short ways is desired, **line_index** exists and **line_index->

length is a function of the history of calls to the **min::rewind** function (see below) and to the function:

```
min::uns32 min::flush_spool^S
                ( min::file file,
                  min::uns32 line_number = min::NO_LINE )
```

Provided **spool_lines > 0**, this function deletes lines from the file buffer if there are more than **spool_lines** lines before the line of the given **line_number** in the buffer. In this case all lines before the one with number

```
max ( line_number - spool_lines, 0 )
```

are deleted from **file->buffer** and **file->line_index**. If the **line_number** argument is omitted or equivalently given as **min::NO_LINE**, it is replaced by **next_line_number**. It is required that

```
0 <= line_number <= file->next_line_number
```

but it is never a programming error if **line_number** is within this range, even if no lines are deleted and even if **spool_lines** is **0** (and therefore **line_index** does not exist).

It is possible to 'rewind' a file with the function:

```
min::uns32 min::rewind
                ( min::file file,
                  min::uns32 line_number = 0 )
```

This resets the next line that will be returned by **min::next_line** to some previously returned line. The file **buffer** is not changed, but the file **line_index**, if it exists, may be truncated.

min::rewind(n) is in error if **min::line(n)** would return **min::NO_LINE**, (so line number **n** has no entry in the file **line_index**), except for two special cases. The first special case is when **n == next_line_number**, in which case **min::rewind(n)** does nothing. The second special case is when **n == 0** (the default) and also **spool_lines == 0** (so that no lines have ever been deleted from the file), in which case the file is reset so the next line returned will be the first line of the file.

Normally rewind with **n > 0** is only used with files whose **spool_length == min::ALL_LINES**, which eliminates the possibility of error.

The following function is used to copy or 'flush' the lines in a file to any **ostream**, **ofile**, and/or **printer** that the file has

```
min::uns32 min::flush_file^S
                ( min::file file,
                  bool copy_completion = true )
```

This function first calls **min::next_line** repeatedly for the file, and flushes each line returned by that function. Then it flushes any partial line left over at the end of the file, and calls **min::skip_remaining** (p113) to skip over this partial line (so it will be invisible to subsequent calls to **min::next_line**). Lastly, if the **copy_completion** argument is **true**, if the file is complete, and if the file has an **ofile**, the **min::complete_file** function is called for **ofile** to mark that file as being complete.

See descriptions of the **ostream** (p108), **ofile** (p108), and **printer** (p108) members of a file for more specifics on how lines are flushed.

The flushing sub-actions of **min::flush_file** are available by using the following functions:

```
min::uns32 min::flush_line S
                    ( min::file file,
                      min::uns32 offset )
min::uns32 min::flush_remaining S ( min::file file )
```

min::flush_line copies the line beginning at the given offset to any **ostream**, **ofile**, and/or **printer**, while **min::flush_remaining** copies any remaining bytes in the partial line at the end of the file to any **ostream**, **ofile**, and/or **printer**. Note that neither of these functions calls **min::next_line**, **min::skip_remaining**, or **min::complete_file**.

None of the **min::flush_file/line/remaining** functions call **min::flush_spool**.

The following **<<** operators do the same thing as **min::flush_file** with a **false copy_complete** argument, except that, instead of copying to the file's **ostream**, **ofile**, and/or **printer**, they copy to their lefthand operand:

```
std::ostream & operator <<
                    ( std::ostream & out,
                      min::file file )
     min::file operator << S
                    ( min::file ofile,
                      min::file ifile )
  min::printer operator << S
                    ( min::printer printer,
                      min::file file )
```

Note that these **<<** operators call the **min::next_line** and **min::skip_remaining** functions in the same way as **min::flush_file** does, and therefore these **<<** operators interact with each other and with the **min::flush_file** function. Generally such interaction is avoided by using only one of the **<<** operators or only the **min::flush_file** function on a given file. Also note that the **<<** operators never call **min::complete_file**.

5.16 Identifier Maps

An ***identifier map*** is a vector whose elements are **`min::gen`** values. This vector associates vector indices with **`min::gen`** values, and allows the input/ouput token '**`@`***id*' to be used to denote the **`min::gen`** value associated with the given *id* which is an index of the map vector.

An identifier map may have an associated output hash table which associates **`min::gen`** values with indices and is the inverse of the vector map. Or alternatively it may have an associated input hash table which associates symbols (e.g., MIN strings or labels) with **`min::gen`** values.

Identifier maps can be created and read by the following:

```
typedef min::packed_vec_ptr
                     < min::gen,
                       min::id_map_header<min::uns32> >
                 min::id_map

min::id_map min::init^R ( min::ref<min::id_map> map )

const min::uns32 map->length
  const min::gen map[id]
```

The length of the identifier map vector is **`map->length`**. The value of the vector element with index **`id`** is **`map[id]`**. **`map[0] == NONE()`** is always true, but if **`map[id] == NONE()`** for **`0 < id < map->length`** or if **`id >= map->length`** then **`id`** has <u>not</u> been given a value by the map.

The **`min::init`** function may be used to create or reset a **`map`** so that **`map->length == 1`** and **`map[0] == NONE()`** and neither the output or input hash tables exist.

5.16.1 Output Using an Identifier Map

When an identifier map is used for output, map elements are read and written by the functions:

```
min::uns32 min::find^R
                    ( min::id_map map,
                      min::gen g )

min::uns32 min::find_or_add^R
                    ( min::id_map map,
                      min::gen g )
```

These functions maintain the identifier map ***output hash table*** that maps **`min::gen`** values back to vector indices:

`map[id]==g` if and only if **`find(map,g)==id`**

provided **g != NONE()**

Thus there is a 1-to-1 correspondance between identifier map vector indices and **min::gen** values stored in the identifier map vector (except for the value **min::NONE()**).

min::find(map,g) returns `0` if **g** is not in the identifier map vector or if **g == min::NONE()**.

The **min::find_or_add** function can be used to find the index id of a **min::gen** value **g** for use in output. If **g** is already in the **map**, its associated index id is returned. Otherwise **g** is pushed to the end of the **map** vector and assigned an index id which is returned. As special cases, **min::find_and_add(map,g)** always returns `0` if **g == min::NONE()** or if **map** is NULL.

When an identifier map is used for output, the following identifier map component is used:

```
min::uns32 map->next
```

When **min::find_and_add** assigns new identifiers, **map->length** is incremented but **map->next** is not. The identifiers **map->next**, . . . , **map->length-1** are for objects that have not yet been output. The **printer « min::flush_id_map** operation (see p170) outputs the objects and strings referenced by these identifiers in the format:

!@*identifier* **:=** *object-or-string*

In this format '**!**', '**@**', '**:=**', and the **min::gen_format** used to print *object-or-string* are parameters determined by:

map Member	// Default
min::gen map->**ID_prefix**	// MIN string **"!"**
min::Uchar map->**ID_character**	// **U'@'**
min::gen map->**ID_assign**	// MIN string **":="**
min::gen_format * map->**id_gen_format**	// **min::id_map_gen_format**

The **printer « min::flush_id_map** operation outputs smallest identifiers first and increments **map->next** until it equals **map->length**. Outputting an identifier with an object value may create new identifiers and increase **map->length**.

The identifier map vector is lengthed when it needs to become longer. The output hash table is created by the first call that requires it to exist. If this hash table is too small, it is recreated with a larger size. The

```
const min::uns32 map->occupied
```

member of the map, which records the number of **map** vector elements that are not **min::NONE()**, is used to determine the proper size of the hash table when it is recreated.

There is more information on Printing Using an Identifier Map in section 5.18.6 p169.

It is an error for an identifier map to have both an output hash table and an input hash table.

5.16.2 Input Using a Numeric Identifier Map

When an identifier map is used to translate '**@***id*' input tokens, map elements are read and written by the functions:

```
min::gen min::map_get
                  ( min::id_map map,
                    min::uns32 id )

void min::map_set R
              ( min::id_map map,
                min::uns32 id,
                min::gen g )

void min::map_clear
              ( min::id_map map,
                min::uns32 id )
```

These functions do not use any hash table; they just read and write the identifier map vector. It is an error to use them if the identifier map has an output hash table.

The **min::map_get** function returns **min::NONE()** if the **id** is not mapped. The **min::map_set** function adds elements (containing **min::NONE()**) to the map vector as neccessary until the vector has an element with index **id**, and then sets this element equal to **g**. Here **g** cannot equal **min::NONE()**. The **min::map_clear** function removes **id** from the map, by setting its element to **min::NONE()** if it is in the vector, and by doing nothing if it is not.

When a '**@***id*' token is encountered in the input and *id* is not in the map, normal procedure is to set the **map[***id***]** element using **min::map_set** to a new preallocated stub (5.6.4 *p41*). Then when a definition for '**@***id*' is encountered later in the input, the preallocated stub is filled in.

This works for objects (see **min::copy** on p179) and non-identifier strings (see **min::copy** on p77).

However, a preallocated stub cannot be filled in with a name component: number, identifier string, or label. So a '**@***id*' token that is to be defined as referencing a name component must be defined before it is used in the input.

If it is not, the references to the token before it is defined will be left dangling when the token is defined. References before the definition will refer to the token's preallocated stub which, because it cannot be filled in, is not the final value of the token. The preallocated stub contains the index *id* of the token that created it, so an error message can be output identifying precisely the input tokens that were left dangling.

MIN does not provide input scanners to convert text to tokens, but the **ID_character** identifier map parameter (p118) can be used by such a scanner to determine whether a token has the form '**@***id*' where '**@**' is the value of the **ID_character** parameter and *id* is a string of digits.

5.16.3 Input Using a Symbolic Identifier Map

Optionally, on input '`@`*symbol*`@`' tokens may be used like variables with **`min:gen`** values. These ***symbolic identifiers*** are useful when input is generated by people, and not automatically. The '`@`*id*' tokens, on the other hand, behave like unchangeable constants and not like variables.

'`@`*symbol*`@`' tokens are not input or output by **`min`** functions.

An identifier map used for input may have an associated ***input hash table*** that maps symbols to **`min::gen`** values. Elements in this table are gotten and set by:

```
min::gen min::map_get
                    ( min::id_map map,
                      min::gen symbol )
```

```
void min::map_set R
                ( min::id_map map,
                  min::gen symbol,
                  min::gen g )
```

```
void min::map_clear
                ( min::id_map map,
                  min::gen symbol )
```

```
void min::map_clear_input
                ( min::id_map map )
```

If symbolic identifiers are being input, **`min::map_get`**, **`min::map_set`**, and **`min::map_clear`** can be used as for numeric identifiers, except that in place of an **`min::uns32`** numeric identifier one uses a **`min::gen`** MIN string. The string is the *symbol*, with or without the bracketing **`@`**'s. The **`min::map_set`** function creates the input hash table if it does not previously exist. The **`min::map_clear_input`** function deletes the input hash table.

It is an error for an identifier map to have both an input hash table and an output hash table.

5.17 Printers

A ***printer*** may be used to print general values, print graphics for control characters, and break long lines to enforce line length. Internally a **`min::printer`** is a packed structure that points at a **`min::file`** and contains additional information used to format lines put into the file. Values are written to printers via the **`<<`** operator, much like the way values are written to **std::ostream**'s. A special printer line end operation writes a line terminating **NUL** at the end of the printer file buffer, and optionally flushes the file.

The MIN system has a per-thread printer to hold error messages:

```
min::locatable_var<min::printer> min::error_message
```

This functions like **errno** in UNIX, but holds multi-line error messages instead of an integer error code. The protocol for writing an error message into this printer is to first apply the 1-argument **min::init** function to the printer to create the printer if necessary, remove any previous message from the printer, and put the printer into a default state, and then write an error message consisting of one or more complete lines to the printer. The following code is often used to do this:

```
# define ERR min::init ( min::error_message )
...
if ( error-occurred )
{
    ERR << ... << min::eol;
    return value-indicating-error-occurred;
}
...
```

Here the **min::init** function erases any previous error message.

Creation and initialization of printers is accomplished by the following:

```
     typedef min::packed_struct_updptr<min::printer_struct>
                          min::printer
min::printer min::init R
                    ( min::ref<min::printer> printer,
                      min::file file = min::NULL_STUB )
min::printer min::init_ostream S
                    ( min::ref<min::printer> printer,
                      std::ostream & ostream )

     min::file printer->file
std::ostream * printer->ostream
```

The **min::init** function both re-initializes existing printers and creates new ones. It takes a variable as its first argument, and if the variable value is **min::NULL_STUB**, then a new printer is created and the variable is set to point at the new printer. The second argument to **min::init** supplies the **min::file** to which the printer is attached. If this argument is **min::NULL_STUB**, then **min::init_input** is applied to **printer->file** to create that file if necessary and empty it if it already exists; but if the argument is not **min::NULL_STUB** then **printer->file** is set to the argument and the file is <u>not</u> emptied or changed.

The **min::init_ostream** function just executes

```
min::init ( printer );
min::init_ostream ( printer->file, ostream );
printer << min::flush_on_eol;
```

which is a very common way of setting up a printer.

Both initialization functions return the printer, which is sometimes convenient, as in the code often used with **min::error_message**: see p121.

Parameterization of printing is accomplished using the following printer parameters:

```
struct    min::line_break
{
    min::uns32 offset          // Default 0
    min::uns32 column          // Default 0
    min::uns32 line_length     // Default 72
    min::uns32 indent          // Default 4
}

                        const min::uns32 printer->column
                   const min::line_break printer->line_break
                       const min::line_break min::default_line_break
const min::packed_vec_insptr<min::line_break>
                                      printer->line_break_stack
```

```
struct              min:: print_format
                          // Defaults:
{
            min::uns32    op_flags
                          // min::EXPAND_HT
            min::uns32 *  char_flags
                          // min::standard_char_flags
    min::support_control  support_control
                          // min::latin1_support_control
    min::display_control  display_control
                          // min::graphic_and_hspace_
                          //      display_control
    min::display_control  quoted_display_control
                          // min::graphic_and_sp_
                          //      display_control
      min::break_control  break_control
                          // min::break_after_space_
                          //      break_control
 min::char_name_format *  char_name_format
                          // min::standard_char_name_format
       min::gen_format *  gen_format
                          // min::compact_gen_format
const min::line_format *  line_format
                          // min::standard_line_format
             min::uns32   max_depth
                          // 10
}
```

```
            const min::print_format printer->print_format
const min::packed_vec_insptr<min::print_format>
                                    printer->print_format_stack
```

```
const min::print_format min::default_print_format
```

```
min::id_map printer->id_map
 min::uns32 printer->id_map->next
```

```
min::uns32 printer->depth
```

print_format.op_flags:

Flags shared with **line_format.op_flags** (see p125):

```
const min::uns32 min::DISABLE_LINE_BREAKS
const min::uns32 min::EXPAND_HT
const min::uns32 min::DISPLAY_EOL
const min::uns32 min::DISPLAY_PICTURE
const min::uns32 min::DISPLAY_NON_GRAPHIC
```

Other flags:

```
const min::uns32 min::OUTPUT_HTML

const min::uns32 min::FLUSH_ON_EOL
const min::uns32 min::FLUSH_ID_MAP_ON_EOM

const min::uns32 min::FORCE_SPACE
const min::uns32 min::DISABLE_STR_BREAKS
const min::uns32 min::FORCE_PGEN
```

print_format.support_control: See p64.

print_format.display_control:

```
struct     min::  display_control
{
     min::uns32   display_char
     min::uns32   display_suppress
}

const min::display_control min::graphic_and_sp_display_control =
              { min::IS_SP + min::IS_GRAPHIC, 0 };
const min::display_control min::graphic_and_hspace_display_control =
              { min::IS_HSPACE + min::IS_GRAPHIC, 0 };
const min::display_control min::graphic_only_display_control =
              { min::IS_GRAPHIC, 0 };
const min::display_control min::graphic_and_vhspace_display_control =
              { min::IS_VHSPACE + min::IS_GRAPHIC, 0 };
const min::display_control min::display_all_display_control =
              { min::ALL_CHARS, 0 };
```

print_format.break_control:

```
struct     min::  break_control
{
     min::uns32   break_before
     min::uns32   break_after
     min::uns32   conditional_break
     min::uns32   conditional_columns
}
```

```
const min::break_control min::no_auto_break_break_control =
              { 0, 0, 0, 0 };
const min::break_control min::break_after_space_break_control =
              { min::IS_BHSPACE, 0, 0, 0 };
const min::break_control min::break_before_all_break_control =
              { 0, min::ALL_CHARS, 0, 0 };
const min::break_control min::break_after_hyphens_break_control =
              { min::IS_BHSPACE, 0, min::CONDITIONAL_BREAK, 4 };
```

print_format.char_name_format:

```
struct              min::  char_name_format
{
     // Members must encode non-zero numbers of columns.
             min::ustring   char_name_prefix
             min::ustring   char_name_postfix
}

const min::char_name_format min::standard_char_name_format =
              { (min::ustring) "\x01\x01" "<",
                (min::ustring) "\x01\x01" ">" };
```

The **ustring** members of **min::char_name_format** must not contain non-graphic characters (e.g., no spaces) and must have non-zero numbers of columns encoded in their second bytes.

print_format.line_format (see also **print_format.op_flags**, p123):

```
struct              min::  line_format
{
             min::uns32     op_flags
             char           mark
             const char *   blank_line
             const char *   end_of_file
             const char *   unavailable_line
             const char *   line_class
             const char *   line_number_class
             const char *   line_mark_class
}
```

```
const min::line_format * min::standard_line_format:
      min::DISABLE_LINE_BREAKS   // op_flags
          + min::EXPAND_HT,
      0   ,                      // mark
      NULL,                      // blank_line
      "<END-OF-FILE>",           // end_of_file
      "<UNAVAILABLE-LINE>",      // unavailable_line
      "MIN-LINE",                // line_class
      "MIN-LINE-NUMBER"          // line_number_class
      NULL                       // line_mark_class

const min::line_format * min::marked_line_format:
      // Same as min::standard_line_format except for:
      '^' ,                      // mark
      "MIN-LINE-MARK"            // line_mark_class

const min::line_format * min::picture_line_format:
      // Same as min::standard_line_format except for:
      min::DISABLE_LINE_BREAKS   // op_flags
          + min::EXPAND_HT
          + min::DISPLAY_PICTURE
      "<BLANK-LINE>"             // blank_line

const min::line_format * min::non_graphic_line_format:
      // Same as min::standard_line_format except for:
      min::DISABLE_LINE_BREAKS   // op_flags
          + min::EXPAND_HT
          + min::DISPLAY_PICTURE
          + min::DISPLAY_NON_GRAPHIC
      "<BLANK-LINE>"             // blank_line

const min::line_format * min::eol_line_format:
      // Same as min::standard_line_format except for:
      min::DISABLE_LINE_BREAKS   // op_flags
          + min::EXPAND_HT
          + min::DISPLAY_PICTURE
          + min::DISPLAY_NON_GRAPHIC
          + min::DISPLAY_EOL
      "<BLANK-LINE>"             // blank_line
```

The basic algorithm for printing a UNICODE character **c** is:

1. Compute character flags of **c** (see 5.8.1 p53):

```
min::uns16 cindex = min::Uindex ( c );
min::uns32 cflags = printer->print_format.char_flags[cindex];
```

2. Replace character flags **cflags** if **c** is unsupported:

```
min::support_control sc = printer->print_format.support_control;
if ( ( cflags & sc.support_mask ) == 0 )
    cflags = sc.unsupported_char_flags;
```

3. Compute printed representation of **c**:

```
min::display_control dc =
    [are we printing quoted character] ?
        printer->print_format.quoted_display_control :
        printer->print_format.display_control;
if ( printer->print_format.op_flags & min::DISPLAY_NON_GRAPHIC )
{
    dc.display_char &= min::IS_GRAPHIC;
    dc.display_suppress &= min::IS_GRAPHIC;
}
if ( cflags & dc.display_char )
    // Character represents itself, but if c == '\t', c is
    // represented by a sequence of 8 - printer->column % 8
    // ' ' spaces if EXPAND_HT op_flag is on.
else if ( cflags & dc.display_suppress )
    // Ignore character and do not print it
else
    // Character is represented by min::unicode::picture[cindex]
    // if that is not NULL and DISPLAY_PICTURE op_flag is on;
    // else character is represented by <X> where < and > are
    // the printer->print_format.char_name_format prefix and
    // postfix, and X is min::unicode::name[cindex] if that is
    // not NULL, or is the character code in hexa-decimal with
    // a leading decimal digit (may be 0) otherwise.
```

Note that **<FF>** represents the form feed character with (hex) code **0C**, while **<0FF>** represents the latin small letter y with diaeresis, ÿ, with code **FF**.

Note that only the low order 16 bits of **cflags** are used by a **display_control**.

4. Compute automatic break points for **c**:

```
min::break_control bc = printer->print_format.break_control;
if ( cflags & bc.break_after )
    set break after representative of c
if ( cflags & bc.break_before )
    set break before representative of c
if ( cflags & bc.conditional_break
     &&
```

```
    printer->column - printer->line_break.column
 >= bc.conditional_columns )
set break after representative of c
```

When we say that a break is set after the representative of **c**, what we mean precisely is that a break is set before the representative of the next character after **c**, unless that character also sets a break after its representative. Thus given a sequence of single spaces with the **IS_SP** flag in **bc.break_after**, followed by a letter, only one break, after the last single space and before the letter, will be set.

Note that only the low order 16 bits of **cflags** are used by a **break_control**.

The members of a printer may be read but not written. A complete list of these members follows:

file The **min::file** which contains the lines produced by the printer. Initialized by the second argument of **min::init**, if that is not **min::NULL_STUB**, or otherwise created when the printer is created. Initialized by **min::init(printer->file)** (which resets the **file** to empty) except when the 2-argument **min::init** is called (see above).

column The number of columns currently in the line being assembled, which equals the column number of the next character to be input to the printer. The first column number is **0**. Initialized to **0**.

line_break.column
The column of the first byte in the current line after the last break point in the line. Initialized to **0**.

line_break.offset
The **file->buffer** index of the first byte in the current line after the last break point in the line. Initialized to **0**.

line_break.line_length
Nominal maximum line length in columns. Initialized to **72**.

More specifically, when a character representative other than a single space, a horizontal tab, or a sequence of single spaces representing a horizontal tab is to be inserted into the line, and this would cause the number of columns in the line to exceed **line_break.line_length**, then if

```
line_break.column > line_break.indent
```

a break, consisting of a line end followed by **line_break.indent** spaces, is inserted into the printer file buffer just after the break point (i.e., is inserted at **line_break.offset**).

Single space and horizontal tab characters immediately before the inserted line end are deleted. After the break is inserted, **line_break.column** is reset to the number of indentation spaces (**= line_break.indent**) and **line_break.offset** to the first position after the inserted indentation spaces.

Break points may be inserted automatically by using **print_format.break_control**; see below.

If a non-empty **line_break_stack** exists because a **min::save_line_break** or **min::save_indent** was executed, then an element of the line break stack may provide overrides for the **column**, **offset**, and **indent** members, but not the **line_length** member, of **printer->line_break**. See p141 for details.

line_break.indent

Indention of a new line created by automatic break insertion. Also used by the **min::indent** operation (p141). Initialized to **4**.

Specifically, when a line break is automatically pushed into a printer file buffer, the break consists of a line end (**NUL** character) followed by **line_break.indent** single space characters.

line_break_stack

Used with **min::save_line_break** and **min::save_indent** to set multiple break points when printing nested lists. See p141 for details. Initialized to an empty stack.

print_format.op_flags

These are printer ***operation flags*** that control printing. When passed as function arguments they are referred to as the **print_op_flags**. The flags are listed immediately below.

DISABLE_LINE_BREAKS

If this flag is on, **printer_format.break_control** is ignored and automatic break points are not inserted (that is, step 4 on p127 is skipped).

EXPAND_HT

EXPAND_HT causes the character **'\t'** to be represented by a sequence of single spaces when it is to be displayed as 'itself' (and not using its picture or name). As tabs are set every 8 spaces, the number of single spaces is:

```
8 - printer->column % 8
```

If **EXPAND_HT** is <u>off</u> and the character **'\t'** is to be displayed as 'itself', the character represents itself, but its width in columns is

```
8 - printer->column % 8
```

DISPLAY_EOL
: If this flag is on, the end of line prints as the ␤ picture character if **DISPLAY_PICTURE** is on, and as the name **NL** (e.g., in **<NL>**) otherwise.

DISPLAY_PICTURE
: If this flag is on, characters not displayed as themselves are displayed as picture characters using **min::unicode::picture[cindex]** if this is not **NULL**, as per p127. Otherwise they are displayed using their name if that exists, or using their hex code if no name exists.

DISPLAY_NON_GRAPHIC
: If this flag is on, non-graphic characters (including single spaces and horizontal tabs) are not displayed as themselves or suppressed, regardless of the setting of the display control.

OUTPUT_HTML
: If this flag is on, HTML special characters that are displayed as per **print_format.display_control** (see step 3 on p127) are replaced in the printer output buffer as follows:

HTML Special Character	Output	Column Action
<	<	+= 1
>	>	+= 1
&	&	+= 1
<LF>	<CR>	= 0
<CR>	none	none
<FF>	<CR><CR>	= 0
<VT>	<CR><CR>	= 0
<HT>	— see EXPAND_HT —	

FLUSH_ON_EOL
: If this flag is on, a **min::flush()** printer operation (p134) is executed at the very end of each **min::eol** operation.

FLUSH_ID_MAP_ON_EOM
: If this flag is on, a **min::eom()** printer operation performs a **min::flush_id_map** operation (p170) right after its **min::bol** operation.

FORCE_SPACE
: This flag forces the **min::leading** and **min::trailing** printer operations to output a single space character. See p147.

DISABLE_STR_BREAKS
: This flag disables breaking quoted strings and breakable strings. See p165 and p165.

FORCE_PGEN

This flag causes the **<<** operator to treat numbers and characters strings that are to be printed directly by the **<<** operator as **min::gen** numbers or strings, which just means that **print_format.gen_format** is used to format them. Printer operations such as **min::space**, **min::pint**, and **min::punicode** are not affected. Non-operator printing functions with names of the form **min::print_...** are also not affected. This flag is used mostly for debugging and demonstrations of **min::leading/trailing**.

print_format.char_flags

This is the vector of character flags indexed by UNICODE character indices. See 'Compute character flags' above (p126) and Section 1 ^{p126}.

print_format.support_control

This **min::support_control** controls which characters are supported and which are not. The **support_mask** (e.g., **min::IS_LATIN1**) is bitwise AND'ed with the character flags and if the result is zero, the character is unsupported and its flags are replaced by the **unsupported_char_flags** (e.g., **min::IS_UNSUPPORTED**). See 'Replace character flags ... if ... unsupported' above (p127).

print_format.display_control

print_format.quoted_display_control

These control which characters display as themselves and which are suppressed (ignored and not printed). For characters in quoted strings, **quoted_display_control** is used, while for all other characters **display_control** is used.

The **display_char** mask (e.g., **min::IS_SP + min::IS_GRAPHIC**) is AND'ed bitwise with the character flags and if the result is non-zero, the character is displayed as itself. Note however there may be special processing in this case for the horizontal tab character.

If the character is not to be displayed as itself, then the **display_suppress** mask (e.g., **min::IS_CONTROL**) is bitwise AND'ed with the character flags and if the result is non-zero, the character is suppressed (ignored).

If a character is not displayed as itself or suppressed, its picture or name are used to represent it.

See 'Compute character representation' above (p127) for details.

print_format.break_control

This **min::break_control** controls when breaks are automatically inserted into the output (see **min::set_break**, p139).

The **break_after** mask (e.g., **min::IS_SP + min::IS_OTHER_HSPACE**) is bitwise AND'ed with the character flags and if the result is non-zero, a break is inserted 'after the character representative', or more precisely, a break is inserted before the representative of the next character if that character is not also having a break automatically inserted after it..

The **break_before** mask (e.g., **min::ALL_CHARS**) is bitwise AND'ed with the character flags and if the result is non-zero, a break is inserted before the character representative.

The **conditional_break** mask (e.g., **min::CONDITIONAL_BREAK**, set for the hyphen **'-'**) is bitwise AND'ed with the character flags and if the result is non-zero, and if

```
printer->column - printer->line_break.column >=
                conditional_columns
```

a break is inserted 'after the character representative' in the same manner as for **break_after**.

See 'Compute automatic break points' above (p127) for more details.

print_format.char_name_format
: This defines the prefix and suffix on character names and codes: e.g., provides the prefix **<** and suffix **>** for the representative **<FF>** of a form feed character.

print_format.gen_format

print_format.id_map_gen_format
: These control the printing of **min::gen** values as described in Section 5.18 *p157*. The **gen_format** is used normally and the **id_map_gen_format** member is used by id map mapping and flushing functions (5.18.6 *p169*).

print_format.max_depth
: If **printer->depth** exceeds this value when **min::standard_pgen** is called, '**...**' will be printed instead of the **min::gen** argument to **min::standard_pgen**.

print_format_stack
: Used with **min::save_print_format** and **min::restore_print_format** to save and restore the **print_format** parameters. See p136 for details. Initialized to an empty stack.

id_map
: Used to find or create identifiers when printing values with stubs and **min::gen** values that contain stub pointers. See 5.16 *p117* and **flush_id_map** (p170).

depth
: The current nesting depth. Incremented when **min::print_gen** is called to print a value with a **false disable_mapping** argument, and decremented when this function finishes. See **print_format.max_depth**, p132.

Copying characters into a printer is done by the **<<** operator:

```
min::printer operator <<^S
                 ( min::printer printer, const char * s )
min::printer operator <<^S
                 ( min::printer printer,
                   min::ptr<const char> s )
min::printer operator <<^S
                 ( min::printer printer,
                   min::ptr<char> s )
min::printer operator <<^S
                 ( min::printer printer, min::str_ptr const & s )

min::printer operator <<^S
                 ( min::printer printer, char c )
min::printer operator <<^S
                 ( min::printer printer, min::int32 i )
min::printer operator <<^S
                 ( min::printer printer, min::int64 i )
min::printer operator <<^S
                 ( min::printer printer, min::uns32 u )
min::printer operator <<^S
                 ( min::printer printer, min::uns64 u )
min::printer operator <<^S
                 ( min::printer printer, min::float64 f )
```

A '**const char ***' is interpreted as a UTF-8 encoded string of UNICODE characters as per **min::utf8_to_unicode**, p52. However, a '**const char ***' cannot be an unprotected body pointer, as the body it points into may be relocated during the execution of the **<<** ^{S} operator on a printer. A '**min::ptr<char>**' value can be used instead as a pointer to a '**const char ***' string inside a body. And a '**min::str_ptr**' can be used instead to point at the contents of a string **min::gen** value.

Horizontal tabs are set every 8 columns. When a horizontal tab is to be copied to the printer buffer, then if the **min::EXPAND_HT** printer operation flag is on, the horizontal tab is converted to between 1 and 8 single space characters. But if the flag is off, the horizontal character itself is copied. In either case the printer column is adjusted to the next multiple of 8.

When a **NUL** is to be copied into a printer file buffer, it is encoded as the overlong UTF-8 encoding **"\xC0\x80"**.

Values of types other than '**const char ***' which are presented to **<<** are converted to '**const char ***' strings as follows: '**char**' converts to a 1-**char** string whose only element is the **char** value. Numbers are converted by **printf** formats as follows:

`min::uns32`	`"%u"`
`min::uns64`	`"%llu"`
`min::int32`	`"%d"`
`min::int64`	`"%lld"`
`min::float64`	`"%.15g"`

Operations can be performed on printers by applying the **`<<`** operator to a **`min::op`**:

```
min::printer operator <<^S
                ( min::printer printer,
                  min::op const & op )
```

Operations behave for printers as **`iomanip`** C++ objects behave for **`iostreams`**. For example, the printer analog of **`std::endl`** is the ***end of line*** operation:

```
const min::op min::eol
```

which inserts a line end into the output and then executes a flush operation if the **`min::FLUSH_ON_EOL`** printer operation flag is set.

The flush operation:

```
const min::op min::flush
```

just executes

```
min::flush_file ( printer->file );
```

The ***beginning of line*** operation:

```
const min::op min::bol
```

executes **`min::eol`** if and only if the printer is <u>not</u> already at the beginning of a line.

A ... `<< printer` operation can be used to flush the contents of **`printer->file`** into an **`std::ostream`**, a **`min::file`**, or into another **`min::printer`**:

```
std::ostream & operator <<
                ( std::ostream & out,
                  min::printer printer )
     min::file operator <<^S
                ( min::file file,
                  min::printer printer )
  min::printer operator <<^S
                ( min::printer oprinter,
                  min::printer iprinter )
```

In general ... `<< printer` is equivalent to ... `<< printer->file` (see p116).

A single UNICODE character **`c`**, or a string of UNICODE characters **`str`** of a given **`length`**, can be printed using the following operations:

```
min::op min::punicode ( min::Uchar c )
min::op min::punicode
                 ( min::unsptr length,
                   const min::Uchar * str )
min::op min::punicode
                 ( min::unsptr length,
                   min::ptr<const min::Uchar> str )
min::op min::punicode
                 ( min::unsptr length,
                   min::ptr<min::Uchar> str )
```

Here it must be remembered that a '**const min::uns32 ***' pointer **str** must not be relocatable.

Numbers may be formatted by specified **printf** formats and the results printed by using the following operations:

```
min::op min::pint
                 ( min::int32 i,
                   const char * printf_format )
min::op min::pint
                 ( min::int64 i,
                   const char * printf_format )

min::op min::puns
                 ( min::uns32 u,
                   const char * printf_format )
min::op min::puns
                 ( min::uns64 u,
                   const char * printf_format )

min::op min::pfloat
                 ( min::float64 f,
                   const char * printf_format )
```

min::gen values may be printed under the control of a **min::gen_format**. As printing objects can be complex, printing **min::gen** values is described in the separate Section 5.18 p157.

Sometimes it is desirable to find out how many columns a Modified UTF-8 string would take if printed. For the horizontal tab character, this depends upon the column the character is printed in. So the following function updates a **column** variable whose initial value is that of **printer->column** before the string is printed and whose final value is **printer->column** after the string is printed.

```
min::uns32 min::pwidth
                      ( min::uns32 & column,
                        const char * s, min::unsptr n,
                        const min::print_format & print_format,
                        const min::line_format * line_format = NULL )
```

This function updates '**column**' to what it would be if the **n** byte string **s** were printed with the given **print_format**. Here **s** may be relocatable, as the **min::pwidth** function does not allocate or resize objects. This function uses **line_format.op_flags** if **line_format** is <u>not</u> **NULL**, and **print_format.op_flags** otherwise. The number of columns added to '**column**' is returned.

A function and a constant that can be useful as arguments to message printing functions are:

```
const min::op min::printf_op<unsigned length>
                        ( const char * format, ... )
const min::op min::pnop
```

Here **min::printf_op<length>** is a constructor that constructs a **min::op** containing a '**length**' character buffer which in turn contains the results of executing **sprintf (buffer, format, ...)**. This buffer is then output as a '**const char ***' string by the printer operation. Overrunning the end of the buffer is an undetected error.

The **min::pnop** constant is a no-operation that can be used as the default value of a message printing function argument.

5.17.1 Adjusting Printer Parameters

The **print_format** parameters of a printer may be saved in the **print_format_stack** of the printer, and restored from that stack. This may be done by the following operations:

```
const min::op min::save_print_format
const min::op min::restore_print_format
```

min::save_print_format pushes into the format stack a new element containing the current **print_format**, while **min::restore_print_format** resets **print_format** from the top format stack element and then pops this element from the stack.

Some printer parameters may be set by the following printer operations:

```
min::op min::set_line_length ( min::uns32 line_length )
min::op min::set_indent ( min::uns32 indent )
min::op min::set_print_op_flags ( min::uns32 print_op_flags )
min::op min::clear_print_op_flags ( min::uns32 print_op_flags )
```

min::set_print_op_flags logically OR's its argument into **print_format.op_flags** and **min::clear_print_op_flags** logically AND's the bitwise complement of its argument into

print_format.op_flags (recall that **print_format.op_flags** are referred to as **print_op_flags** when used as arguments to functions). For example,

```
printer<<min::set_print_op_flags(min::DISPLAY_PICTURE)
```

sets the **min::DISPLAY_PICTURE** flag of **printer->print_format.op_flags**.

Particular **print_format.op_flags** may be set and cleared by the following printer operations:

```
const min::op min::expand_ht
const min::op min::noexpand_ht

const min::op min::display_eol
const min::op min::nodisplay_eol
const min::op min::display_picture
const min::op min::nodisplay_picture
const min::op min::display_non_graphic
const min::op min::nodisplay_non_graphic

const min::op min::flush_on_eol
const min::op min::noflush_on_eol
const min::op min::flush_id_map_on_eom
const min::op min::noflush_id_map_on_eom

const min::op min::force_space
const min::op min::noforce_space
const min::op min::disable_str_breaks
const min::op min::nodisable_str_breaks
const min::op min::force_pgen
const min::op min::noforce_pgen
```

Here **printer << min::display_picture** sets the **min::DISPLAY_PICTURE** printer operation flag in **printer->print_format.op_flags**, while the **printer << min::nodisplay_picture** clears this flag. The other operations similarly set or clear their associated flags.

The **print_format.support_control** can be set by the following printer operations:

```
const min::op min::ascii
const min::op min::latin1
const min::op min::support_all
      min::op min::set_support_control
                      ( const min::support_control & sc )
```

Here **min::ascii** is equivalent to

```
min::set_support_control ( min::ascii_support_control )
```

and similarly for **min::latin1** and **min::support_all**.

The **print_format.display_control** can be set by the following printer operations:

```
const min::op min::graphic_and_sp
const min::op min::graphic_and_hspace
const min::op min::graphic_only
const min::op min::graphic_and_vhspace
const min::op min::display_all
      min::op min::set_display_control
                       ( const min::display_control & dc )
```

Here **min::graphic_and_sp** is equivalent to

```
min::set_display_control ( min::graphic_and_sp_display_control )
```

and similarly for **min::graphic_and_hspace**, **min::graphic_only**, **min::graphic_and_vhspace**, and **min::display_all**.

The **print_format.quoted_display_control** can be set by the following:

```
min::op min::set_quoted_display_control
                 ( const min::display_control & dc )
```

The **print_format.break_control** can be set by the following printer operations:

```
const min::op min::no_auto_break
const min::op min::break_after_space
const min::op min::break_before_all
const min::op min::break_after_hyphens
      min::op min::set_break_control
                       ( const min::break_control & bc )
```

Here **min::no_auto_break** is equivalent to

```
min::set_break_control ( min::no_auto_break_break_control )
```

and similarly for **min::break_after_space**, **min::break_before_all**, and **min::break_after_hyphens**.

The value of **print_format.max_depth** can be set by the following printer operation:

```
min::op min::set_max_depth ( min::uns32 d )
```

The printer operation:

```
const min::op min::verbatim
```

is used when it is desired to copy characters verbatim to the printer buffer. **printer << min::verbatim** is equivalent to:

```
printer << min::noexpand_ht
        << min::support_all
        << min::display_all
        << min::no_auto_break
```

This permits characters in a **printer->file** line to be copied to another printer verbatim. Note that the **min::DISPLAY_EOL** printer operation flag is not affected, as when copying to a printer from some file line terminating NULs are translated to **min::eol** operations on the target printer and may or may not be displayed according to the setting of **min::DISPLAY_EOL** on the target printer. Also note that the operation to undo the results of **min::verbatim** does not exist (you must save and restore **print_format**).

5.17.2 Printer Line Breaks

The **min::set_break** printer operation:

```
const min::op min::set_break
```

can be used to explicitly set a break point: More specifically, **printer<<min::set_break** sets a break point after the last character in the printer file buffer by executing:

```
printer->line_break.offset = printer->buffer->length;
printer->line_break.column = printer->column;
```

If a **min::set_break** operation is executed as a non-initial member of a sequence of **min::leading...**, **min::trailing...**, **min::save_indent**, and **min::set_break** operations, then the **min::set_break** operation is delayed until after the other operations in the sequence have executed. See p148 for details.

Recall from the description of **line_break.line_length** on p128 that when a character representative other than single space, horizontal tab, or spaces representing a horizontal tab is to be inserted into the line, and this would cause the number of columns in the line to exceed **line_break.line_length**, then if

```
line_break.column > line_break.indent
```

a break, consisting of a line end followed by **line_break.indent** spaces, is inserted into the printer file buffer just after the break point (i.e., at **line_break.offset**). Also single space and horizontal tab characters immediately before the inserted line end are deleted.

One typically sets a break point just before printing text that one wants to appear all on one line. Then if a character representative other than single space, horizontal tab, or spaces representing a horizontal tab is to be inserted into the line, and this would cause the number of columns in the line to exceed the **line_break.line_length**, a line end will be inserted at the break point.

For example, if **line_break.line_length** is 72 one might print

```
[ aaa, bbb, ccc, ddd, eee, fff, ggg ]
```

in which **`min::set_break`** was called after every single space character (which could be done automatically using **`break_control.break_after`**). If instead **`line_break.line_length`** was just 14 and **`line_break.indent`** was 2, the same output would print as

```
[ aaa, bbb,
  ccc, ddd,
  eee, fff,
  ggg ]
```

In particular, after '**`[ aaa, bbb, `**' a break point would be set, and then when the third '**`c`**' is about to be printed in column 15, after '**`[ aaa, bbb, cc`**' has already been printed, this break point would be changed to a line end with 2 following single spaces and the single space preceding the line end would be deleted. Similarly a break point would be set after '**`... ddd, `**' and changed when '**`e`**' is about to be printed in column 15, and a break point would be set after '**`... fff, `**' and changed when '**`g`**' is about to be printed in column 15. Break points would also be set after '**`[ `**', '**`... aaa, `**', etc. but these would never be used.

The break point is said to be '***enabled***' whenever

```
line_break.column > line_break.indent
```

and '***disabled***' whenever

```
line_break.column <= line_break.indent
```

A break point must be enabled to be used. Whenever a line end is inserted, the break point is disabled by either setting it to be at the beginning of the line (so **`line_break.column`** is **`0`**) or setting it to just after the **`line_break.indent`** spaces if these are inserted after the line end. Thus **`min::eol`** and the **`min::indent`** operation (see p141) both end by executing an implied **`min::set_break`**, and an implied **`min::set_break`** is also executed just after inserting a line end and **`line_break.indent`** spaces at a break point.

The **`min::left`** and **`min::right`** operations:

```
const min::op min::left ( min::uns32 width )
const min::op min::right ( min::uns32 width )
```

can be used to left or right adjust text in a line. These operations add spaces to make the current printer column equal to **`line_break.column + width`**. The **`min::left`** operation simply appends spaces to the current line. The **`min::right`** operation inserts spaces at the last break point. Both operations end with an implicit **`min::set_break`** that resets the breakpoint. Both operations behave exactly as if spaces were inserted at the appropriate point by **`printer << " "`**, except that **`min::right`** will give undefined results if a horizontal tab has been output since the last break point (which can only happen if the horizontal tab is to represent itself and the **min::EXPAND_HT** printer operation flag is off).

The **min::reserve** operation:

```
const min::op min::reserve ( min::uns32 width )
```

can be used to force a line break if there are fewer than **width** columns remaining in a line. Here the line break consists of a **min::eol** end of line followed by **line_break.indent** single spaces followed by setting a break as per **min::set_break**.

The **min::indent** operation:

```
const min::op min::indent
```

produces a line break if the current printer column is greater than **line_break.indent**, and then (always) inserts single spaces until the current printer column is equal to **line_break.indent** (whether or not a line break was inserted). The operation ends by setting a break point as per **min::set_break**.

Several operations are conditioned on the relative values of **printer->line_break.indent**, the printer indent, and **printer->column**, the printer column:

```
const min::op min::eol_if_after_indent
const min::op min::spaces_if_before_indent
const min::op min::space_if_after_indent
const min::op min::space_if_none
```

The **min::eol_if_after_indent** operation performs an **min::eol** operation if and only if the printer column is after the printer indent. The **min::spaces_if_before_indent** operation performs an **min::indent** operation if and only if the printer column is before the printer indent, inserting single space characters to make the column equal to the indent in this case. The **min::space_if_after_indent** operation outputs a single space character if and only if the printer column is after the printer indent. The **min::space_if_none** operation outputs a single space character if and only if the last thing in the current line is a non-space (more specifically, if the last item has non-zero string class, i.e., it executes the **min::print_space_if_none** function, p149).

Two operations can be used to erase single spaces at the end of the current printer line:

```
const min::op min::erase_space
const min::op min::erase_all_space
```

The first, **min::erase_space**, erases a single space from the end of the current line, if there is such a space, and does nothing otherwise. If it erases a space, **printer->column** is decremented. The second, **min::erase_all_space**, erases as many single spaces as possible from the current line, and decrements **printer->column** by the number of spaces erased.

The line break parameters of a printer may be saved in the **line_break_stack** of the printer, and restored from that stack. This may be done by the following operations:

```
const min::op min::save_line_break
const min::op min::restore_line_break
```

`min::save_line_break` pushes into the line break stack a new element containing the current value of **`line_break`**, while **`min::restore_line_break`** resets **`line_break`** from the top line break stack element and then pops this element from the stack.

In addition, a non-empty line break stack modifies break point behavior in the following way. When a character representative other than single space, horizontal tab, or spaces representing a horizontal tab is to be inserted into the line, and this would cause the number of columns in the line to exceed the printer **`line_break.line_length`** parameter, then if any break stack entry is enabled, the first such entry's **`offset`**, **`column`**, and **`indent`**, are used in place of the printer **`line_break.offset`**, **`line_break.column`**, and **`line_break.indent`** parameters. If this happens, then all later line break stack entries and the printer **`line_break`** are adjusted by adding to offsets the change in offset of the character following the break and adding to columns and indents the change in column of this character.

To be more specific, we say that a line break stack entry **`break_stack[i]`** is '***enabled***' if

```
break_stack[i].column > break_stack[i].indent
```

and '***disabled***' otherwise. The entries are ordered by **`i`**, so the 'first enabled entry' means the enabled entry with lowest **`i`**, which is the bottommost (not topmost) enabled element of the line break stack. The 'later entries' are entries with larger **`i`**. Note that after using an entry to insert a line end, that entry's **`line_break.offset`** and **`line_break.column`** are reset to make its break point be just after the indentation spaces that are inserted after the line end. This sets **`break_stack[i].column`** equal to **`break_stack[i].indent`**, and thus disables **`break_stack[i]`** so it will not be reused. Note also that if <u>no</u> line break stack entry is enabled, break point operation proceeds using the printer **`line_break`** as if the line break stack did not exist. Lastly note that **`break_stack[i].line_length`** is never used during these operations; its only use is to restore **`line_break.line_length`** when the **`break_stack`** is popped.

Management of nested list indentation can be done with the help of the **`line_break_stack`** using the operations:

```
const min::op min::save_indent
const min::op min::restore_indent
```

It is intended that a list header will be printed by first executing a **`min::set_break`**, then printing the header (e.g. **`"[ "`**), then executing **`min::save_indent`**, while the list trailer will be printed by first <u>not</u> executing **`min::set_break`**, then printing the trailer (e.g. **`" ]"`**), then executing **`min::restore_indent`**.

`min::save_indent` executes **`min::save_line_break`** and then sets **`line_break.indent`** to equal the printer **`column`**. **`min::restore_indent`** just executes **`min::restore_line_break`**.

For example, suppose we print the following with **`line_length`** equal to 72:

```
{ aaa, bbb, [ ccc, ddd, eee, ( fff, ggg ), hhh ], iii, jjj }
```

by using code equivalent to the statement:

```
printer << min::set_break << "{ " << min::save_indent
        << min::set_break << "aaa, "
        << min::set_break << "bbb, "
        << min::set_break << "[ " << min::save_indent
        << min::set_break << "ccc, "
        << min::set_break << "ddd, "
        << min::set_break << "eee, "
        << min::set_break << "( " << min::save_indent
        << min::set_break << "fff, "
        << min::set_break << "ggg "
        << " ), " << min::restore_indent
        << min::set_break << "hhh "
        << " ], " << min::restore_indent
        << min::set_break << "iii, "
        << min::set_break << "jjj "
        << " }" << min::restore_indent
        << min::eol;
```

Then if instead **`line_length`** was just 34 the same output would print as

```
{ aaa, bbb,
  [ ccc, ddd, eee, ( fff, ggg ),
    hhh ], iii, jjj }
```

After printing the 34 characters

```
{ aaa, bbb, [ ccc, ddd, eee, ( fff
```

a line break is triggered just before printing the '**`,`**' following '**`fff`**'. The first element in the line break stack was pushed just after printing '**`{ `**'and is disabled as it is at the beginning of the line. The second element, pushed just after printing '**`[ `**', has **`column`** just after '**`bbb, `**', and **`indent`** just after '**`{ `**', and so it is enabled and used, and in the process it is disabled. At this point the output is

```
{ aaa, bbb,
  [ ccc, ddd, eee, ( fff
```

The third element was pushed after printing '**`( `**' and it is updated when the second element is used so it still faithfully reflects the point just after '**`( `**'. Next this third element is popped after printing ‘ `),` ’. Then when the second ‘**`h`**’ is about to be printed, neither of the two break points left in the line break stack are enabled, so the non-stack break point before the first ‘**`h`**’ is used. At this point the current **`indent`** is that which was saved in and restore from the second element, and this indent is just after ‘`[` ’.

If a **`min::save_indent`** operation is executed as a non-initial member of a sequence of **`min::leading...`**, **`min::trailing...`**, **`min::set_break`**, and **`min::save_indent`** operations, then the **`min::save_indent`** operation is delayed until after the other operations in the sequence have executed. See p148 for details.

The ***begin message*** and ***end message*** operations can be used to surround the body of an error message or other message:

```
const min::op min::bom
const min::op min::eom
```

`min::bom` executes both a **`min::save_indent`** and a **`min::save_print_format`** operation. **`min::eom`** executes in order a **`min::restore_indent`**, a **`min::bol`**, a **`min::flush_id_map`** (p170) if the printer operation flag **`FLUSH_ID_MAP_ON_EOM`** printer operation flag is on, and lastly a **`min::restore_print_format`**. Note that the print format and its operation flags in effect just before **`min::eom`** is executed govern all the operations executed by **`min::eom`**.

For example, the following might be used to print an error message:

```
printer << "ERROR: " << min::bom;
. . . . . .
// Print contents of error message
. . . . . .
printer << min::eom;
```

Here the error message will be indented by the number of columns consumed by ‘**`ERROR: `**’, and any format parameters changed while printing the error message body will be restored at the end of the message by the **`min::eom`**.

`min::bom` will set **`line_break.indent`** to the current value of **`column`**, but if a different indent is desired, it can be set by following **`min::bom`** with **`min::set_indent`** (p136) or one of the operations:

```
min::op min::place_indent ( min::int32 offset )
min::op min::adjust_indent ( min::int32 offset )
```

These operations set the indent in **`line_break.indent`** to the sum of ‘**`offset`**’ and either the printer ‘**`column`**’ for **`min::place_indent`** or the indent itself for **`min::adjust_indent`**. If the indent would be set to a negative value, **`0`** is used instead.

5.17.3 Leading and Trailing Separators

Consider a lexical scanner in which proto-lexemes that contain no space characters are first scanned, and then '***leading separators***' such as the opening single quote (`‘`) and dollar sign (`$`) are removed from the beginning of the proto-lexeme, '***trailing separators***' such as closing single quote (`’`) and comma (`,`) are removed from the end of the proto-lexeme, and anything left over of the proto-lexeme is designated to be a '***middle lexeme***'. So the proto-lexeme `$10,000,` translates to the three lexemes `$`, `10,000`, and comma (`,`). Note that the first comma is part of the middle lexeme and the last is a trailing separator by itself.

The same effect can be somewhat achieved by a different kind of scanner that declares a `$` character to be a separator only if it is followed by a digit and a comma to be a separator unless it is surrounded by digits.

Suppose you want to print the 3 element list of strings, `$ 10,000` comma (`,`), and you know the first string may be a leading separator, the second may be middle, and the last may be trailing. But maybe not.

You can do this with the code:

```
printer << min::new_str_gen ( "$")
        << min::leading
        << min::new_str_gen ( "10,000" )
        << min::trailing
        << min::new_str_gen ( "," );
```

or if the `min::FORCE_PGEN` flag is on, with the equivalent code:

```
printer << "$" << min::leading << "10,000" << min::trailing << ",";
```

(The `min::FORCE_PGEN` flag is mostly of use in testing and debugging `min::leading/trailing`.)

Here we have used the printer operations:

```
min::op min::leading
min::op min::trailing
```

Each of these denotes a space character which may be output or not according to its context.

To give as second example, consider the code, with `min::FORCE_PGEN` flag on:

```
printer << "100" << min::trailing << "," << " " << "200";
printer << "100" << min::trailing << "#" << " " << "200";
```

We want the first line to output '`100, 200`' in which `min::trailing` does not output a space, while we want the second line to output '`100 # 200`' in which `min::trailing` does output a space.

And to consider one last example, consider the code, with `min::FORCE_PGEN` flag on:

```
printer << "("   << min::leading << "100" << min::trailing <<  ")";
printer << "(+"  << min::leading << "100" << min::trailing << "+)";
```

We want the first line to output '`(100)`' in which there are no spaces, while we want the second line to output '`(+ 100 +)`' in which there are two spaces.

So how do we compute whether **`min::leading/trailing`** output a space?

First, we run every string that is output though a string classifier. If we use **`min::standard_str_classifier`** on the strings in the above examples we get:

String	Class
`"$"`	`IS_LEADING+IS_GRAPHIC`
`"10,000"`	`IS_GRAPHIC`
`","`	`IS_GRAPHIC+IS_TRAILING`
`"100"`	`IS_GRAPHIC`
`" "`	`0`
`"200"`	`IS_GRAPHIC`
`"#"`	`IS_GRAPHIC`
`"("`	`IS_LEADING+IS_GRAPHIC`
`")"`	`IS_GRAPHIC+IS_TRAILING`
`"(+"`	`IS_GRAPHIC`
`"+)"`	`IS_GRAPHIC`

Then **`min::leading`** produces a single space if both the string before it and the string after it have the **`IS_GRAPHIC`** flag and the string before it does not have the **`IS_LEADING`** flag, and similarly **`min::trailing`** produces a single space if both the string before it and the string after it have the **`IS_GRAPHIC`** flag and the string after it does not have the **`IS_TRAILING`** flag.

Thus **`"100" << min::trailing << ","`** does not output a space while **`"100" << min::trailing << "#"`** does output a space.

There are a number of extra details.

First, if the above had not been printed with the **`FORCE_PGEN`** flag on, then **`min::null_str_classifier`** would have been used instead of **`min::standard_str_classifier`**, and the string classes would have been the same but without the **`IS_LEADING`** or **`IS_TRAILING`** flags, so that the **`min::leading`** and **`min::trailing`** operations would have all produced spaces.

Second, if the **`min::FORCE_SPACE`** flag is on, **`min::leading`** and **`min::trailing`** always output a space unless either the preceding or following string does not have the **`IS_GRAPHIC`** flag. But there are variant printer operations:

```
min::op min::leading_always
min::op min::trailing_always
```

which are just like **`min::leading/trailing`** except they ignore the **`min::FORCE_SPACE`** flag. So:

```
printer << "(" << min::leading
        << "100" << min::trailing_always << ", " << "200"
        << min::trailing << ")";
```

prints '**`(100, 200)`**' if **`min::FORCE_SPACE`** is off, and prints '**`( 100, 200 )`**' if **`min::FORCE_SPACE`** is on. Some people prefer extra spacing next to parentheses, and **`min::FORCE_SPACE`** accommodates this.

Third, above we have dealt with 'strings', but the printer actually deals with '***print items***'. A character string is a print item, but so is a number, and so are **`min::gen`** values that are not labels or objects. Labels and objects are '***composite***' and output multiple print items. The default rule is that **`min::gen`** string values are print items classified by

`printer->print_format.gen_format->str_format.str_classifier`

if it exists, whereas in all other cases print items are classified by the **`min::null_str_classifier`**. This last classifier just separates out whitespace strings, to which it gives the class `0`, from other strings, to which it gives the class `IS_GRAPHIC`.

Fourth, a **`min::gen`** string whose string class contains the `NEEDS_QUOTES` flag, or does not contain the `IS_GRAPHIC` flag, is quoted, and the whole quoted string is treated as a single print item with just the `IS_GRAPHIC` flag.

In general print items may be separated by consecutive sequences of one or more of the print operations :

`min::leading`
`min::leading_always`
`min::trailing`
`min::trailing_always`
`min::save_indent`
`min::set_break`

Such a sequence of print operations outputs either a single space or nothing according to the following rules:

1. If either the preceding or following print items do not have the `IS_GRAPHIC` flag, no space is emitted.

2. Otherwise if either **`min::leading`** or **`min::trailing`** is in the sequence and the `FORCE_SPACE` printer operation flag is on, a space is output.

3. Otherwise if either **min::leading** or **min::leading_always** is in the sequence and the preceding print item has the **IS_LEADING** flag, no space is output.

4. Otherwise if either **min::trailing** or **min::trailing_always** is in the sequence and the following print item has the **IS_TRAILING** flag, no space is output.

5. Otherwise a space is output.

If the operation sequence contains a **min::save_indent** or **min::set_break** operation that is preceded by a **min::leading...** or **min::trailing...** operation in the sequence, this **min::save_indent** or **min::set_break** operation is delayed until the space is output or a decision not to output the space has been made. The sequence must not contain more than one **min::save_indent** operation, not counting the first operation in the sequence, due to implementation limitations.

5.17.4 Printer Strings

Sequence of print items can be represented by '***printer strings***':

```
typedef min::printer (* min::pstring ) ( min::printer printer )
                 min::printer operator <<S
                                   ( min::printer printer,
                                     min::pstring pstring )
```

where the **<<** operator is defined so that if **p** is a **min::pstring** then

printer<<p is equivalent to **(*p)(printer)**

min::pstring functions usually call the following primitive string functions:

```
min::printer min::print_item
                    ( min::printer printer,
                      const char * p,
                      min::unsptr n,
                      min::uns32 columns,
                      min::uns32 str_class = min::IS_GRAPHIC )
min::printer min::print_space
                    ( min::printer printer,
                      min::unsptr n = 1 )
min::printer min::print_space_if_none
                    ( min::printer printer )
min::printer min::print_erase_space
                    ( min::printer printer,
                      min::uns32 n = 1 )
min::printer min::print_leading
                    ( min::printer printer )
min::printer min::print_trailing
                    ( min::printer printer )
min::printer min::print_leading_always
                    ( min::printer printer )
min::printer min::print_trailing_always
                    ( min::printer printer )
```

WARNING: The character string **p** argument to **min::print_item** may not be relocatable.

The **min::print_item** function prints a single print item that is a UTF-8 string of **n chars** taking the given number of columns and having the given string class. However, this function ignores the **printer->print_format**, and in particular the **support_control**, **display_control**, and **break_control** parts. Instead only the following is done in order:

1. If **n == 0** this function does nothing.
2. The string class is used to determine whether preceding **min::leading/trailing...** operations should produce a space.
3. The string class is saved for use in processing subsequent **min::leading/trailing...** operations.
4. The UTF-8 string is copied directly to the printer buffer.
5. The printer column is updated.
6. If the printer column exceeds the printer line length, and the printer line break information warrants, a line break is inserted.

Therefore the UTF-8 string should encode only characters that are graphic, that are supported according to the printer format **support_control**, that display as themselves according to the printer format **display_control**, and that do not cause breaks according to the printer format **break_control**. Short items containing only ASCII graphics characters typically meet these qualifications except possibly for the **break_control** in cases where a break is not really necessary (the item being short).

Therefore **min::print_item** is mostly of use for inserting punctuation.

The **min::print_space** function outputs **n** single spaces as an item with **0** string class. It does the same thing as **print_item** would do it the item were **n** single spaces, except it does not insert line breaks and it does do the printer format **break_control.break_after** processing, so it respects the common case where breaks are inserted automatically after single spaces.

The **min::print_space_if_none** function outputs a single space as per **print_space** if and only if the string class of the previous item is not **0**, and does nothing if it is **0**.

The **min::print_erase_space** function erases single spaces from the end of the current line, decrementing **printer->column** by the number of spaces erased. It erases at most **n** spaces, and stops when the current line does not end in a single space.

The **min::print_leading/trailing...** functions do the same thing as the corresponding **min::leading/trailing...** printer operations.

Sometimes it is useful to perform parts of **min::print_item** using the functions:

```
min::printer min::print_item_preface
                    ( min::printer printer,
                      min::uns32 str_class )
min::printer min::print_chars
                    ( min::printer printer,
                      const char * p,
                      min::unsptr n,
                      min::uns32 columns )
```

WARNING: The character string **p** argument to **min::print_chars** may not be relocatable.

The **min::print_item_preface** function does the string class processing steps of **min::print_item**, while the **min::print_chars** function does the string buffer and column processing steps. Both functions assume the string is of non-zero length. The main use of these is to concatenate several strings using multiple calls to **min::print_chars** after a single call to **min::print_item_preface**. Its also possible to optimize by replacing a call to **min::print_item** by a call to **min::print_chars** if that call is sandwiched between calls to **min::print_space**.

Lastly a **min::ustring** can be printed using

```
min::printer min::print_ustring
                    ( min::printer printer,
                      min::ustring s )
```

This performs the same actions as **min::print_chars** but uses the length and number of columns encoded in the **ustring** argument. It also does nothing if **s** is **NULL**.

5.17.5 Printing File Lines and Phrases

The following function uses **min::line** to print a line as part of an error message:

```
min::uns32 min::print_line^S
                  ( min::printer printer,
                    min::file file,
                    min::uns32 line_number,
                    const min::line_format * line_format = NULL )
```

The **file** line with the given **line_number** is printed on a single **printer** line with **line_format.op_flags** temporarily merged into **printer->print_format.op_flags** during the printing (see p107), and the number of columns in the result, including any end of line representation printed by the **min::DISPLAY_EOL**, is returned.

If **line_format** is **NULL**, **file->line_format** is used in its place. If that is also **NULL**, **printer->print_format.line_format** is used (this last may not be **NULL**).

If a line is unavailable (i.e., **min::line** returns **min::NO_LINE**, see p114) and **line_number** is not equal **file->file_lines**, then **printer->print_format.line_format->unavailable_line** is printed on a line by itself instead, and **0** is returned, if **printer->print_format.line_format->unavailable_line** is not **NULL**. But if **printer->print_format.line_format->unavailable_line** is **NULL**, nothing is done except to return **min::NO_LINE**.

If a line would print as blank, and if **printer->print_format.line_format->blank_line** is not **NULL**, then **printer->print_format.line_format->blank_line** is printed instead of the line, and **0** is returned. More specifically, a line is deemed to print as blank if it only contains horizontal space characters (characters with the **min::IS_HSPACE** flag: see p57) and **min::DISPLAY_NON_GRAPHIC** is not in effect.

If there is no line because **line_number == file->file_lines**, then any partial line at the end of the file buffer is printed, immediately followed by **printer->print_format.line_format->end_of_file**, all on one line, and the number of columns used to print the partial line is returned. However, if **printer->print_format.line_format->end_of_file** is **NULL**, then if there is a partial line, only that is printed, on a single line without any visible ending line feed, and if there is no partial line, nothing is printed and **NO_LINE** is returned.

Note that the **line_format** is not used to print **line_format->unavailable_line**, **line_format->blank_line**, or **line_format->end_of_file**, or any line feed following these, and it is best if these special strings consist of nothing but graphic ASCII characters and single spaces. Also, these ASCII strings may not be unprotected body pointers, as printing may relocate bodies.

In all cases where anything is printed, exactly one complete print line is printed.

Error messages typically consist of a description of the error followed by the lines involved printed with the above function. Often the portions of the lines involved are marked by printing '^' characters under them. The error description before the lines typically ends with the name of the file containing the lines and the numbers of the lines, which can be produced by the code:

```
(constructor) min::pline_numbers
                        ( min::file file,
                          min::uns32 first, min::uns32 last )
```

```
min::printer operator <<^S
                    ( min::printer printer,
                      min::pline_numbers const & pline_numbers )
```

An example use is

```
printer << ... error description ...
        << min::pline_numbers ( file, first, last )
        << min::eol;
for ( min::uns32 line_number = first;
      line_number <= last,
      ++ line_number )
    min::print_line ( printer, file, line_number );
```

The line numbers printed by **min::pline_number** are `1` greater than the line numbers given in the arguments, so the line number of the first line of a file is denoted by a `0` argument which is printed as `1`.

There is a better way to print the lines of a file within an error message if the positions of the erroneous text in the file are known and are encoded in **min::position** structures:

```
struct min::position
                {
                  min::uns32 line;
                  min::uns32 offset;
                };
```

Here **line** is the number of full lines before the line containing the position and **offset** is the number of bytes before the position in the line.

A missing value and the following operations are defined for **min::position** structures:

```
const min::position min::MISSING_POSITION = { 0xFFFFFFFF, 0xFFFFFFFF }
                    bool operator ==
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator !=
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator <
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator <=
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator >
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator >=
                             ( const min::position & p1,
                               const min::position & p2 )
                    bool operator (bool)
                             ( const min::position & p )
```

A **`min::position`** is less than another if its line is less or its line is equal and its offset is less. A **`min::position p`** converts to **`true`** if **`p.line != 0xFFFFFFFF`** and to **`false`** otherwise. This permits special values other than **`min::MISSING_POSITION`** to be defined with **`line`** equal to **`0xFFFFFFFF`**.

Two **`min::position`** structures are needed to encode the boundaries of a piece of text in a file. Such a piece of text is called a '***phrase***', and its positions are encoded in a **`min::phrase_position`** structure:

```
struct min::phrase_position
                {
                  min::position begin;
                  min::position end;
                };
```

Here '**`begin`**' encodes the position of the first character of the phrase in the file, and '**`end`**' encodes the position of the first character after the phrase in the file. If the phrase is empty, **`begin == end`**, and the position referenced by both '**`begin`**' and '**`end`**' is the position of the empty phrase in the file.

To print the line numbers of a phrase one can use the constructor:

```
(constructor) min::pline_numbers
                  ( min::file file,
                    min::phrase_position const & position )
```

which uses **position.begin.line** and **position.end.line** as line numbers, except that when

position.end.offset == 0
and
position.end.line > position.begin.line

then **position.end.line - 1** is used in place of **position.end.line**.

To print a phrase one can use the function:

```
min::uns32 min::print_phrase_lines S
                ( min::printer printer,
                  min::file file,
                  min::phrase_position const & position,
                  const min::line_format * line_format = NULL )
```

The lines containing the phrase are printed, and under each phrase character a mark is printed. The lines are printed as per **min::print_line**. An empty phrase is marked as if it were 1-column wide, that column being at the empty phrase position.

The mark is the character **line_format->mark**. If it is **0**, marking lines will be omitted. This permits just printing the lines containing a phrase, without marking the phrase.

The column number of **position.begin** is returned (this is **0** for the first column). If position bounds the non-whitespace portion of some lines, this is the indent of that portion in the first of these lines.

An example using a **phrase_position** is

```
printer << ... error description ...
        << min::pline_numbers ( file, phrase_position )
        << min::eol;
min::print_phrase_lines ( printer, file, phrase_position );
```

Internally **min::print_phrase_lines** makes the call:

```
min::print_line_column
    ( file, position,
      printer->print_format, line_format );
```

which uses the function:

```
min::uns32 min::print_line_column^S
                    ( min::file file,
                      min::phrase_position const & position,
                      const min::print_format & print_format,
                      const min::line_format * line_format = NULL )
```

that, given a file, a position within that file, print format, and a line format, returns the number of columns before the position in a **min::print_line** printout of the line. To do this, **min::print_line_column** calls **min::pwidth** (see p136).

The above functions use **line_format** to control how lines are printed and therefore which column a position corresponds to. Specifically, if **min::DISPLAY_EOL** is on in **line_format**, **<NL>** will print at the end of a line if **min::DISPLAY_PICTURE** is off, and ${}^{N}{}_{L}$ will print if **min::DISPLAY_PICTURE** is on. Also, non-graphic, non-horizontal-space characters always print as pictures (e.g., ${}^{F}{}_{F}$) if **min::DISPLAY_PICTURE** is on, and otherwise as their names with prefix and postfix determined by **printer->print_format.char_name_format** (e.g., **<FF>**). If in addition **min::DISPLAY_NON_GRAPHIC** is on, horizontal space characters will also print as pictures (e.g., ␣ and ${}^{H}{}_{T}$), if **min::DISPLAY_PICTURE** is on, and otherwise as as their names (e.g., **<SP>** and **<HT>**).

In the above functions, if the **line_format** is **NULL**, then **file->line_format** is used, but if that is also **NULL**, then **printer->print_format.line_format** is used (this last may <u>not</u> be **NULL**).

There is a type of packed vector, the '***phrase position vector***', that is specialized to record the phrase positions of elements of any linear vector:

```
typedef min::packed_vec_ptr<min::phrase_position_vec_header,
                            min::phrase_position>
                 min::phrase_position_vec
typedef min::packed_vec_insptr<min::phrase_position_vec_header,
                               min::phrase_position>
                 min::phrase_position_vec_insptr
```

```
min::phrase_position_vec_insptr min::init^S
        ( min::ref<min::phrase_position_vec_insptr> vec,
          min::file file,
          min::phrase_position const & position,
          min::uns32 max_length )
```

```
    const min::uns32 vpp->length
     const min::file vpp->file
min::phrase_position vpp->position
min::phrase_position vpp[i]
```

A phrase position vector is associated with some other data vector. For example, a phrase position vector may be the value of the **.position** attribute of an object (see below), in which case it will be associated with the attribute vector of the object. If **vpp** is a phrase position vector pointer, **vpp->position** is the position of the phrase that is encoded in the data vector, **vp[i]** is the position of the phrase that is encoded in the **i+1**'st element of the data vector, and **vpp->file** is the file containing these phrases. **vpp->length** is, as per the definition of packed vectors, the number of elements in the phrase position vector, and should match the number of elements in the data vector. The **min::init** function above can be used to create or reinitialize a **min::phrase_position_vec**, and sets the **file**, **position**, and initial maximum length of the phrase position vector.

A MIN object may have a **.position** attribute equal to a phrase position vector. If an object does, the phrase position pointer is returned by the following function:

```
min::phrase_position_vec min::position_of ( min::obj_vec_ptr & vp )
```

Here the object is identified by the **min::obj_vec_ptr** which is being used to access the object elements; see p181. If the object has no **.position** attribute, or if the attribute is not a **min::phrase_position_vec** value, **min::NULL_STUB** is returned.

5.17.6 Printing File Lines and Phrases to HTML

UTF-8 encoded file lines and part lines may be printed in a form suitable for including in HTML **<pre>** tagged text. The UTF-8 string is printed as is except:

1. **<** is printed as **<**.
2. **>** is printed as **>**.
3. **&** is printed as **&**.
4. A horizontal tab is expanded to single spaces.
5. A control character (in UNICODE class C) that is not a horizontal tab, line feed, or carriage return is printed as **<0xxxx>** where **xxxx** is the hexadecimal represention of the character's code.

printer->columns is updated by adding one for characters without the **min::IS_NON_SPACING** character flag (p55), except for horizontal tabs which update the column assuming tab stops every 8 columns.

The HTML printing functions are:

```
min::printer min::html_print_unicode
                        ( min::printer printer,
                          min::unsptr n,
                          min::ptr<const min::Uchar> p )
```

Note the vector of **min::Uchar**'s may be in a relocatable body.

```
min::printer min::html_print_cstring
                        ( min::printer printer,
                          min::unsptr n,
                          const char * str )
```

Note the string length **n** is explicitly given.

The various printer flags and formats are ignored by these functions.

5.18 Printing General Values

min::gen values are printed using the **min::pgen** printer operations:

```
min::op min::pgen ( min::gen v )
min::printer operator <<^S
                ( min::printer printer, min::gen v )
min::op min::pgen
                ( min::gen v,
                  const min::gen_format * gen_format )
min::op min::set_gen_format
                ( const min::gen_format * gen_format )
min::op min::pgen_name ( min::gen v )
min::op min::pgen_quote ( min::gen v )
min::op min::pgen_never_quote ( min::gen v )
```

The expression '**printer << min::pgen(v)**' prints the **min::gen** value **v** to the **printer** using the **printer**'s **gen_format** value

printer->print_format.gen_format

to control the printing.

'**printer << v**' can be used as the equivalent of '**printer << min::pgen(v)**'. The following function has the same effect:

```
min::printer min::print_gen
                        ( min::printer printer,
                          min::gen v )
```

The expression '**printer << min::pgen(v,f)**' is similar but uses **f** as the **gen_format** value. The following function with 3 arguments has the same effect:

```
min::printer min::print_gen
                        ( min::printer printer,
                          min::gen v,
                          const min::gen_format * f,
                          bool disable_mapping = false )
```

The fourth **disable_mapping** argument disables use of defined formats (5.18.7 *p*171) for printing the top level of **v** if it is set to **true**. This argument does not affect the printing of components of **v**, and is normally only used by defined format printing functions.

The expression '**printer << min::set_gen_format(f)**' sets

printer->print_format.gen_format

to the value '**f**'. The default value of

printer->print_format.gen_format

is **min::compact_gen_format** (p159).

min::pgen_name(v), **min::pgen_quote(v)**, and **min::pgen_never_quote(v)** are respectively shorthand for

```
min::pgen ( v, min::name_gen_format )
min::pgen ( v, min::always_quote_gen_format )
min::pgen ( v, min::never_quote_gen_format )
```

A **min::gen_format** cannot be stored in a relocatable body, and is intended to be a static C++ structure. In making new versions, one must copy an existing **min::gen_format** and then change particular members in the copy, because one cannot count on all the members being organized in a particular way. Copies can be made in the stack.

The detailed structure and available standard values for general value formats is as follows:

```
typedef min::printer ( * min:: pgen_function )
    ( min::printer printer,
      min::gen v,
      const min::gen_format * gen_format,
      bool disable_mapping )
```

```
struct min::gen_format
    {
      min::pgen_function pgen;

      // The following are used by min::standard_pgen:
      //
      const min::num_format *  num_format;
      const min::str_format *  str_format;
      const min::lab_format *  lab_format;
      const min::special_format *  special_format;
      const min::obj_format *  obj_format;

      // The following is NOT used by min::standard_pgen:
      //
      const void *  non_standard;
    };

const min::gen_format * min::compact_gen_format:
    & min::standard_pgen                   // pgen
    min::long_num_format                   // num_format
    min::quote_separator_str_format        // str_format
    min::bracket_lab_format                // lab_format
    min::bracket_special_format            // special_format
    min::compact_obj_format                // obj_format
    NULL                                   // non_standard

const min::gen_format * min::line_gen_format:
    // Same as min::compact_gen_format except for:
    min::line_obj_format                   // obj_format

const min::gen_format * min::paragraph_gen_format:
    // Same as min::compact_gen_format except for:
    min::paragraph_obj_format              // obj_format

const min::gen_format * min::compact_value_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_value_str_format            // str_format

const min::gen_format * min::compact_id_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_value_id_str_format         // str_format
    min::compact_id_obj_format             // obj_format
```

```
const min::gen_format * min::id_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_value_id_str_format          // str_format
    min::id_obj_format                      // obj_format

const min::gen_format * min::id_map_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_value_str_format             // str_format
    min::isolated_line_id_obj_format        // obj_format

const min::gen_format * min::name_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_value_str_format             // str_format
    min::name_lab_format                    // lab_format

const min::gen_format * min::leading_always_gen_format:
    // Same as min::compact_gen_format except for:
    min::standard_str_format                // str_format
    min::leading_always_lab_format          // lab_format

const min::gen_format * min::trailing_always_gen_format:
    // Same as min::compact_gen_format except for:
    min::standard_str_format                // str_format
    min::trailing_always_lab_format         // lab_format

const min::gen_format * min::always_quote_gen_format:
    // Same as min::compact_gen_format except for:
    min::quote_all_str_format               // str_format

const min::gen_format * min::never_quote_gen_format:
    // Same as min::compact_gen_format except for:
    NULL                                    // str_format
    min::name_lab_format                    // lab_format
    min::name_special_format                // special_format
```

A general value format can be any structure whose first member is:

```
min::printer ( * pgen )
     ( min::printer printer,
       min::gen v,
       const min::gen_format * gen_format,
       bool disable_mapping )
```

This first member is called with the general value format itself as third argument to execute the **pgen** operation. So in effect a general value format is a '***closure***' whose first member is the function to execute and whose remaining members are data for this function.

The standard first member function value is:

```
min::printer min::standard_pgen
                        ( min::printer printer,
                          min::gen v,
                          const min::gen_format * gen_format,
                          bool disable_mapping = false )
```

The **min::gen_format** structure described above is what is expected by this '***standard* pgen**'. In this document it is this format and the standard **pgen** function **min::standard_pgen** that are being described unless specified otherwise.

The **disable_mapping** argument disables use of defined formats (5.18.7 *p171*) for printing the top level of **v** if it is set to **true**. This argument does not affect the printing of components of **v**.

Most members of a general value format control the printing of different kinds of general values. For example, the **str_format** member controls the printing of string general values. We will describe how each member is used in the following subsections.

The **non_standard** member is an exception. It is not used by the **min::standard_pgen** function and is available for use as a pointer to extra format data by non-standard **pgen** functions.

5.18.1 Printing Numeric General Values

The format for printing numeric **min::gen** values is:

```
struct min::num_format
    {
      const char *        int_printf_format;
      min::float64        non_float_bound;
      const char *        float_printf_format;
      const min::uns32 *  fraction_divisors;
      min::float64        fraction_accuracy;
    };
```

const min::num_format * min::**short_num_format**:

```
    "%.0f"                  // int_printf_format
    1e7                     // non_float_bound
    "%.6g"                  // float_printf_format
    NULL                    // fraction_divisors
    0                       // fraction_accuracy
```

const min::num_format * min::**fraction_num_format**:

```
"%.0f"                  // int_printf_format
1e7                     // non_float_bound
"%.6g"                  // float_printf_format
min::standard_divisors  // fraction_divisors
1e-9                    // fraction_accuracy
```

const min::num_format * min::**long_num_format**:

```
"%.0f"                  // int_printf_format
1e15                    // non_float_bound
"%.15g                  // float_printf_format
NULL                    // fraction_divisors
0                       // fraction_accuracy
```

const min::uns32 * min::**standard_divisors**:

```
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
16, 32, 64, 128, 256, 512, 1024, 0
```

The following function is called when a numeric **min::gen** value is printed, and can be used to print a **min::float64** value according to the above formats:

```
min::printer min::print_num
                    ( min::printer printer,
                      min::float64 value,
                      const min::num_format * num_format = NULL )
```

If the third argument is not provided, **printer->print_format.num_format** is used.

Given an **min::gen** numeric value **g**, **printer << g** and **printer << min::pgen (g)** are both equivalent to

```
min::print_num ( printer, min::float_of ( g ) )
```

printer << min::pgen (g, f) is equivalent to

```
min::print_num ( printer, min::float_of ( g ), f->num_format )
```

The algorithm used by **min::print_num** is as follows when the **min::num_format** argument is **nf**:

1. If the absolute value of the **value** argument is strictly less than **nf->non_float_bound**, then:

   (a) If **value** is an integer, it is printed using the **nf->int_printf_format**.

(b) If **`nf->fraction_divisors`** is not `NULL` and within this vector a divisor `D` can be found such that for suitable integers `I` and `N`, `0 <= N < D` and

`|value - I - N/D| < nf->fraction_accuracy`

then '`I N/D`' is printed if `I>0`, '`N/D`' is printed if `I == 0`, '`-M/D`' is printed if `I == -1` where `M = D-N`, and '`-J M/D`' is printed if `I < -1` where `J = I+1`.

2. If **`value`** has not been printed by the above, it is printed using **`nf->float_printf_format`**.

The list of allowed divisors `D` in **`nf->fraction_divisors`** must be in ascending order and be terminated by a `0`.

5.18.2 Printing String General Values

The formats for printing string **`min::gen`** values are:

```
struct min::str_format
    {
      min::str_classifier   str_classifier;
      min::ustring          str_break_begin;
      min::ustring          str_break_end;
      min::quote_format     quote_format;
      min::uns32            id_strlen;
    };
```

const min::str_format * min::**`standard_str_format`**:

```
    min::standard_str_classifier          // str_classifier
    (min::ustring) "\x01\x01" "#"         // str_break_begin
    (min::ustring) "\x01\x01" "#"         // str_break_end
    min::standard_quote_format            // quote_format
    0xFFFFFFFF                            // id_strlen
```

const min::str_format * min::**`quote_separator_str_format`**:

```
    // Same as min::standard_str_format except for:
    min::quote_separator_str_classifier    // str_classifier
```

const min::str_format * min::**`quote_value_str_format`**:

```
    // Same as min::standard_str_format except for:
    min::quote_value_str_classifier // str_classifier
```

const min::str_format * min::**quote_all_str_format**:

```
// Same as min::standard_str_format except for:
min::quote_all_str_classifier                // str_classifier
0                                            // id_strlen
```

```
struct min::quote_format
    {
      min::ustring    str_prefix;
      min::ustring    str_postfix;
      min::ustring    str_postfix_name;
    };
```

```
const min::quote_format min::standard_quote_format:
    (min::ustring) "\x01\x01" "\""   // str_prefix
    (min::ustring) "\x01\x01" "\""   // str_postfix
    (min::ustring) "\x03\x03" "<Q>"  // str_postfix_name
```

The **ustring** members of **min::str_format** and **min::quote_format** must not contain non-graphic characters (e.g., no spaces).

The following function prints a string of of **n** UNICODE **Uchar**'s pointed at by **p**. Functions are given below that print UTF-8 encoded strings by first copying the UTF-8 string into the stack unpacking the UTF-8 characters into **Uchar**'s and then calling this function.

```
min::printer min::print_unicode
                    ( min::printer printer,
                      min::unsptr n,
                      min::ptr<const min::Uchar> p,
                      const min::str_format * str_format = NULL )
```

Note the vector of **min::Uchar**'s may be in a relocatable body.

If **str_format** is **NULL**, the string is printed as is without quoting or breaking the string into parts. Otherwise **str_format->str_classifier** is used to compute the string class of the string. If this string class has the **min::NEEDS_QUOTES** flag or does not have the **min::IS_GRAPHIC** flag, the string is printed by the **min::print_quoted_unicode** function (p165) with surrounding quotes and special representations of control characters. Otherwise if the string class has the **min::IS_BREAKABLE** flag, the string is printed by the **min::print_breakable_unicode** function (p165) that may break the string into parts.

Otherwise, the string is considered to be a print item (p147). As such its string class is computed by **str_format->str_classifier** if **str_format** is not **NULL**, and by **min::null_str_classifier** otherwise, and this class is used to determine whether or not to insert a space before the string: see 5.17.3 ^{p145}.

The following function prints a string with surrounding quotes:

```
min::printer min::print_quoted_unicode
                    ( min::printer printer,
                      min::unsptr n,
                      min::ptr<const min::Uchar> p,
                      const min::str_format * str_format )
```

This function assigns the string class containing just the **IS_GRAPHIC** flag to the whole quoted string for the purpose of determining whether a space is to be inserted before and/or after the quoted string: see 5.17.3 [p145].

Then this function uses **str_format->quote_format** to determine how the string is quoted. Neither this nor **str_format** may be **NULL**. The quoted string is begins with the **str_prefix** member (e.g., **"**) and ends with the **str_postfix** member (e.g., **"**). If the string being quoted itself contains a copy of the **str_postfix** member (e.g., **"**), that copy is replaced by the **str_postfix_replacement** member (e.g., **<Q>**).

The quoted string is printed without any breaks inside it (break control is suspended). However, if the quoted string is too long to fit in the columns between **printer->line_break.indent** and **printer->line_break.line_length** (p128), and if the printer **print_format.op_flags min::DISABLE_STR_BREAKS** flag is not set, then the quoted string is broken into multiple quoted partial strings, each fitting within the columns, with all quoted partial strings but the last immediately followed by **str_format->str_break_begin** (e.g., **#**) and all quoted partial strings but the first immediately preceded by **str_format->str_break_end** (e.g., **#**). A single space is inserted between consecutive quoted partial strings (e.g., between the two **#**'s), with a break after each of these spaces.

Lastly, the characters of the string being quoted (but not those of **str_format->quote_format.str_...**) are printed using **printer->print_format.quoted_display_control** instead of **printer->print_format.display_control**. E.g., only graphic characters may be allowed to print as themselves, and everything else may be represented by a character name (or if the **min::DISPLAY_PICTURE** printer operation flag is set, by a character picture).

The following function prints a string that can be broken into parts by using the string format **str_break_begin** and **str_break_end** members. In this case the string should be a word, number, or mark consisting of all graphic characters.

```
min::printer min::print_breakable_unicode
                    ( min::printer printer,
                      min::unsptr n,
                      min::ptr<const min::Uchar> p,
                      const min::str_format * str_format )
```

This function assigns the string class containing just the **IS_GRAPHIC** flag to the whole string for the purpose of determining whether a space is to be inserted before and/or after the string: see 5.17.3 [p145].

The string is printed without any breaks inside it (break control is suspended). However, if the string is too long to fit in the columns between `printer->line_break.indent` and `printer->line_break.line_length` (p128), and if the printer `print_format.op_flags` `min::DISABLE_STR_BREAKS` flag is not set, then the string is broken into multiple partial strings, each fitting within the columns, with all partial strings but the last immediately followed by `str_format->str_break_begin` (e.g., `#`) and all partial strings but the first immediately preceded by `str_format->str_break_end` (e.g., `#`). A single space is inserted between consecutive partial strings (e.g., between the two `#`'s), with a break after each of these spaces.

The following function prints a UTF-8 encoded `const char *` string by copying it into the stack, translating UTF-8 encoded characters to `Uchar`'s, and then calling `min::print_unicode` (p164).

```
min::printer min::print_cstring
                        ( min::printer printer,
                          const char * str,
                          const min::str_format * str_format = NULL )
```

The operator `<<` in '`printer << min::pgen ( str )`' calls `min::print_cstring` using

```
str_format = printer->print_format.gen_format->str_format
```

However, `printer->print_format.str_format` is <u>not</u> implicitly used when `min::print_cstring` is called directly with a `NULL` third argument.

The following function may be called to print a string `min::gen` value:

```
min::printer min::print_str
                        ( min::printer printer,
                          min:gen str,
                          const min::str_format * str_format = NULL )
```

This function just calls the `min::print_cstring` function described above with the `const char *` part of the string general value. The `str` argument must be a string general value. The `str_format` parameter is passed directly to `min::print_cstring` even if it is `NULL`.

The `min::standard_pgen` function that prints general values for `min::pgen` will print a string as an identifier of the form `@`<*id*> if the string length is longer than both the `id_strlen` member of the `min::str_format` being used to print the string and the global variable `min::max_id_strlen` (see p71).

5.18.3 Printing Label General Values

The format for printing label `min::gen` values is:

```
struct min::lab_format
    {
      min::pstring        lab_prefix;
      min::pstring        lab_separator;
      min::pstring        lab_postfix;
    };

const min::lab_format min::name_lab_format:
    NULL                                     // lab_prefix
    min::space_if_none_pstring               // lab_separator " "
    NULL                                     // lab_postfix


const min::lab_format min::leading_always_lab_format:
    NULL                                     // lab_prefix
    min::leading_always_pstring              // lab_separator
         // Equivalent to printer << min::leading
    NULL                                     // lab_postfix


const min::lab_format min::trailing_always_lab_format:
    NULL                                     // lab_prefix
    min::trailing_always_pstring             // lab_separator
         // Equivalent to printer << min::trailing
    NULL                                     // lab_postfix


const min::lab_format min::bracket_lab_format:
    min::left_square_angle_space_pstring     // lab_prefix    "[< "
    min::space_if_none_pstring               // lab_separator " "
    min::space_right_angle_square_pstring    // lab_postfix   " >]"
```

A label format is only of use as part of a **`min::gen_format`**, as the latter is needed to print the elements of the label.

Given a label format **`lf`**, a **`min::gen`** label value is printed as **`lf->lab_prefix`**, followed by the label elements separated by **`lf->lab_separator`**'s, followed by **`lf->lab_postfix`**. Note that `NULL` values for **`lf->lab_prefix/separator/postfix`** act like empty strings that print nothing, and a **`NULL`** value for **`lf`** acts like **`min::bracket_lab_format`**.

5.18.4 Printing Special General Values

The format for printing special **`min::gen`** values is:

```
struct min::special_format
    {
      min::pstring                        special_prefix;
      min::pstring                        special_postfix;
      min::packed_vec_ptr<min::ustring>   special_names;
    };
```

```
const min::special_format min::name_special_format:
    NULL                                    // special_prefix
    NULL                                    // special_postfix
    min::standard_special_names             // special_names
```

```
const min::special_format min::bracket_special_format:
    min::left_square_dollar_space_pstring   // special_prefix   "[$ "
    min::space_dollar_right_square_pstring  // special_postfix  " $]"
    min::standard_special_names             // special_names
```

```
min::packed_vec_ptr<min::ustring> min::standard_special_names
      // Names of specials 0x1000000 - 0 .. 0x1000000 - 27 are:
      //     first (does not exist): NULL
      //     next 13 (unused): SPECIAL -1 ... SPECIAL -13
      //     MULTI_VALUED ANY  NONE
      //     MISSING  DISABLED  ENABLED
      //     UNDEFINED  UNUSED  SUCCESS  FAILURE  ERROR
      //     LOGICAL_LINE INDENTED_PARAGRAPH
```

Given a special format **sf**, a **min::gen** special value **v** is printed as its name surrounded by **lf->special_prefix/postfix**. Let

```
i = min::special_index_of ( v )
j = 0x1000000 - i
```

Here **i** is the index of **v** and **0x1000000 - 1** is the maximum possible value of **i**. Then if **sf->special_names** is not **min::NULL_STUB**, and

```
1 <= j < sf->special_names->length
```

then **sf->special_names[j]** is the name of **v**.

Otherwise the name of **v** can be either '**SPECIAL** i' or '**SPECIAL** $-j$', where i and j are represented as decimal numbers. Standardly the former is chosen if $i <$ `0x800000` and the latter if $j \leq$ `0x800000`.

NULL values for **sf->lab_prefix/postfix** act like empty strings that print nothing, and a **NULL** value for **sf** acts like **min::bracket_special_format**.

WARNING: The **special_names** member of a **min::special_format** is not locatable by the ACC, and its value must therefore be stored elsewhere in a locatable variable. It is expected that values of **special_names** will never be garbage collected and will be pointed at by **static min::locatable_var** or **min::locatable_gen** variables.

5.18.5 Printing Other General Values

Printing **min::gen** values that are objects uses the **obj_format** element of a a **min::gen_format**. The **min::flush_id_map** printer operation uses the **id_map_gen_format** element of the printer's **min::print_format** to print a general value, which is most often an object. Printing objects is complex and its discussion is deferred until 5.19.9 *p*229, after the description of object general values.

The other kinds of general values are similar to special values but with a different types and without names such as '**MISSING**'. They are printed in the same way as special values and given names such as **AUX(0x000000004)**. They are also printed using **printer->print_format.gen_format->special_format.special_prefix/postfix** to surround the value name.

5.18.6 Printing Using An Identifier Map

General values that have stubs, namely objects, packed structures and vectors, and long strings, can be output as an identifier of the form **@**<*id*> by using the printer ***identifier map*** (5.16 *p*117).

The function:

```
min::printer min::print_id
                        ( min::printer printer,
                          min::gen v )
```

executes (p117):

```
min::uns32 ID = min::find_or_add ( id_map, v );
printer << "@" << ID;
```

It is used when an ID must be printed to represent an object but the object may not already have gotten an ID.

A printer's **id_map** member is initialized automatically when needed. However, it can also be set by:

```
min::id_map min::set_id_map^S
                        ( <min::printer printer,
                          min::id_map map = min::NULL_STUB )
```

which permits maps to be shared among printers. If no second argument is given, an identifier map is created for the printer if none previously exists. This can then be shared with other printers. The final **id_map** member of the printer is returned, and this is never **min::NULL_STUB**.

When a value with a stub is printed as '**@**<*id*>', it is recorded in **printer->id_map** so it may be printed on separate lines. This may be done by the printer operations:

```
min::op min::flush_one_id
min::op min::flush_id_map
```

The first operation prints the value at **printer->id_map[i]** where **i** equals **printer->id_map->next**, and then increments **printer->id_map->next**. However it does nothing if **printer->id_map->next** is initially beyond the end of **printer->id_map[i]**. The second operation simply repeats the first operation as long as **printer->id_map->next** is less than **printer->id_map->length**. The **min::flush_id_map** operation is also performed by the **min::eom** end-of-message operation if the print format **FLUSH_ID_MAP_ON_EOM** printer operation flag is on.

These **id_map** flushing operations use the **min::gen_format**:

printer->print_format.id_map_gen_format

to print the **printer->id_map[i]** *values* in lines of the form:

@<*id*> **=** *value*

In the following functions, if the **gen_format** argument is not given or **NULL**, it defaults to

printer->print_format.id_map_gen_format

and if the **id_map** argument is not given or **min::NULL_STUB**, it defaults to **printer->id_map**. Also in all these functions the '**printer**' argument is returned as the function value.

The functions

```
min::printer min::print_one_id
                        ( min::printer printer,
                          min::id_map id_map = min::NULL_STUB,
                          const min::gen_format * gen_format = NULL )
min::printer min::print_id_map
                        ( min::printer printer,
                          min::id_map id_map = min::NULL_STUB,
                          const min::gen_format * gen_format = NULL )
```

are equivalent to **printer << min::flush_one_id** and **printer << min::flush_id_map** respectively, except an alternative **id_map** and **gen_format** can be given.

The function

```
min::printer min::print_mapped_id
                        ( min::printer printer,
                          min::uns32 ID,
                          min::id_map id_map = min::NULL_STUB,
                          const min::gen_format * gen_format = NULL )
```

prints **id_map[ID]** in a line of the form:

@<*ID*> **=** *value*

The function

```
min::printer min::print_mapped
                        ( min::printer printer,
                          min::gen v,
                          min::id_map id_map = min::NULL_STUB,
                          const min::gen_format * gen_format = NULL )
```

executes

```
min::uns32 ID = min::find_or_add ( id_map, v );
if ( ID < id_map->next )
    min::print_mapped_id
        ( printer, ID, id_map, gen_format );
min::print_id_map ( printer, id_map, gen_format );
```

5.18.7 Printing Using Defined Formats

Packed structures, packed vectors, and objects with **.type** attribute values can be printed using user defined formats. A ***defined format*** is a closure consisting of a C++ function and arguments. The defined format is specifically a packed vector whose header contains a pointer to its function and whose elements are the closure arguments to that function:

```
typedef min::packed_vec_ptr<min::gen,min::defined_format_header>
          min: defined_format
typedef min::packed_vec_insptr<min::gen,min::defined_format_header>
          min: defined_format_insptr

typedef min::printer ( * min:: defined_format_function )
       ( min::printer printer,
         min::gen v,
         const min::gen_format * gen_format,
         min::defined_format defined_format )
```

```
struct min::defined_format_header
{
                         const min::uns32   control
                         const min::uns32   length
                         const min::uns32   max_length
     const min::defined_format_function   defined_format_function
}
```

A defined format can be created by the function:

```
min::defined_format_insptr min:: new_defined_format R
       ( min::defined_format_function f,
         min::uns32_number of_arguments )
```

There is a ***defined format packed map*** of packed structure/vector subtypes (p94) to defined formats that can be edited by the function:

```
void min::map_packed_subtype R
               ( min::uns32 subtype,
                 min::defined_format defined_format )
```

If the **defined_format** argument is **min::NULL_STUB** the subtype is in effect unmapped.

When **min::standard_pgen** (p161) is about to print a packed structure or vector and its **disable_mapping** argument is **false**, it looks up the subtype in the defined format packed map, and if the value found is a defined format and not **min::NULL_STUB**, the defined format function is called to do the printing. This call passes the value **v** to be printed (which contains a pointer to the packed structure or vector), the **min::gen_format** that **min::standard_pgen** was using print **v**, and the defined format found in the map.

Similarly there is a ***defined format type map*** of **.type** attribute values to defined formats that can be edited by the function:

```
void min::map_type R
               ( min::gen type,
                 min::defined_format defined_format )
```

If the **defined_format** argument is **min::NULL_STUB** the **type** is in effect unmapped.

When **min::standard_pgen** (p161) is about to print an object with a **.type** value and its **disable_mapping** argument is **false**, it looks up the **.type** value in the defined format type map, and if the value found is a defined format and not **min::NULL_STUB**, the defined format function is called to do the printing. This call passes the value **v** to be printed (which points at the object), the **min::gen_format** that **min::standard_pgen** was using to print **v**, and the defined format found in the map. See p235 for more details.

When a defined format function is called it must act as a substitute for **min::standard_pgen**. If it wishes to decline this opportunity, it may call

```
min::print_gen ( printer, v, gen_format, true )
```

with the **`disable_mapping`** argument set to **`true`** to suppress defined format mapping for **`v`** (but not for the components of **`v`**).

5.19 Objects

An ***object*** is conceptually a hash table that maps ***attribute names*** to ***attribute values***. An attribute name is a sequence of name components, which are numbers, strings, and labels. The part of the hash table that maps attribute names that are equal to or begin with small unsigned integers is actually a vector. An attribute value is a `min::gen` datum.

Each attribute name-value pair in an object map represents an ***arrow*** in the data base. The ***source*** of the arrow is the object which is the map, the ***label*** of the arrow is the attribute name, and the ***destination*** of the arrow is the attribute value. Some arrows can be ***double arrows*** pointing both to and from the arrow destination. If the arrow is a double arrow, a reverse direction label, called the ***reverse attribute name*** is attached to the arrow, and the object at the destination end of the arrow is the source of the reversed arrow, whose label is the reverse attribute name and whose destination is the unreversed arrow's source.

There can be several arrows with the same attribute name, and even several arrows with both the same attribute name and the same value. There can be several double arrows with the same attribute name and the same reverse attribute name, and even several double arrows with the same names and destination object.

Therefore an attribute name and reverse attribute name together name a multi-set of values, where the reverse attribute name can be missing to indicate that only single arrows are to be considered, and is present to indicate that only double arrows are to be considered. For single arrows the values are any `min::gen` values, but for double arrows the values are always other objects. It is possible to add a value to one of these multi-sets or delete a value from a multi-set. It is possible to delete an entire multi-set. It is possible to treat a multi-set as a set, by adding a value to it only if the value does not already occur in the multi-set. When testing for value equality, **`==`** is normally used, as this tests for equality of both name components and objects.

Flags, called ***attribute flags***, can be attached to an attribute name. Note that flags are attached to object attribute names, and not to arrows, values, or reverse attribute names.

There are also graph typed objects, each of which is a pair of objects, one called the graph type that is constant, is shared among many graph typed objects, and contains attribute labels and constant attribute values, and one called the context, which contains a vector of ***variables*** that hold variable attribute values. Graph typed objects are described in more detail in 5.19.8 ^{p224}.

5.19.1 Object Bodies

An object has a body that consists of the following 6 parts in the order given:

header
variable vector
hash table
attribute vector
unused area
auxiliary area

The ***header*** contains object flags (see Object Flags, p296, for a complete list of object flags) and the sizes of the other 5 parts. The ***variable vector*** stores the object's variables. The ***hash table*** stores attribute name/value pairs, for attributes whose names do not begin with small unsigned integers. The ***attribute vector*** stores attribute values for attributes whose names begin with small unsigned integers. The ***auxiliary area*** stores elements of lists headed by hash table and attribute vector elements, and any other data that would overflow a single **`min::gen`** value. The ***unused area*** provides for growth of the attribute vector and auxiliary area.

The variable vector and hash table are of fixed size; their size can only be changed by resizing, reorganizing, and often relocating the object body. The attribute vector grows up from the end of the hash table into the unused area, and the auxiliary storage grows down from the end of the body into the unused area.

A variable vector, hash table, and attribute vector element is accessed using an index relative to the beginning of the vector or table that contains the element. Auxiliary area elements are accessed by ***auxiliary pointers*** that give the index of the element relative to the end of the object body. Auxiliary pointers with zero index do not address a body vector element. For more details see the **`var/hash/attr/aux`** functions on p182, and their equivalents on p188.

An object may be grown or compacted by relocating and reorganizing its body. It may be grown to expand its unused area or hash table, or, less commonly, its variable vector. An object may be compacted to eliminate, or less commonly to shrink, its unused area, and possibly to shrink its hash table.

There are four kinds of objects: tiny, short, long, and huge. These have respectively 4, 8, 16, and 32 byte headers, and are capable of storing information for successively larger objects. There is no essential difference between these headers and their internal structure is not visible. There is no practical limit on the size of an object[8].

An object body may be ***relocated***, which means the body is simply copied to a new address, or ***reorganized***, which means the object body is reformatted, and possibly converted from

[8]Currently the maximum virtual address space size is 2^{48} bytes.

one header size to another. Frequently an object whose attribute labels and number of values for each label are no longer subject to change will be compacted, to make it as small as possible, and this is one kind of object body reorganization. Objects are compacted by the '**min::publish**^{O}' function that sets the object **OBJ_PUBLIC** flag which prevents changes to the attribute labels, attribute label flags, and number of attribute values of an object. Reorganization can also occur whenever an object is relocated by the compactor, provided that the **OBJ_PRIVATE** and **OBJ_PUBLIC** object flags are not set.

The data structure of an object body can be viewed at any of three levels: vector level (5.19.5 ^{p180}), list level (5.19.6 ^{p189}), and attribute level (5.19.7 ^{p204}). The interfaces to all of these levels is protected. In addition there is an unprotected vector level interface (5.19.5.3 ^{p187}) and a graph typed object interface (5.19.8 ^{p224}).

5.19.2 Object Creation

An object can be created by the protected function:

```
min::gen min::new_obj_gen R
                    ( min::unsptr unused_size,
                      min::unsptr hash_size = 0,
                      min::unsptr var_size = 0,
                      bool expand = true )
```

A preallocated stub (5.6.4 ^{p41}) may be filled by a new object using the protected function:

```
min::gen min::new_obj_gen R
                    ( min::gen preallocated,
                      min::unsptr unused_size,
                      min::unsptr hash_size = 0,
                      min::unsptr var_size = 0,
                      bool expand = true )
```

where the **preallocated** argument designates the preallocated stub.

The new object has a zero length attribute vector, but this can be expanded by growing the attribute vector upward into the unused area. Similarly the auxiliary area is zero length, but can be expanded by growing downward into the unused area.

The sizes of parts of an object may be changed after the object has been created by the object maintenance functions described in Section 5.19.3 ^{p176}.

Allocators often maintain sets of free memory blocks of particular sizes, often powers of two, and allocate new object bodies to one of these blocks. In such a case the block may be larger than the requested body size. If the '**expand**' argument is **true**, **min::new_obj_gen** will round the unused area size of the new object up so that the new object body fills the free block to which it is allocated.

On the flip side, the compactor part of the allocator/collector/compactor may reduce the size of the unused area. For an object that has been around a long enough time, the compactor may eliminate the unused area completely, and may reorganize the object to downsize its hash table and compact its auxiliary area.

The hash table of the returned object is filled with **min::LIST_END()** values (p189), and is therefore an empty hash table. The variable vector is filled with **min::UNDEFINED()** values. Unused area elements are initialized to some implementation defined value like **0**. The attribute vector and auxiliary areas of the returned object are zero length, and all space not used by the header, hash table, and variable vector is allocated to the unused area. The attribute vector and auxiliary area can be filled by 'push' instructions described below (p185).

A **min::gen** value may be tested to see if it points at an object by the function:

```
bool min::is_obj ( min::gen v )
```

The **min::type_of** function (p25) applied to an object returns one of the following:

```
min::TINY_OBJ
min::SHORT_OBJ
min::LONG_OBJ
min::HUGE_OBJ
```

5.19.3 Object Maintenance Functions

Object maintenance functions are used to change the sizes of object components and compact objects. The most basic of these functions do nothing but change the size of the unused area and variable area:

```
void min::resize^S
            ( min::gen object,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
void min::resize^S
            ( min::gen object,
              min::unsptr unused_size )
void min::expand^S
            ( min::gen object,
              min::unsptr unused_size )
```

The 4-argument **min::resize** function can change the size of both the variables vector and unused area at the same time. The object body is reallocated, and the function's '**expand**' argument, if '**true**', will increase the size of the requested unused area until the object fills the actual allocated block. This is useful because allocators typically allocate blocks whose size is a power of two or a bit less, so if the requested size is not just right, some memory will be wasted.

The 2-argument **`min::resize`** function and the **`min::expand`** function are both equivalent to **`min::resize`** with no change in variables vector size. The 2-argument **`min::resize`** has implied **`false`** '**`expand`**' argument, and the **`min::expand`** function has implied **`true`** '**`expand`**' argument.

The **`min::expand`** function is the one implicitly used when an object simply needs more memory for its attribute vector or auxiliary area to grow into. In this case the argument is just the memory required, and extra unused area is allocated to fill the next real allocated size block. Repeated uses of **`min::expand`** on the same object will more or less keep doubling the actual size of the object.

The **`min::resize`** and **`min::expand`** functions with object arguments are called '***resizing functions***' because they change the size of the object. They also change the offset of some elements within the object, and usually relocate the object. The names of such functions are marked with the subscript $^{\mathbf{S}}$. A resizing function is also a relocating function (p12).

The remaining maintenance functions assume objects are organized as per Section 5.19.6 ^{p189}, Object List Level. These functions all ***reorganize*** an object so it has no discontinuities in its lists, and therefore no unnecessary invisible jumps (auxiliary list pointers, actually) within its lists, and makes no use of auxiliary stubs (5.19.6.2 ^{p201}). So in this sense these functions clean up and compactify the object. Object reorganization involves changing auxiliary pointers in the object and changing the locations of list elements in the object auxiliary area. It also involves resizing the object.

A function that changes the size of an object hash table further assumes that hash table elements are *attribute/node-name-descriptor-pairs* as per Section 5.19.7 ^{p204}, Object Attribute Level. More specifically, it is assumed just that hash table entries are lists whose elements are grouped into pairs and the first element of each pair is a hashable **`min::gen`** value such that if the hash of this element is H, then the pair is put into the list of the $H\%S+1$'st hash table element, where S is the size of the hash table.

With all this in mind, the following function can be used to reorganize and object and change the sizes of its variable vector, hash table, and unused area.

```
void min::reorganize^O
            ( min::gen object,
              min::unsptr hash_size,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
```

The '**`expand`**' argument to this function works to increase the size of the unused area in the same was as the the '**`expand`**' argument to the **`min::resize`** function above. If the hash table size is not actually changed, the Object Attribute Level assumptions made above do not apply, but the Object List Level assumptions do apply to all object reorganizations.

After all an object's attributes have been set, the number of *attribute/node-name-descriptor-pairs* in the object hash table will cease to change, and the hash table size an be optimized to about twice this number. The following functions use this idea:

```
void min::compact^O
            ( min::gen object,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
void min::compact^O ( min::gen object )

void min::publish^O ( min::gen object )
```

These functions are like the **min::reorganize** function but they pick the size of the hash table automatically as follows. If N is the number of *attribute/node-name-descriptor-pairs* in the hash table and S is the size of the hash table, the hash table size is left unchanged if $2N/3 \le S \le 3N$. Otherwise the hash table size is reset to $2N$.

The 1-argument **min::compact** function does not change the size of the variables vector and forces the unused area size to be zero. Its goal is to compact the object under the assumption the object will not change any of its sizes in the future.

The **min::publish** function just executes a 1-argument min::compact function and then sets the object **OBJ_PUBLIC** flag which prevents further changes to the to the object sizes or organization (but not to attribute values).

The **min::reorganize**, **min::compact**, and **min::publish** functions with object arguments are called '***reorganizing functions***', because they move elements around inside the object. The names of such functions are marked with the subscript $^{\mathbf{O}}$. A reorganizing function is also a resizing function (p177) and a relocating function (p12).

It is sometimes useful to discover the values of an object's **OBJ_PRIVATE** and **OBJ_PUBLIC** flags. This can be done by the functions:

```
bool min::private_flag_of ( min::gen object )

bool min::public_flag_of ( min::gen object )
```

All of the above maintenance functions except the last two flag reading functions work by creating vector pointers and using versions of the functions in which the **min::gen** object argument is replaced by a vector pointer argument. The functions that actually change the object use a **min::obj_vec_insptr** insertable object vector pointer and therefore will not work if the object's **min::OBJ_PUBLIC** flag as already been set.

If you want to build your own compactification function implementing a different algorithm for choosing hash table size, you may use the vector pointer version of **min::reorganize** (p186) and the following:

```
min::unsptr min::hash_count_of ( min::obj_vec_ptr & vp )

        void min::set_public_flag_of ( min::obj_vec_insptr & vp )
```

The **min::hash_count_of** function just returns the number of *attribute/node-name-descriptor-pairs* in the object's hash table, and the **min::set_publish_flag** function just sets the object's **min::OBJ_PUBLIC** flag.

5.19.4 Object Copy Functions

The following functions are like the **min::resize** functions on p176 except that instead of resizing the object specified by their argument, they make a new resized object which they return:

```
min::gen min::copy^R
                    ( min::gen object,
                      min::unsptr var_size,
                      min::unsptr unused_size,
                      bool expand = true )
min::gen min::copy^R
                    ( min::gen object,
                      min::unsptr unused_size )
```

The new object copy may also be used to fill a preallocated stub (5.6.4 ^{p41}) by using the function:

```
min::gen min::copy^R
                    ( min::gen preallocated,
                      min::gen object,
                      min::unsptr unused_size )
```

where the **preallocated** argument designates the preallocated stub.

The new object is <u>not</u> public even if the original object is. The forms of **min::copy** with no **expand** argument always expand the unused area to fit the space actually allocated to the new object, as if they had a **true expand** argument.

Alternative forms of these functions take object vector pointers (see 5.19.5.1 ^{p181}):

```
min::gen min::copy R
                    ( min::obj_vec_ptr & vp,
                      min::unsptr var_size,
                      min::unsptr unused_size,
                      bool expand = true )
min::gen min::copy R
                    ( min::obj_vec_ptr & vp,
                      min::unsptr unused_size )
min::gen min::copy R
                    ( min::gen preallocated,
                      min::obj_vec_ptr & vp,
                      min::unsptr unused_size )
```

5.19.5 Object Vector Level

At the ***vector level***, the object body is viewed as a ***body vector*** of **min::gen** values. The entire body vector, except for the header, consists of **min::gen** elements of the body vector. A ***body vector index*** is an index of an element in this vector, and ranges from 0 to one less than the total size of the object, which is the size of the body vector in **min::gen** units. The index 0 cannot be used, as it corresponds to the header, which does not contain **min::gen** values. 0 can be used as a form of null pointer.

An auxiliary pointer index **i** corresponds to the body vector index total size **- i**, so **i=1** corresponds to the last element of the auxiliary area. The 0 auxiliary pointer index corresponds to the location just beyond the body vector and is not usable as the index of an auxiliary area element, so it can also be used as a form of null pointer. By indexing auxiliary area elements in this way, the unused area size can be changed without changing any auxiliary pointers.

There are three kinds of vector level protected object pointers. A read-only **min::obj_vec_ptr** pointer permits read-only access to body vector elements. A read-write or updatable **min::obj_vec_updptr** pointer permits read-write access to body vector elements, but does not permit pushing or popping elements from the object's attribute vector or auxiliary area. An insertable **min::obj_vec_insptr** pointer does permit these pushes and pops, in addition to permitting read-write access to body vector elements.

An object has two flags that regulate the creation of vector level pointers: **min::OBJ_PRIVATE** and **min::OBJ_PUBLIC**. If an object has neither of these flags, an object vector pointer to the object may be created, and this will set the **min::OBJ_PRIVATE** flag. When the object vector pointer is destructed, this flag will be cleared. While this flag is set, no other vector pointer to the object may be created. The object is therefore private to the code possessing the vector pointer.

If an object has neither flag, its **min::OBJ_PUBLIC** flag may be set by calling the **min::**

publish^O function (p178). When this is set, any number of read-only and read-write (updatable) vector pointers to the object may be created, but no insertable vector pointers to the object may be created. Constructing and destructing pointers in this case does not set or clear object flags. The **min::OBJ_PUBLIC** flag may not be cleared by protected functions. The idea here is that setting the **min::OBJ_PUBLIC** flag fixes the attribute label structure of the object, the flags attached to attribute labels, and the size of the value multiset of each attribute, but permits the existing values of an attribute to be read or written. This permits read-only sharing of the object and read-write sharing of attribute values of the object, but does not permit attribute values to be added to or deleted from the object.

An object with its **min::OBJ_PUBLIC** flag set is said to be a ***public*** object.

Note that object bodies can be relocated whenever a relocating function (p12) is called. However object bodies may be reorganized by a relocating function only if neither the **min::OBJ_PRIVATE** nor the **min::OBJ_PUBLIC** flags are set. Object bodies may also be reorganization when insertions are made using the **min::insert_reserve**^{S} function (p197) on an insertable pointer, and object bodies are usually reorganized when the **min::OBJ_PUBLIC** flag is set on an object by the **min::publish**^{O} function (p178).

5.19.5.1 Protected Object Vector Pointers. An ***object vector pointer*** points at the stub of an object and caches information derived from the header of the object, so that the differences between different sized object headers are hidden. A non-public object can have at most one object vector pointer pointing at it. A public object can have any number of read-only or read-write (update) object vector pointers pointing at it, but may not change the sizes of its various parts (like the attribute vector or auxiliary area), so that the information cached in the object vector pointers never changes.

A **min::obj_vec_ptr** read-only object vector pointer may be created and set by the following functions:

```
(constructor) min::obj_vec_ptr vp ( min::gen v )
(constructor) min::obj_vec_ptr vp ( const min::stub * s )
(constructor) min::obj_vec_ptr vp ( void )

                   operator const min::stub *
                       ( min::obj_vec_ptr & vp )
                   operator bool
                       ( min::obj_vec_ptr & vp )
 min::obj_vec_ptr & operator =
                       ( min::obj_vec_ptr & vp,
                         min::gen v )
 min::obj_vec_ptr & operator =
                       ( min::obj_vec_ptr & vp,
                         const min::stub * s )
```

Here if **v** or **s** do not point at the stub of an object the new object vector pointer is set to **min::NULL_STUB**, which gives undefined results, but usually a memory fault, if used to access the parts of a object. The case where **s** equals **min::NULL_STUB** also sets the object vector pointer to **min::NULL_STUB**, as does the no-argument object vector pointer constructor. An **=** operator sets an object vector pointer just as if the pointer had just been constructed by a constructor whose argument was the right-side argument of the **=** operator.

Because an object vector pointer can be converted to a **const min::stub *** value, it is possible to check whether an object vector pointer equals **min::NULL_STUB** by using the **==** function. It is also possible to set one vector pointer from another, as the latter will be converted to a **const min::stub *** value, but note that the object must be public in order for more than one object vector pointer to point at it.

The conversion of a vector pointer to a **bool** returns **true** iff the vector pointer stub is not **min::NULL_STUB**.

Unlike label, packed structure, or packed vector pointers, object vector pointers cache information about the object, and must always be writable, and never **const**.

The following protected functions may be used to discover information about the object pointed at by a read-only **min::obj_vec_ptr**:

```
min::unsptr min::var_size_of ( min::obj_vec_ptr & vp )
min::unsptr min::hash_size_of ( min::obj_vec_ptr & vp )
min::unsptr min::attr_size_of ( min::obj_vec_ptr & vp )
min::unsptr min::unused_size_of ( min::obj_vec_ptr & vp )
min::unsptr min::aux_size_of ( min::obj_vec_ptr & vp )
min::unsptr min::total_size_of ( min::obj_vec_ptr & vp )
```

Here the sizes are in **min::gen** units. When an object body is reorganized, some of these sizes may change, but simply relocating the object body does not change these sizes. The automatic resizing of the object that may occur when new elements are pushed into the attribute vector or auxiliary area only changes the unused and total sizes.

The following functions can be used for read-only access to object elements:

```
const min::gen & var ( min::obj_vec_ptr & vp, min::unsptr index )
const min::gen & hash ( min::obj_vec_ptr & vp, min::unsptr index )
const min::gen & attr ( min::obj_vec_ptr & vp, min::unsptr index )
const min::gen & aux ( min::obj_vec_ptr & vp, min::unsptr aux_ptr )
```

Given a vector pointer **vp**, **min::var(vp,i)** can be used to read the **i+1**'st variable in the object pointed at by **vp**, for $0 \le i <$ **min::var_size_of(vp)**. Similarly, **min::hash(vp,i)** can be used to read the **i+1**'st hash table entry for $0 \le i <$ **min::hash_size_of(vp)**, **min::attr(vp,i)** can be used to read the **i+1**'st attribute vector entry for $0 \le i <$ **min::attr_size_of(vp)**. **min::aux(vp,p)** differs slightly in that it reads the **i**'th auxiliary area element ordering the elements from the end of the auxiliary area to its beginning, for $1 \le i \le$ **min::attr_size_of(vp)**. Auxiliary area indices are defined in this manner so that an object body may be resized by simply expanding or contracting its unused area, without modifying the contents of the other parts of the object body.

The operations

```
const min::gen & operator [ ]                [same as min::attr]
                 ( min::obj_vec_ptr const & vp,
                   min::unsptr index )
min::unsptr min::size_of                      [same as min::attr_size_of]
                 ( min::obj_vec_ptr & vp )
```

are can be used as alternatives that make the attribute vector part of an object look like a simple vector.

Note that addresses such as **&min::var(vp,i)** and **&vp[i]** are relocatable addresses, and as such should only be passed to a non-relocating function like **memcpy**, and never saved in a local variable. Both **vp[i]** and **min::attr(vp,i)** include a **MIN_ASSERT** check that $0 \leq i <$ **min::attr_size_of(vp)**.

Also in this vein, the operations

```
min::ptr<const min::gen> operator +
                             ( min::obj_vec_ptr & vp,
                               min::unsptr index )
min::ptr<const min::gen> min::begin_ptr_of ( min::obj_vec_ptr & vp )
min::ptr<const min::gen> min::end_ptr_of ( min::obj_vec_ptr & vp )
```

can be used to return **min::ptr<const min::gen>** pointers to elements of the attribute vector. **vp + i** returns a pointer to the **i**+1'st element, and includes a **MIN_ASSERT** check that **i** is less than the size of the attribute vector. The **min::begin_ptr_of** and **min::end_ptr_of** functions are the same as **vp + 0** and **vp + min::size_of(vp)** but do not include this **MIN_ASSERT** check, so that, for example, **min::begin_ptr_of** can be used when the attribute vector has zero length.

A **min::obj_vec_updptr** read-write (update) object vector pointer is like a **min::obj_vec_ptr** read-only object vector pointer, but has element access functions that have been modified to permit writing elements. Its functions are:

```
(constructor) min::obj_vec_updptr vp ( min::gen v )
(constructor) min::obj_vec_updptr vp ( const min::stub * s )
(constructor) min::obj_vec_updptr vp ( void )
```

```
                        operator const min::stub *
                            ( min::obj_vec_updptr const & vp )
    min::obj_vec_updptr & operator =
                            ( min::obj_vec_updptr & vp,
                              min::gen v )
    min::obj_vec_updptr & operator =
                            ( min::obj_vec_updptr & vp,
                              const min::stub * s )

min::ref<min::gen> var
                        ( min::obj_vec_updptr & vp,
                          min::unsptr index )
min::ref<min::gen> hash
                        ( min::obj_vec_updptr & vp,
                          min::unsptr index )
min::ref<min::gen> attr
                        ( min::obj_vec_updptr & vp,
                          min::unsptr index )
min::ref<min::gen> aux
                        ( min::obj_vec_updptr & vp,
                          min::unsptr index )

    min::ref<min::gen> operator [ ]
                        ( min::obj_vec_updptr const & vp,
                          min::unsptr index )

        min::ptr<min::gen> operator +
                            ( min::obj_vec_updptr const & vp,
                              min::unsptr index )
    min::ptr<min::gen> min::begin_ptr_of ( min::obj_vec_updptr & vp )
    min::ptr<min::gen> min::end_ptr_of ( min::obj_vec_updptr & vp )
```

Updatable pointer read-write element access functions return either **min::ref<min::gen>** or **min::ptr<min::gen>** values while read-only pointer element access functions return '**const min::gen &**' or **min::ptr<const min::gen>** values.

A **min::obj_vec_updptr** read-write pointer may be automatically downcast to a **min::obj_vec_ptr** read-only pointer. This means that the non-constructor functions not redefined above that are applicable to **min::obj_vec_ptr**'s are applicable to **min::obj_vec_updptr**'s.

A **min::obj_vec_insptr** insertable object vector pointer is like a **min::obj_vec_updptr** read-write object vector pointer but has additional functions which support pushing or popping elements from the end of the object body attribute vector or the beginning of the object body auxiliary area and resizing the unused area and variable vector portions of the object body. A **min::obj_vec_insptr** point may not point at a public object (p178). The additional functions for this type of pointer are:

```
(constructor) min::obj_vec_insptr vp
                    ( min::gen v )
(constructor) min::obj_vec_insptr vp
                    ( const min::stub * s )
(constructor) min::obj_vec_insptr vp
                    ( void )

                    operator const min::stub *
                        ( min::obj_vec_insptr const & vp )
 min::obj_vec_insptr & operator =
                        ( min::obj_vec_insptr & vp,
                          min::gen v )
 min::obj_vec_insptr & operator =
                        ( min::obj_vec_insptr & vp,
                          const min::stub * s )

void min::ref<min::gen> min::attr_push ( min::obj_vec_insptr & vp )
void min::ref<min::gen> min::aux_push ( min::obj_vec_insptr & vp )

void min::attr_push
              ( min::obj_vec_insptr & vp,
                min::unsptr n, const min::gen * p = NULL )
void min::aux_push
              ( min::obj_vec_insptr & vp,
                min::unsptr n, const min::gen * p = NULL )

min::gen min::attr_pop
                  ( min::obj_vec_insptr & vp )
min::gen min::aux_pop
                  ( min::obj_vec_insptr & vp )

void min::attr_pop
              ( min::obj_vec_insptr & vp,
                min::unsptr n, min::gen * p = NULL )
void min::aux_pop
              ( min::obj_vec_insptr & vp,
                min::unsptr n, min::gen * p = NULL )
```

The push functions push values to the end of the attribute vector and the beginning of the unused area. The pop functions pop values from the end of the attribute vector and the beginning of the unused area.

There are two versions of each push or pop: one that returns either a **min::ref<min::gen>** reference to a single **min::gen** value, and one that handles **n** values stored in a C/C++ vector pointed at by **p**. These latter push and pop functions preserve the order of the values in memory, so **p[0]** is pushed into or popped from a body vector location immediately before the location into which **p[1]** is pushed into or popped from, etc. Thus the attribute vector push function pushes in the order **p[0]**, **p[1]**, **p[2]**, ...; while the auxiliary vector push function pushes in the order **p[n-1]**, **p[n-2]**, **p[n-3]**, ... The order of the elements of **p** in memory is the same as the order of the elements in the object body.

If the push functions are called with argument **p = NULL**, they set the elements pushed to zero. If the pop functions are called with argument **p = NULL**, they discard the values of the popped elements.

If the push functions are called when the unused area size is smaller by k elements than the number of **min::gen** values being pushed, these functions call the **min::expand** function (see below) to add k elements to the unused area before the push is executed.

A **min::obj_vec_insptr** may be automatically downcast to either a **min::obj_vec_updptr** or a **min::obj_vec_ptr**. This means that all the non-constructor functions applicable to **min::obj_vec_updptr**'s or **min::obj_vec_ptr**'s are also applicable to **min::obj_vec_insptr**'s.

5.19.5.2 Vector Level Object Maintenance. The object maintenance functions of Section 5.19.3 ^{p176} which operate on **min::gen** values have counterparts that operate on object vector pointers. These last are merely listed here, and are as described in the section just mentioned, with the exception that the two functions that set the **min::PUBLIC** flag, **min::set_public_flag_of** and **min::publish**, end by setting the insertable object vector pointer they use to **min::NULL_STUB**, as an insertable object vector pointer is not allowed to point at a public object.

```
void min::resize S
            ( min::obj_vec_insptr & vp,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
void min::resize S
            ( min::obj_vec_insptr & vp,
              min::unsptr unused_size )
void min::expand S
            ( min::obj_vec_insptr & vp,
              min::unsptr unused_size )

void min::reorganize O
            ( min::obj_vec_insptr & vp,
              min::unsptr hash_size,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
```

```
void min::compact^O
            ( min::obj_vec_insptr & vp,
              min::unsptr var_size,
              min::unsptr unused_size,
              bool expand = true )
void min::compact^O
            ( min::obj_vec_insptr & vp )
void min::publish^O
            ( min::obj_vec_insptr & vp )

bool min::private_flag_of
            ( min::obj_vec_ptr & vp )
bool min::public_flag_of
            ( min::obj_vec_ptr & vp )

min::unsptr min::hash_count_of
                  ( min::obj_vec_ptr & vp )
      void min::set_public_flag_of
                  ( min::obj_vec_insptr & vp )
```

5.19.5.3 Unprotected Object Vector Level. Using unprotected functions, an object body can be treated as a simple C/C++ vector of **min::gen** values. The required functions are:

```
const min::gen * & MUP::base ( min::obj_vec_ptr & v )
      min::gen * & MUP::base ( min::obj_vec_updptr & v )

min::stub * MUP::stub_of ( min::obj_vec_ptr & vp )

min::unsptr MUP::var_offset_of ( min::obj_vec_ptr & vp )
min::unsptr MUP::attr_offset_of ( min::obj_vec_ptr & vp )
min::unsptr MUP::unused_offset_of ( min::obj_vec_ptr & vp )
min::unsptr MUP::aux_offset_of ( min::obj_vec_ptr & vp )

min::unsptr & MUP::unused_offset_of
                       ( min::obj_vec_insptr & vp )
min::unsptr & MUP::aux_offset_of
                       ( min::obj_vec_insptr & vp )

void MUP::acc_write_update
            ( min::obj_vec_ptr & vp,
              bool interrupts_allowed = true )
```

The offset values returned by the above are those of the variable vector(**var**), hash table(**hash**), attribute vector(**attr**), unused area (**unused**), and auxiliary area(**aux**). These offsets are in **min::gen** units, and these offsets and the sizes returned by the functions on page p182 are related by

variable vector offset	=	header size
hash table offset	=	variable vector offset + variable vector size
attribute vector offset	=	hash table offset + hash table size
unused area offset	=	attribute vector offset + attribute vector size
auxiliary area offset	=	unused area offset + unused area size
total size	=	auxiliary area offset + auxiliary area size

Note that when an object is reorganized all these values may change, including the header size.

Note that the **min::unused_offset_of** and **min::aux_offset_of** functions for a **min::obj_vec_insptr** return an lvalue. For insertable pointers these offsets change when values are pushed into or popped from the attribute vector or auxiliary area. Using these unprotected functions it is possible to write your own push and pop functions. The offset lvalues actually reference cache locations in the **min::obj_vec_insptr** which are copied back into the object when the **min::obj_vec_insptr** is destructed.

The following are equivalences except for the omission of **MIN_ASSERT** checks that the index **i** is not too large or too small (auxiliary area indices may not be 0), calls to **min::expand** when the unused area is too small for a push, **MIN_ASSERT** checks that elements to be popped actually exist, conversions to **min::ref<min::gen>** values for appropriate element access functions, and calls to **MUP::acc_write_update** for push operations (p43):

```
       var ( vp, i )  ≡  return base(vp)[var_offset_of(vp) + i]
      hash ( vp, i )  ≡  return base(vp)[hash_offset_of(vp) + i]
      attr ( vp, i )  ≡  return base(vp)[attr_offset_of(vp) + i]
       aux ( vp, i )  ≡  return base(vp)[total_size_of(vp) - i]

attr_push ( vp ) = v  ≡  base(vp)[unused_offset_of(vp)++] = v
 v = attr_pop ( vp )  ≡  v = base(vp)[--unused_offset_of(vp)]
 aux_push ( vp ) = v  ≡  base(vp)[--aux_offset_of(vp)] = v
  v = aux_pop ( vp )  ≡  v = base(vp)[aux_offset_of(vp)++]
```

The header is the only part of an object body that is <u>not</u> **min::gen** values. Even so it is sized in **min::gen** units; e.g., a short object header size is **2** for a compact implementation and **1** for a loose implementation (p14).

The header may contain implementation dependent information used for optimization. For example, objects which have auxiliary stubs (p201) may be flagged, thereby identifying objects that need extra work when deallocated or reorganized.

The **MUP::acc_write_update** function applied to the **min::obj_vec_ptr vp** calls

```
min::acc_write_update ( s1, s2 )
```

(see 5.7.1 p43) with **s1** being the stub of **vp** and **s2** iterating over all the collectable stubs pointed at by the contents of the object that **vp** points at. If **interrupts_allowed** is **true**, **min::interrupt()** is called after each of these calls; otherwise **min::interrupt()** is not called. See Step 5 of the algorithm in 5.6.4.1 p42 for an application of this function.

5.19.6 Object List Level

At the ***list level***, the body consists of two vectors whose elements are lists. The two vectors are the hash table, and the attribute vector (the variable vector is ignored). The elements of the lists are **min::gen** values other than the list or sublist auxiliary pointers, and sublists. The lists are constructed with the help of the list and sublist auxiliary pointers:

list auxiliary pointer The list is continued at the target of the list auxiliary pointer.

sublist auxiliary pointer A sublist starts at the target of the sublist auxiliary pointer.

Lists also make use of two constant values:

`const min::gen min::LIST_END()`
The list ends here. This actually equals a list auxiliary pointer with zero index.

`const min::gen min::EMPTY_SUBLIST()`
A list element value that represents an empty sublist. This actually equals a sublist auxiliary pointer with zero index.

Each hash table or attribute vector element is a ***list head***. Each list head is a ***list continuation***, which is an element of the body vector that has a particular interpretation, described below. However, not all list continuations are list heads.

A list continuation represents a final segment of a list. If it has the value **min::LIST_END()**, the final segment is empty. If the list continuation has a list auxiliary pointer value, that pointer points at another list continuation that continues the list. Otherwise the list continuation represents an element of the list, and is called a ***list element***.

A list element is an element of the list, and cannot be a list auxiliary pointer or the special value **min::LIST_END()**. But it can be a sublist auxiliary pointer, or the special value **min::EMPTY_SUBLIST()**, both of which denote a list element that is a sublist.

Given a list element, the rest of the list after the element begins with a list continuation that has an index one less than that of the list element within the object body vector, unless the list element is a list head, in which case there is no next element, and the list has only one element. All list heads lie in the hash table or attribute vector, so testing whether a list element is a list head can be done if just the index of the list element in the body vector is known. As a consequence of all this, zero and one element lists with heads in the hash table or attribute vector are represented by either **min::LIST_END()** or by the single list element, stored in the hash table or attribute vector entry, without any part of the list being stored in the auxiliary area.

A list element that is a sublist auxiliary pointer or the value **min::EMPTY_SUBLIST()** represents a sublist, and is the ***sublist head*** of that sublist. A sublist head is a list element of the list containing a sublist, but cannot be a list element of the sublist. If it is a sublist auxiliary pointer, it points at a list continuation of the sublist. If it is the value **min::EMPTY_SUBLIST()**, it represents an empty sublist.

There are several rules that the list level obeys that lead to some efficiencies:

> **No Superfluous min::LIST_END()'s**. A list or sublist auxiliary pointer may not point at an auxiliary area element containing a **min::LIST_END()** value.
>
> **No Superfluous List Auxiliary Pointers**. A list or sublist auxiliary pointer may not point at an auxiliary area element containing a list auxiliary pointer.
>
> **No List Sharing**. Parts of lists may not be shared with other lists. In other words, there is only one way to reach any element of the auxiliary area using object list level functions.
>
> Or more specifically, list and sublist auxiliary pointers must point at elements of the auxiliary area, two list or sublist auxiliary pointers are not permitted to point at the same auxiliary area element, and an auxiliary area element that is followed in the auxiliary area by a list element may not be pointed at by a list or sublist auxiliary pointer (because the following list element in effect points at the auxiliary area element as being the continuation of the list containing the following list element).

Thus if the one way to reach an element of the auxiliary area is deleted, the element may be put on a list of free elements for the auxiliary area. As an optimization, an implementation may use the unused area or space in the object's header to hold a count of the freed elements as an aid to determine when to reorganize the object. It is even possible to establish a list of free elements for reuse, but due to auxiliary area fragmentation this may not be efficient.

As an optimization, ***auxiliary stubs*** can be used in place of object auxiliary area elements. When this is done, the description of this section must be modified as explained in 5.19.6.2 p201.

Auxiliary pointers that are used to build lists cannot be stored as values of list elements. The following function returns **true** if and only if its argument can be stored in a list element:

```
bool min::is_list_legal ( min::gen v )
```

Only auxiliary general values (with subtypes **GEN_..._AUX**) and illegal general values (with subtype **GEN_ILLEGAL**) cannot be stored in a list element, with the exception of **min::EMPTY_SUBLIST()**, which can be stored like a normal value in a list element.

5.19.6.1 List Pointers. A ***list pointer*** can be used to move around in a object at the list level. There are three kinds of list pointers. A **min::list_ptr** read-only list pointer permits read-only access to list elements. A **min::list_updptr** read-write or updatable list pointer permits read-write access to list elements, but does not permit adding or removing list elements. A **min::list_insptr** insertable list pointer permits adding and removing elements, in addition to permitting read-write access to elements.

Note that unlike vector pointers, you cannot downcast more capable list pointers to less capable list pointers. E.g., a **min::obj_vec_insptr** object vector pointer may be explicitly or implicitly downcast to a **min::obj_vec_updptr** or **min::obj_vec_ptr** object vector pointer, but a **min::list_insptr** list pointer may not be explicitly or implicitly downcast to an **min::list_updptr** or **min::list_ptr** list pointer.

The functions for using a **min::list_ptr** read-only list pointer are:

```
(constructor) min::list_ptr lp
                    ( min::obj_vec_ptr & vp )
min::list_ptr & operator =
                    ( min::list_ptr & lp,
                      min::obj_vec_ptr & vp )
min::obj_vec_ptr & min::obj_vec_ptr_of
                    ( min::list_ptr & lp )

min::gen min::start_hash
                    ( min::list_ptr & lp,
                      min::unsptr index )
min::gen min::start_attr
                    ( min::list_ptr & lp,
                      min::unsptr index )

min::gen min::start_copy
                    ( min::list_ptr & lp,
                      min::list_ptr & lp2 )
min::gen min::start_copy
                    ( min::list_ptr & lp,
                      min::list_updptr & lp2 )
min::gen min::start_copy
                    ( min::list_ptr & lp,
                      min::list_insptr & lp2 )
```

```
min::gen min::start_sublist
                  ( min::list_ptr & lp,
                    min::list_ptr & lp2 )
min::gen min::start_sublist
                  ( min::list_ptr & lp,
                    min::list_updptr & lp2 )
min::gen min::start_sublist
                  ( min::list_ptr & lp,
                    min::list_insptr & lp2 )
min::gen min::start_sublist ( min::list_ptr & lp )

min::gen min::next ( min::list_ptr & lp )
min::gen min::peek ( min::list_ptr & lp )
min::gen min::current ( min::list_ptr & lp )
min::gen min::update_refresh
                  ( min::list_ptr & lp )
min::gen min::insert_refresh
                  ( min::list_ptr & lp )

min::unsptr min::hash_size_of ( min::list_ptr & lp )
min::unsptr min::attr_size_of ( min::list_ptr & lp )

bool min::is_list_end ( min::gen v )
bool min::is_sublist ( min::gen v )
bool min::is_empty_sublist ( min::gen v )
```

A list pointer is created to move around in a particular object. Once created it can be started on a new list by one of the ***start list functions***: **min::start_hash**, **min::start_attr**, **min::start_copy**, or the 2-argument **min::start_sublist**. The **min::next** function moves forward one element in the current list, while the **min::peek** function returns the next element without moving to it. The list pointer is always pointing at a current element, or at the end of the list. The **min::current** function returns the current element or **min::LIST_END()**.

Assigning a vector pointer to a list pointer (via the '**=**' assignment operator) reconstructs the list pointer to point at the object pointed at by the vector pointer. This must be done whenever the vector pointer itself is reassigned to point at a different object. Once reassigned, the list pointer must be restarted.

Determining whether a list pointer current element value represents a sublist requires the **min::is_sublist** function, which checks whether the value is a sublist auxiliary pointer, is the value **min::EMPTY_SUBLIST()**, or is a pointer to a sublist auxiliary stub as described in 5.19.6.2 ^{p201}. The **min::is_empty_sublist** function, on the other hand, merely checks whether the value is **min::EMPTY_SUBLIST()**.

Determining whether a list pointer current element value represents the end of a list can be done by simply checking whether the value equals **min::LIST_END()**, or can be done equivalently with the **min::is_list_end** function.

If the current element is updated using **min::update** on another updatable or insertable list pointer, the **min::update_refresh** function must be used to reestablish the current element's value, which is ***cached*** in the list pointer. Similarly one of the **min::..._refresh** functions may need to be used when another pointer is used to insert elements into or remove elements from the object, though in some situations this is not adequate and the list pointer must be restarted with a **min::start_...** function (see p199 for details).

A list pointer that has been created but not started by one of the list start functions appears to be at the end of an immutable empty list.

The **min::start_hash** function positions a list pointer at the beginning of the list whose head is at the given index within the hash table, treating the hash table as a vector. This function mod's the index by the hash table size, so the index can be, and usually should be, a hash value returned by **min::hash** (p83). Index 0 refers to the first element of the hash table, and the size of the hash table minus 1 refers to the last element. The size of the hash table may be obtained from the **min::hash_size_of** function. The index is not a body vector index. This function returns the value of the first element of the list, or returns **min::LIST_END()** if the list is empty.

The **min::start_attr** function is analogous except it is given an index within the object attribute vector and positions the list pointer at the beginning of the list whose head is the attribute vector element at that index. Index 0 refers to the first element of the attribute vector, and the maximum index is the size of the attribute vector minus 1. The size of the attribute vector may be obtained from the **min::attr_size_of** function. The index is not a body vector index and is not mod'ed by the attribute vector size.

The **min::start_copy** function positions a list pointer (**lp**) to the same place as another list pointer (**lp2**). The two list pointers must have been constructed from the same object vector pointer. Also note that the list pointer **lp2** must be valid; that is, if it has been invalidated by an update, remove, or insert, it must be made valid by a refresh or restart before **min::start_copy** is executed.

The value of a list element can represent a sublist (see the **min::is_sublist** function described below). The 2-argument **min::start_sublist** function positions a list pointer (**lp**) to the first element of the sublist represented by the current element of another list pointer (**lp2**). The current element of the second pointer must represent a sublist. This function returns the value of the first element of the sublist, or returns **min::LIST_END()** if the sublist is empty. The two list pointers must have been constructed from the same object vector pointer. Also note that the list pointer **lp2** must be valid; that is, if it has been invalidated by an update, remove, or insert, it must be made valid by a refresh or restart before **min::start_sublist** is executed.

The 1-argument **min::start_sublist** function positions a list pointer to the first element of the sublist represented by the current element of the pointer (it is equivalent to 2-argument

min::start_sublist with **lp2** and **lp** being the same). Again the list pointer must be valid before this function is executed.

The **min::next** function moves the list pointer to the next list element of the list the pointer points at, and returns the value of that list element. It returns **min::LIST_END()** if there is no next list element because the end of the list has been reached. After the end of a list has been reached, additional calls to **min::next** will do nothing but return **min::LIST_END()**.

The **min::peek** function returns the next element of the list, i.e., the same value as the **min::next** function would return, but does not change the element the list pointer is pointing at, i.e., does not modify the list pointer.

The **min::current** function just returns the value of the list element the list pointer currently points at, or returns **min::LIST_END()** if there is no such element because the pointer is at the end of a list. This function does not modify the list pointer.

The **min::update_refresh** function must be called for a pointer that points at an element if the **min::update** function is used with <u>another</u> pointer to the same object to update the element, or if the element is a sublist head and a function is used to remove the first element of the sublist or insert elements at the beginning of the sublist or after the first element of the sublist.

The **min::insert_refresh** function must be called if the **min::insert_reserve** function is used with <u>another</u> pointer to the same object and that function returned **true** indicating that it resized the object, or if a resizing function is called for the object (p177). The **min::insert_refresh** function also does what the **min::update_refresh** function does, and therefore it is never necessary to call both functions. There are also situations detailed on p199 in which use of **min::insert_before**, **min::insert_after**, or **min::remove** with <u>another</u> pointer to the same object absolutely requires that the current pointer be restarted by a **min::start_...** function.

A **min::list_updptr** updatable list pointer allows read-write access to list elements without permitting insertion or removal of elements from lists. The functions for using this are:

```
        (constructor) min::list_updptr lp
                          ( min::obj_vec_updptr & vp )
        min::list_updptr & operator =
                          ( min::list_updptr & lp,
                            min::obj_vect_updptr & vp )
min::obj_vec_updptr & min::obj_vec_ptr_of
                          ( min::list_updptr & lp )
```

```
min::gen min::start_hash
                  ( min::list_updptr & lp,
                    min::unsptr index )
min::gen min::start_attr
                  ( min::list_updptr & lp,
                    min::unsptr index )

min::gen min::start_copy
                  ( min::list_updptr & lp,
                    min::list_updptr & lp2 )
min::gen min::start_copy
                  ( min::list_updptr & lp,
                    min::list_insptr & lp2 )

min::gen min::start_sublist
                  ( min::list_updptr & lp,
                    min::list_ptr & lp2 )
min::gen min::start_sublist
                  ( min::list_updptr & lp,
                    min::list_updptr & lp2 )
min::gen min::start_sublist
                  ( min::list_updptr & lp,
                    min::list_insptr & lp2 )
min::gen min::start_sublist
                  ( min::list_updptr & lp )

min::gen min::next ( min::list_updptr & lp )
min::gen min::peek ( min::list_updptr & lp )
min::gen min::current ( min::list_updptr & lp )
min::gen min::update_refresh
                  ( min::list_updptr & lp )
min::gen min::insert_refresh
                  ( min::list_updptr & lp )

min::unsptr min::hash_size_of ( min::list_updptr & lp )
min::unsptr min::attr_size_of ( min::list_updptr & lp )

void min::update
              ( min::list_updptr & lp,
                min::gen value )
```

Functions defined for read-only list pointers are also applicable to updatable (read-write) list pointers with the same results. However, updatable list pointers can<u>not</u> be converted to be read-only list pointers (unlike the situation with vector pointers). Also note that

min::start_sublist works when **lp2** is a read-only list pointer, but **min::start_copy** does not.

The **min::update** function can be used to replace the value of the current element of a list. The value being replaced must be a list element (and not a non-empty sublist or list end), and the new value of the element must be legal for storage in a list element (p190).

If a list pointer **lp2** points at a element that is **min::update**'ed using a different updatable list pointer **lp1**, then **lp2** will be invalid until **min::update_refresh(lp2)**, **min::insert_refresh(lp2)**, or **min::start_...(lp2,...)** has been executed.

Inserting and removing elements from a list requires yet another kind of list pointer, a **min::list_insptr** insertable list pointer. The functions defined for this are:

```
          (constructor) min::list_insptr lp
                             ( min::obj_vec_insptr & vp )
     min::list_insptr & operator =
                             ( min::list_insptr & lp,
                               min::obj_vect_insptr & vp )
min::obj_vec_insptr & min::obj_vec_ptr_of
                             ( min::list_insptr & lp )

min::gen min::start_hash
                ( min::list_insptr & lp,
                  min::unsptr index )
min::gen min::start_attr
                ( min::list_insptr & lp,
                  min::unsptr index )

min::gen min::start_copy
                ( min::list_insptr & lp,
                  min::list_insptr & lp2 )

min::gen min::start_sublist
                ( min::list_insptr & lp,
                  min::list_ptr & lp2 )
min::gen min::start_sublist
                ( min::list_insptr & lp,
                  min::list_updptr & lp2 )
min::gen min::start_sublist
                ( min::list_insptr & lp,
                  min::list_insptr & lp2 )
min::gen min::start_sublist
                ( min::list_insptr & lp )
```

```
min::gen min::next ( min::list_insptr & lp )
min::gen min::peek ( min::list_insptr & lp )
min::gen min::current ( min::list_insptr & lp )
min::gen min::update_refresh
                   ( min::list_insptr & lp )
min::gen min::insert_refresh
                   ( min::list_insptr & lp )

min::unsptr min::hash_size_of ( min::list_insptr & lp )
min::unsptr min::attr_size_of ( min::list_insptr & lp )

void min::update
               ( min::list_insptr & lp,
                 min::gen value )

bool min::insert_reserve S
               ( min::list_insptr & lp,
                 min::unsptr insertions,
                 min::unsptr elements = 0,
                 bool use_obj_aux_stubs =
                     min::use_obj_aux_stubs )
void min::insert_before
               ( min::list_insptr & lp,
                 min::gen * p, min::unsptr n )
void min::insert_after
               ( min::list_insptr & lp,
                 min::gen * p, min::unsptr n )

min::unsptr min::remove
                      ( min::list_insptr & lp,
                        min::unsptr n = 1 )
```

Functions defined for updatable (and read-only) list pointers are also applicable to insertable list pointers with the same results. However, insertable list pointers can<u>not</u> be converted to be updatable or read-only list pointers (unlike the situation with vector pointers). Note that **min::start_sublist** works when **lp2** is a read-only or updatable list pointer, but **min::start_copy** does not.

Also, there is one function, **min::update**, that can do more for an insertable list pointer than it can for an updatable list pointer. For an insertable list pointer (but not for an updatable list pointer) the current value replaced by **min::update** may be a non-empty sublist. Thus **min::update** can be used to truncate a sublist by replacing it with **min::EMPTY_SUBLIST()**.

Making insertions in object lists requires reservation of the necessary space first, in order to be sure that a sequence of insertions will all succeed. The **min::insert_reserve** function reserves space for the given number of insertion function calls and the given total number of list elements to be inserted. If the later is **0**, the default, it is taken to be equal to the number of insertion function calls. If space is not reserved for insertion function calls in this way, the calls will be in error.

Reservations are made against a list pointer and its object, and are not affected by calls to functions such as **min::start_hash**, **min::start_copy**, **min::start_sublist**, and **min::next**, which reposition the list pointer. A reservation on a list pointer may be made as soon as the list pointer has been created and before any start list function has been called for it. Functions like **min::start_copy** do not transfer reservations from one list pointer to another.

Only one pointer at a time for a given object may have an effective reservation. Each call to **min::insert_reserve** for a list pointer to an object invalidates all previous calls to **min::insert_reserve** for any other list pointer to the same object. Errors in this regard involving two different list pointers will be detected if insufficient memory is reserved, but not otherwise.

The **min::insert_reserve** ^{S} function may expand the body of the object in which space is being reserved (using **min::expand** ^{S}, p186). It returns **true** if it expands the object's body, and **false** if it does not.

Importantly **min::insert_reserve** ^{S} is the only list pointer function that may cause a resizing of the object pointed at, and if a resizing occurs (i.e., if **true** is returned), all other list pointers to the same object become invalid and must be refreshed by a call to **min::insert_refresh**.

The **min::insert_reserve** ^{S} function can expand the object if it's size is insufficient, or instead it can depend upon using object auxiliary stubs (5.19.6.2 ^{p201}) if the object runs out of space, in which case **min::insert_reserve** ^{S} must be sure there are sufficient free stubs to satisfy the insertion calls. The last argument to **min::insert_reserve** ^{S} determines which strategy is used. It defaults to the value of **min::use_obj_aux_stubs** (see p201).[9]

The **min::insert_before** function inserts list elements just before the current position of a list pointer and positions the list pointer to the first element inserted. The **min::insert_after** function inserts list elements just after the current position and leaves the pointer position unchanged (pointing at the element before the first inserted element). If the list pointer is at the end of list, **min::insert_before** inserts the elements at the end of the list, and **min::insert_after** is in error. The elements are specified by a length **n** vector **p** of **min::gen** values. **p[0]** is inserted into the list before **p[1]**, etc. If **n=0** elements are to be inserted, both insert functions are no-operations.

To insert a sublist, first insert **min::EMPTY_SUBLIST()**, then position the pointer to the sublist and use **min::start_sublist** to enter the sublist, and then use **min::insert_before** to insert the elements of the sublist. Also, as noted above, you can use **min::update** with

[9]But if the auxiliary stub code is not compiled in, the last argument is treated as if it was always **false**: see **MIN_USE_OBJECT_AUX_STUBS** on p201.

an insertable pointer to change the current element to **`min::EMPTY_SUBLIST()`**.

A sequence of instructions that begins with a call to **`min::insert_reserve`** ^{S} and ends with the last list insertion function for which the beginning call made a reservation is called a '***list insertion sequence***'. A list insertion sequence must not contain any call to a resizing function (^{S}, p177) or reorganizing function (^{O}, p178), aside from the beginning call to **`min::insert_reserve`** ^{S}. In particular, there can be no second call to **`min::insert_reserve`** ^{S}. There can be calls to create and position list pointers, and to read, update, and remove list elements.

If **`min::insert_before`** or **`min::insert_after`** are used to insert elements at the beginning of a sublist or elements just after the first element of a sublist, all list pointers pointing at the sublist viewed as an element inside its containing list become invalid and must be refreshed by a call to **`min::update_refresh`** or **`min::insert_refresh`** or restarted with a call to a **`min::start_...`** function.

The **`min::remove`** function can be used to remove **`n`** consecutive elements of a list, the first of which is the element currently pointed at by the list pointer. After removal, the list pointer points at the first element after the elements removed, or at the end of the list if there is no such element. If there are fewer than **`n`** elements in the list at and after the list pointer current element, then there is no error, but all the elements at and after the current element are removed, and the list pointer is pointed at the end of the list. The returned value is the number of elements actually removed (and is less than **`n`** if the list was too short). Element removal requires no reservation.

If **`min::remove`** is used to remove the first element of a sublist, all list pointers pointing at the sublist viewed as an element inside its containing list become invalid and must be refreshed by a call to **`min::update_refresh`** or **`min::insert_refresh`** or restarted with a call to a **`min::start_...`** function.

If an insertable list pointer **`lp2`** is used to add or remove elements or remove sublists, and there is another list pointer **`lp1`** pointing at the same object, then **`lp1`** is invalidated and must be restarted with a **`min::start_`**... function if **`lp1`** is pointing at:

1. an element that is removed
2. an element in a removed sublist
3. the end of a removed sublist
4. an element adjacent to an element that is inserted or removed
5. the end of a list whose last element is inserted or removed
6. the first element of a sublist when that sublist, viewed as an element of the list containing it, is adjacent in that containing list to an element that is inserted into or removed from that containing list

7. the end of an empty sublist when that sublist, viewed as an element of the list containing it, is adjacent in that containing list to an element that is inserted into or removed from that containing list

The effects can be subtle and not obvious: for example, using **lp2** to insert before or after a list element that is an empty sublist can invalidate **lp1** if **lp1** points at the end of that empty sublist.

If the insertable vector pointer of an insertable list pointer **lp1** is used as an argument to a resizing function (p177) or a reorganizing function (p178) to resize or reorganize the object **lp1** points to, then **lp1** must be restarted with a **min::start_**... function.

The invalidation of **lp1** in the above situations is <u>not</u> a detectable error.

> ***List Implementation Note***: In order to enforce the 'No Superfluous' rules on p190, MUP::**list_insptr**'s keep track of the location of any auxiliary pointer (or equivalent as per 5.19.6.2 *p201*) pointing a the current element. Insertions before or after the current element may change this auxiliary pointer, and may move the current element to another location. Otherwise insertions replace the current element by an auxiliary pointer (or equivalent) pointing at a vector of elements (or equivalent), one of which is a copy of the replaced element. This last is done for insertions after the single element of a one element list headed in the hash table or attribute vector. Any insertion before the end of a list of such a single element list is transformed into an insertion after the list's single element.

An object can be resized with its lists reorganized so as to compact the auxiliary area, and also to move information out of any auxiliary stubs attached to the object and into the auxiliary area. This can be done by the functions:

```
void min::resize S
            ( min::list_insptr & lp,
              min::unsptr unused_size,
              min::unsptr var_size )
bool min::resize S
            ( min::list_insptr & lp,
              min::unsptr unused_size )
```

The size of the unused area is changed by these functions, and the size of the variable area is also changed by the first of these functions.

The various list pointer types are instances of MUP::**list_ptr_type** defined as follows:

```
typedef MUP::list_ptr_type<min::obj_vec_ptr>
                 min::list_ptr;
typedef MUP::list_ptr_type<min::obj_vec_updptr>
                 min::list_updptr;
typedef MUP::list_ptr_type<min::obj_vec_insptr>
                 min::list_insptr;
```

It is important not to instantiate **MUP::list_ptr_type** with any parameter that is not a vector pointer type (which is why it is unprotected).

5.19.6.2 Object Auxiliary Stubs. As an optimization, ***object auxiliary stubs*** can be used instead of object auxiliary area elements. Object auxiliary stubs are a way of adding list elements to an object whose unused area has been exhausted, without relocating the object. Object auxiliary stubs are considered to be ***extensions*** of the object. If the object is later reorganized, use of any object auxiliary stubs that extend the object may be eliminated by moving information from these into a new larger object auxiliary area.

The code implementing object auxiliary stubs can be compiled into a program or left out of a program according to the setting of the macro:

MIN_USE_OBJ_AUX_STUBS
  1 if code to use object auxiliary stubs is to be compiled;
  0 if this code is not to be compiled; the default.

If compiled, the code can be enabled or disabled by setting the following variable:

bool min::**use_obj_aux_stubs**
  true to enable use of object auxiliary stubs; the default
  false to disable use of object auxiliary stubs

The value of an object auxiliary stub is treated like an auxiliary area element value that is always a list element, and is never a list auxiliary pointer or a **min::LIST_END()** value. However, as a list element, the value may be a sublist auxiliary pointer (or equivalent, see below) or a **min::EMPTY_SUBLIST()** value.

The control of an object auxiliary stub is treated like an auxiliary area element value that is always a list auxiliary pointer or a **min::LIST_END()** value, and is never a list element.

The value and control of an object auxiliary stub are treated like consecutive elements of the object auxiliary area, with the control preceding the value in the area, and therefore following the value in list order.

An object auxiliary stub has one of the following two uncollectable stub type codes:

const int min::**LIST_AUX**
  A **min::gen** value or stub control pointing at this stub behaves like a list pointer.

const int min::**SUBLIST_AUX**
  A **min::gen** value pointing at this stub behaves like a sublist pointer.

Any stub pointer to an object auxiliary stub of type **min::LIST_AUX** is treated as a list auxiliary pointer. Such stub pointers may be stored in auxiliary area **min::gen** values or in auxiliary stub controls.

Any stub pointer to an object auxiliary stub of type **min::SUBLIST_AUX** is treated as a sublist auxiliary pointer. Such stub pointers may be stored in auxiliary area **min::gen** values or in auxiliary stub **min::gen** values.

Thus the following rules are obeyed:

> **min::LIST_AUX stubs**. A **min::gen** value that points at a stub S of type **min::LIST_AUX** is the equivalent of a list auxiliary pointer pointing at the list element that is stub S's value.
>
> Similarly, if the control of an object auxiliary stub holds a stub pointer, that pointer must point at an object auxiliary stub S of type **min::LIST_AUX** and the control is the equivalent of a list auxiliary pointer pointing at the list element that is stub S's value.
>
> **min::SUBLIST_AUX stubs**. A **min::gen** value that points at a stub S of type **min::SUBLIST_AUX** is the equivalent of a sublist auxiliary pointer pointing at the list element that is stub S's value.
>
> **Values of Object Auxiliary Stubs**. An object auxiliary stub value is always a list element, and can never be a list auxiliary pointer value, a **min::gen** value pointing at a stub of type **min::LIST_AUX**, or a **min::LIST_END()** value.
>
> However, it can be a sublist auxiliary pointer value, a **min::gen** value pointing at a stub of type **min::SUBLIST_AUX**, or a **min::EMPTY_SUBLIST()** value.
>
> **Controls of Object Auxiliary Stubs**. An object auxiliary stub control is never a list element, and must be a list auxiliary pointer value, a stub pointer value pointing at a stub of type **min::LIST_AUX**, or a **min::LIST_END()** value.
>
> **Pointers to Object Auxiliary Stubs**. Every object auxiliary stub is pointed at by a **min::gen** value or by the control of another object auxiliary stub. There is only one such pointer pointing at each object auxiliary stub. A **min::gen** value pointing at an object auxiliary stub S must be one of the following:

Location of **min::gen**	Type of auxiliary stub
value of auxiliary area element	**LIST_AUX** or **SUBLIST_AUX**
value of auxiliary stub	**SUBLIST_AUX**
value of hash table element	**LIST_AUX**
value of attribute vector element	**LIST_AUX**

Object auxiliary stubs must obey the object list level rules on p190. This means, for example, that a list auxiliary pointer cannot point at an auxiliary area element that holds a **min::gen** value pointing at an object auxiliary stub of type **min::LIST_AUX**, as this would be equivalent to a list auxiliary pointer pointing at an element holding another list auxiliary pointer.

The use of object auxiliary stubs by an implementation is hidden from the user of MIN by object list level functions. There are no functions for dealing explicitly with object auxiliary stubs. There are, however, unprotected functions to read and write stubs of all kinds in 5.7.4 ^{p46} and 5.7.3 ^{p45}, though the only use of these functions on object auxiliary stubs would be for debugging.

5.19.6.3 List Debugging Functions. The following list level functions are only intended for debugging:

```
bool min::list_equal
                ( min::gen v1, min::gen v2,
                  bool include_var = true,
                  bool include_attr = true,
                  bool include_hash = true )
bool min::list_equal
                ( min::obj_vec_ptr & vp1,
                  min::obj_vec_ptr & vp2,
                  bool include_var = true,
                  bool include_attr = true,
                  bool include_hash = true )
```

These functions return true if and only if:

- **v1** and **v2** are both points to objects,
  or equivalently,
  neither **vp1** nor **vp2** equal **min::NULL_STUB**.
- The two object hash table sizes are equal.
- All the lists headed by the two object hashes are equal.
- The two object attribute vector sizes are equal.
- All the lists headed by the two object attribute vectors are equal.
- The two object number of variables are equal.
- The values of the two object variable vector elements are equal.

If the **include_hash** argument is **false**, the tests involving the hash tables are omitted. If the **include_attr** argument is **false**, the tests involving the attribute vector are omitted. If the **include_var** argument is **false**, the tests involving the object variables are omitted.

A printout of the list level object suitable for debugging can be had by the functions:

```
min::printer min::list_print
                        ( min::printer printer,
                          min::gen v,
                          bool include_var = true,
                          bool include_attr = true,
                          bool include_hash = true )
min::printer min::list_print
                        ( min::printer printer,
                          min::obj_vec_ptr & vp,
                          bool include_var = true,
                          bool include_attr = true,
                          bool include_hash = true )
```

The output is indented by the column position at the time the function is called, and the parts of the object indicated by the arguments are included.

The following can be used to compare two lists or print a single list:

```
        bool min::list_equal
                        ( min::list_ptr & lp1,
                          min::list_ptr & lp2 )
min::printer min::list_print
                        ( min::printer printer,
                          min::list_ptr & lp )
```

The list pointer positions are changed, either to the end of the list, or to the locations of the first disagreeing members of the two lists for the equality testing function. The print output is indented by the column position at the time the print function is called.

5.19.7 Object Attribute Level

At the ***attribute level***, the object is a map from attribute-names to flags and multi-sets of values and from attribute-name/reverse-attribute-name pairs to multi-sets of values. Here names are sequences of name components, which are numbers, strings, or labels. The object map is stored in a set of lists which are entries in the hash table or attribute vector. These lists have one of two syntaxes depending upon the setting of the following macro:

MIN_ALLOW_PARTIAL_ATTR_LABELS
1 if partial attribute labels are supported;
0 if partial attribute labels are not supported.

If partial attribute labels are supported, the attribute name given to a function that locates an attribute may be too long, and the function will use only an initial segment of that name, returning the length of that segment.

If partial attribute names are not supported the object lists have the syntax:

hash-table-entry ::= *hash-list*

attribute-vector-entry ::= *attribute-descriptor*

hash-list ::= *attribute-name-descriptor-pair**

attribute-name-descriptor-pair ::= *attribute-name attribute-descriptor*

attribute-name ::= *atom* | *label*

attribute-descriptor ::= *value* | *attribute-sublist*

attribute-sublist ::= *value** *double-arrow-sublist-option flag-set*

double-arrow-sublist ::= *double-arrow-name-descriptor-pair**

double-arrow-name-descriptor-pair ::= *reverse-attribute-name value-multiset*

reverse-attribute-name ::= *atom* | *label*

value-multiset ::= *value* | *value-sublist*

value-sublist ::= *value**

flag-set ::= *control-code**

value ::= *atom* | *label* | *object* | *indirect-pointer*

atom ::= *identifier string* | *number*

indirect-pointer ::= *index* | *indirect-auxiliary*

Here the syntactic categories represent **min::gen** values or lists or sublists of **min::gen** values in the sense of the object list level.

An *X-list* is a list, in the sense of the object list level. Thus a *hash-list* and *vector-list* are lists.

An *X-sublist* is a sublist, in the sense of the object list level, which is to say it is a list that is an element of another list. Thus *attribute-sublists* and *double-arrow-sublists* are sublists.

An *X-option* is an optional element of a list that is an *X* if it is not omitted.

Everything else is a single list element or a sequence of elements in some list or sublist. *X-pairs* are sequences of two elements. *Flag-sets* are sequences of *control-codes*.

An *atom* is a **min::gen** number or identifier string. An *label* is a **min::gen** label. An *object* is a **min::gen** value pointing at an object stub. A *control-code* is a **min::gen** control code value. An *index* is a **min::gen** index value. An *indirect-auxiliary* is a **min::gen** indirect auxiliary value.

The above form of object map maps an *attribute-name* to a list of *values*, a list of flag *control codes*, and a optional *double-arrow-sublist*. This last maps *reverse-attribute-names* to lists of values.

The lists of *values* here represent multisets of *values*. That is, the order of the *values* in the list is not meaningful. Similarly the order of *attribute-name-descriptor-pairs* in a *hash-list* or the order of *double-arrow-name-descriptor-pairs* in a *double-arrow-sublist* is not meaningful. On the other hand, a *flag-set* is actually an ordered list of *control-codes*; here order matters.

A *hash-list* is simply a list of alternating *attribute-names* and *attribute-descriptors*. The long form of an *attribute-descriptor* is an *attribute-sublist* that gives a possibly empty list of *values*, an optional *double-arrow-sublist*, and a possibly empty list of flag control codes. An *attribute-sublist* may be empty. The short form of an *attribute-descriptor* is just a single *value*, and is equivalent to an *attribute-sublist* that contains nothing but that *value*. This is a common case, and is optimized to conserve memory.

An *attribute-name-descriptor-pair* with an empty *attribute-sublist* is equivalent to a missing *attribute-name-descriptor-pair*. Such *attribute-name-descriptor-pairs* are not removed from the object until the object is completely reorganized, in order to avoid invalidating other attribute pointers referencing the object.

An *attribute-vector-entry* is just an *attribute-descriptor* stored in an attribute vector element whose index is the *attribute-name* associated with the *attribute-descriptor*.

Attribute-names can be either *atoms* or *labels*. If an *attribute-name* is an integer *atom* that is in the range of a legal attribute vector index then the associated *attribute-descriptor* is put in the attribute vector element indexed by the integer. Otherwise an *attribute-name-descriptor-pair* containing the *attribute-name* is put in the *hash-list* of the object's hash table entry whose index in the hash table equals the hash of the *attribute-name* modulo the length of the hash table.

A *double-arrow-sublist* is simply a sublist of alternating *reverse-attribute-names* and *value-multisets*. The former are just like *attribute-names* and the latter are just like *attribute-descriptors* that have no flag control codes or *double-arrow-sublists*. The values in the *value-multisets* of *double-arrow-sublists* are restricted to be *objects*, i.e., pointers to objects.

A *double-arrow-name-descriptor-pair* with an empty *value-multiset* is equivalent to a missing *double-arrow-name-descriptor-pair*. Such *double-arrow-name-descriptor-pairs* are not removed from the object until the object is completely reorganized, in order to avoid invalidating other attribute pointers referencing the object.

When partial attribute names are not supported, multi-component attribute names are represented in object data lists as labels. Thus an *attribute-name* is an *atom* if it represents a 1-component attribute name, and a *label* if it represents a several component attribute name, or if it represents a 1-component name whose one component is itself a label.

In the interests of compatibility with the case where partial attribute names are supported, *attribute-names* and *reverse-attribute-names* that are *labels* whose sole element is an *atom* are not permitted in the object data lists, and when functions are presented with such names, the functions replace them with their sole element, namely the *atom*. In addition, *attribute-names* that are labels with no elements are forbidden, for reasons of compatibility.

If partial attribute names are supported the object lists have the following alternative syntax:

hash-table-entry ::= *node-list*

attribute-vector-entry ::= *node-descriptor*

node-list ::= *node-name-descriptor-pair**

node-name-descriptor-pair ::= *attribute-name-component node-descriptor*

attribute-name-component ::= *atom* | *label*

node-descriptor ::= *value* | *node-sublist*

node-sublist ::= *value** *flag-set*
| *value** *child-sublist double-arrow-sublist-option flag-set*

child-sublist ::= *node-name-descriptor-pair**

double-arrow-sublist ::= *double-arrow-name-descriptor-pair**

double-arrow-name-descriptor-pair ::= *reverse-attribute-name value-multiset*

reverse-attribute-name ::= *atom* | *label*

value-multiset ::= *value* | *value-sublist*

value-sublist ::= *value**

flag-set ::= *control-code**

value ::= *atom* | *label* | *object* | *indirect-pointer*

atom ::= *string* | *number*

indirect-pointer ::= *index* | *indirect-auxiliary*

This differs from the previous representation in that the object map is represented by a tree of nodes, where each node is labeled by an *attribute-name-component*. An *attribute-name* is viewed as a sequence of *attribute-name-components*, so the *attribute-name* defines a path in the tree from the root to a node. Each node has an associated *node-descriptor* that contains the node *values*, optional *double-arrow-sublist*, and flag control codes, which are associated with the *attribute-name* that names the path from the root to the node. The *node-descriptor* also may contain an optional *child-sublist* which describes the children of the node in the tree.

A *hash-table-entry* is a *node-list* that gives alternating *attribute-name-component*/*node-descriptor* pairs for children of the object root node. A *attribute-vector-entry* is just a *node-descriptor* whose associated *attribute-name-component* is the index of the attribute vector element that contains the *attribute-vector-entry*. For a node that is a child of the object root, if the child node's *attribute-name-component* is an integer in the legal range of the object's attribute vector indices, then the child node's *node-descriptor* is stored in the indexed vector element. If a vector index is not the first component of any of the object's attribute names, then the corresponding *attribute-vector-entry* must be an empty *node-sublist*.

For a child of the root whose name component is not a legal attribute vector index, its *attribute-name-component*/*node-descriptor* pair is placed in the *hash-table-entry* whose index in the object's hash table equals the hash of the *attribute-name-component* modulo the length of the hash table.

A *node-descriptor* that consists of nothing but a single *value* can be represented by just that *value*; otherwise it is represented by a *node-sublist*. If this last contains a sublist, the first such sublist must be the *child-sublist*, which has the same structure as a *node-list* and describes the children of the node. Otherwise neither the *child-sublist* nor the *double-arrow-sublist* can be present. Note that to avoid ambiguity it is not permitted to omit the *child-sublist* without also omitting the *double-arrow-sublist*, but the *child-sublist* can be empty. Except for these details the *node-descriptor* structure is the same was as that of the *attribute-descriptor* in the case where partial attribute names are not allowed. The *double-arrow-sublist* structure is exactly the same.

A *node-name-descriptor-pair* with an empty *node-sublist* is equivalent to a missing *node-name-descriptor-pair*. Such *node-name-descriptor-pairs* are not removed from the object until the object is completely reorganized, in order to avoid invalidating other attribute pointers referencing the object.

When partial attribute names are supported, an attribute name is viewed as a sequence of *attribute-name-components*. If this sequence is a single *atom*, function arguments representing the name can be either this *atom* or a *label* whose only element is the *atom*. Otherwise function arguments must be non-empty *labels* whose elements are the *attribute-name-components*. Note that zero length labels may not be used as attribute names.

Attribute flags are represented by a *flag-set* in an *attribute-sublist* or *node-sublist*. The *flag-set* is a sequence of **min::gen** control codes.

The flags are numbered 0, 1, 2, ... Flag N corresponds to the bit in the $I+1$'st control code selected by the mask 2^K where $I = \text{floor}(N/\texttt{VSIZE})$, $K = N \bmod \texttt{VSIZE}$, where **VSIZE**, the number of bits in a control code integer, is the value of the **min::VSIZE** constant (p17), which is 24 if **min::gen** values are 32-bits, and 40 if **min::gen** values are 64-bits. A flag is set for an attribute name if and only if its corresponding bit is present and set and in the attribute's *flag-set*. If a flag's bit is not present in the *flag-set*, the flag is treated as if its bit were present and cleared.

Control codes, auxiliary pointers, and some specials values (**min::NONE()**, **min::ANY()**, etc.: see p23), cannot be values of attributes. The following function returns **true** if and only if its argument can be a value of an attribute:

```
bool min::is_attr_legal ( min::gen v )
```

Only special values (with subtype **GEN_SPECIAL**) having indices greater than or equal to the index of **min::NONE()**, control code values (with subtypes **GEN_CONTROL_CODE**), auxiliary general values (with subtypes **GEN_..._AUX**), and illegal general values (with subtype **GEN_ILLEGAL**) cannot be attribute values.

Only pointers to objects (satisfying **min::is_obj**, see p176) can be double-arrow attribute values.

Attribute names must be strings, numbers, or labels (**min::gen** labels as per 5.11 *p79*).

5.19.7.1 Attribute Pointers.

An ***attribute pointer*** can be used to access attribute flags and values in an object. An attribute pointer stores an attribute name and a reverse attribute name. The latter can take the special values **min::NONE()** and **min::ANY()**. The attribute name designates the object attribute pointed at by the attribute pointer. The flag set pointed at is always the flag set of this attribute, regardless of the setting of the reverse attribute name. The attribute name and reverse attribute name together designate the value multiset pointed at. If the reverse attribute name is **min::NONE()**, the value multiset pointed at is the set of all values not associated with any reverse attribute name. If the reverse attribute name is **min::ANY()**, the value multiset is the set of all values that are associated with any reverse attribute name (see p213 for details). Otherwise the value multiset pointed at is the set of values associated with the reverse attribute name stored in the pointer.

There are three kinds of attribute pointers. A read-only **min::attr_ptr** permits read-only access to attribute values and flags. An updatable **min::attr_updptr** permits read-only access to attribute values and flags, and write access to attribute non-multiset-values that already exist, but does not permit values to be added to or removed from attributes. An insertable **min::attr_insptr** permits read-write access to attribute values and flags and also permits values to be added to or removed from attributes.

5.19.7.1.1 Read-Only Attribute Pointers.

Read-only attribute pointers are used to read information from objects. The functions for using a **min::attr_ptr** are:

```
         (constructor) min::attr_ptr ap
                            ( min::obj_vec_ptr & vp )
         min::attr_ptr & operator =
                            ( min::attr_ptr & ap,
                              min::obj_vec_ptr & vp )
min::obj_vec_ptr & min::obj_vec_ptr_of
                            ( min::attr_ptr & ap )
```

Locator Functions:

```
void min::locate
            ( min::attr_ptr & ap,
              min::gen name )
void min::locatei
            ( min::attr_ptr & ap, int name )
void min::locatei
            ( min::attr_ptr & ap, min::unsptr name )
void min::locate
            ( min::attr_ptr & ap,
              min::unsptr & length, min::gen name )

void min::locate_reverse
            ( min::attr_ptr & ap,
              min::gen reverse_name )
void min::relocate
            ( min::attr_ptr & ap )
```

Accessor Functions:

```
min::unsptr min::get
                  ( min::gen * out, min::unsptr n,
                    min::attr_ptr & ap )
   min::gen min::get ( min::attr_ptr & ap )

unsigned min::get_flags
               ( min::gen * out, unsigned n,
                 min::attr_ptr & ap )
    bool min::test_flag
               ( min::attr_ptr & ap,
                 unsigned n )
```

Information Functions:

```
struct min::attr_info
                {
                  min::gen    name;
                  min::gen    value;
                  min::uns64  flags;
                  min::unsptr value_count;
                  min::unsptr flag_count;
                  min::unsptr reverse_attr_count;
                };
```

```
struct min::reverse_attr_info
                {
                  min::gen    name;
                  min::gen    value;
                  min::unsptr value_count;
                };


min::gen min::name_of
                 ( min::attr_ptr & ap )
min::gen min::reverse_name_of
                 ( min::attr_ptr & ap )

bool min::attr_info_of
               ( min::attr_info & info,
                 min::attr_ptr & ap,
                 bool include_reverse_attr = true )
bool min::reverse_attr_info_of
               ( min::reverse_attr_info & info,
               ( min::attr_ptr & ap )

min::unsptr min::attr_info_of
                      ( min::attr_info * out, min::unsptr n,
                        min::attr_ptr & ap,
                        bool include_reverse_attr = true,
                        bool include_attr_vec = false )
       void min::sort_attr_info
                      ( min::attr_info * out, min::unsptr n )

min::unsptr min::attr_info_of
                      ( min::attr_info * out, min::unsptr n,
                        min::obj_vec_ptr & vp,
                        bool include_reverse_attr = true,
                        bool include_attr_vec = false )

min::unsptr min::reverse_attr_info_of
                      ( min::reverse_attr_info * out, min::unsptr n,
                        min::attr_ptr & ap )
       void min::sort_reverse_attr_info
                      ( min::reverse_attr_info * out, min::unsptr n )
```

Attribute pointers are tied to an object, which is specified by giving a vector pointer to the object to the constructor of the attribute pointer. The **min::obj_vec_ptr_of** function can be used to retrieve the vector pointer used by an attribute pointer.

Assigning a vector pointer to an attribute pointer (via the '`=`' assignment operator) reconstructs the attribute pointer to point at the object pointed at by the vector pointer. This must be done whenever the vector pointer itself is reassigned to point at a different object. Once reassigned, the effect of previous locate functions is lost.

5.19.7.1.2 Attribute Locator Functions. Attribute locator functions find attributes within an object and set the current attribute name and reverse attribute name of an attribute pointer. The **`min::locate`** function sets the attribute name of the pointer. If the '**`length`**' argument is not given, **`min::locate`** uses the complete attribute name. The **`min::locatei`** function does the same thing as **`min::locate`** but is optimized for the case where the attribute name has a single component that is an integer.[10]

The form of the **`min::locate`** function that takes a '**`length`**' argument exists only if the **`MIN_ALLOW_PARTIAL_ATTR_LABELS`** (p204) macro is set to **`1`**. This form of **`min::locate`** uses the longest initial segment of the attribute name that locates an attribute that has a non-empty multiset of values. The length of this initial segment is returned in the '**`length`**' argument. If **`0`** is returned in '**`length`**', no initial segment having a non-empty multiset of values was found.

The **`min::locate_reverse`** function sets the reverse attribute name of a pointer. This can take the special value **`min::NONE()`** or **`min::ANY()`**, as noted above. A call to **`min::locate`** or **`min::locatei`** sets the reverse attribute name to **`min::NONE()`**.

Both the **`min::locate`** and **`min::locate_reverse`** functions take name arguments that are labels (5.11 ^{p79}). If an atom (number or string) is given instead, this is treated as the equivalent of a label whose only element is the atom.

If an insertable attribute pointer **`ap1`** is used to add or remove attribute values or set or clear attribute flags, every other attribute pointer **`ap2`** to the object will become ***invalid*** (p218) until a **`min::locate`** or **`min::relocate`** function is called for **`ap2`**. The later function is equivalent to repeating the calls to **`min::locate`**, and if necessary **`min::locate_reverse`**, that positioned **`ap2`** to where it was before **`ap1`** was used to modify the object. Note that simply calling **`min::locate`** and **`min::locate_reverse`** on **`ap1`** will not add or remove attribute values or set or clear attribute flags.

It is an **undetected error** to use an invalid attribute pointer. Note, however, that for **`ap2`** to become invalid as above, it must share the object vector pointer with **`ap1`**, as otherwise an error will be detected when either **`ap1`**'s vector pointer or **`ap2`**'s vector pointer is created.

Note that no special functions need to be called when an attribute value is updated using **`min::update`** on an updatable attribute pointer (p216).

[10] In some implementations **`min::gen`** is defined to be **`min::uns64`**, and in these implementations **`int`** and **`min::unsptr`** are incorrectly convertible to the **`min::gen`** type by a C++ implicit conversion. As a consequence of this the **`min::locatei`** function must have a different name from the **`min::locate`** function.

5.19.7.1.3 Attribute Accessor Functions. Attribute accessor functions can be used for reading attribute values and flags. For the purposes of describing these functions, an attribute pointer is considered to point at the attribute named by the pointer's attribute name, and the reverse attribute pointed at by the pointer's attribute and reverse attribute names, unless the latter are **`min::NONE()`** or **`min::ANY()`**. If the **`min::locate`** function has never been called to set these names it is an error to call an accessor function for the pointer.

The value multiset pointed at by a pointer is that of the attribute pointed at if the reverse attribute name is **`min::NONE()`**, or is that of the reverse attribute pointed at if the reverse attribute name is not **`min::NONE()`** or **`min::ANY()`**. If the reverse attribute name is **`min::ANY()`** the value multiset is the set of all values associated with any reverse attribute name, but this **`min::ANY()`** value set can only be read by the 3-argument **`min::get`** function, and made empty by the **`min::set`** ^{S} function with 0 new values (see p221). It is an error to attempt any other access to this **`min::ANY()`** value set.

The flag set pointed at by a pointer is that of the attribute pointed at regardless of the setting of the reverse attribute name. If there are N control codes in a flag set, N**`*VSIZE`** flags are represented by the set, and any flag with number $\geq N$**`*VSIZE`** is read as zero. 'High order' control codes may contain all zero flags, and such zero high order control code may be removed when an object is reorganized.

Attributes whose value and flag sets have never been set are treated as if they have empty value multisets and flag control code lists. Similarly reverse attributes whose value multisets have never been set are treated as having empty value multisets.

The 3-argument **`min::get`** function gets the values in the value multiset pointed at and stores them in the '**`out`**' vector. The total number of values in the value multiset is returned (regardless of the value of **`n`**). The argument **`n`**, the length of '**`out`**', is the maximum number of values that may be returned in '**`out`**'. If there are more then **`n`** values, only the first **`n`** are returned in '**`out`**'. If **`n==0`** the function just returns the number of values in the value multiset.

The 1-argument **`min::get`** function assumes that there is exactly one value in the value multiset and returns that value, or returns the special value **`min::NONE()`** (p23) if the value multiset is empty, or returns the special value **`min::MULTI_VALUED`** (p23) if the value multiset has more than one value.

The **`min::get_flags`** function returns the control codes in the flag set (see Attribute Flags, p208) pointed at and stores these in the **`min::gen`** vector '**`out`**'. The total number of control codes in the flag set is returned (regardless of the value of **`n`**). The argument **`n`**, the length of '**`out`**', is the maximum number of control codes that may be returned in '**`out`**'. If there are more then **`n`** control codes, only the first **`n`** are returned in '**`out`**'. If **`n==0`** the function just returns the number of control codes in the flag set.

In all of this, high order zero control codes are treated as if they did not exist (control codes are stored lowest order first in '**`out`**'). If there are no non-zero flags, no control codes are stored in '**`out`**', and **`0`** is returned as the number of control codes.

In the '`out`' vector, flag N is the bit selected by mask 2^K in the vector element with index I, where $K = N$ mod `VSIZE` and $I = \text{floor}(N/\texttt{VSIZE})$. If there are J control codes in the flag set, and $N \geq J * \texttt{VSIZE}$, then flag N is zero.

The **`min::test_flag`** function returns the value of flag **`n`**.

5.19.7.1.4 Attribute Information Functions. Attribute information functions return information about an attribute pointer. The **`min::name_of`** function returns the **`name`** argument of the last call to a **`min::locate...`** function for an attribute pointer. The **`min::reverse_name_of`** function returns the **`reverse_name`** argument of the last call to **`min::locate_reverse`**, or returns **`min::NONE()`** if no **`min::locate_reverse`** calls have been made since the last **`min::locate...`** call. Both functions require a previous call to a **`min::locate...`** function.

The 3-argument **`min::attr_info_of`** function returns information about the current attribute of an attribute pointer (i.e., the attribute whose name was given by the last **`min::locate...`** call for the attribute pointer). Values stored in the **`min::attr_info`** structure members are as follows:

`name`	Name of the current attribute, i.e., the **`name`** argument of last **`min::locate...`** call. If a name has only one component that is an *atom*, this is that *atom*; all other names are returned as *labels*.
`value`	Value of the current attribute if the attribute has a single value. **`min::NONE()`** if the attribute has no value. **`min::MULTI_VALUED()`** if the attribute has more than one value.
`flags`	First 64 flags of the current attribute. Zero if no current attribute.
`value_count`	Number of values of the current attribute. Zero if no current attribute.
`flag_count`	Minimum number of condition codes required to store the flags of the current attribute (vector length for **`min::get_flags`**). High order zero control codes are treated as if they do not exist. Zero if no current attribute.
`reverse_attr_count`	Number of reverse attribute labels associated with the current attribute that have one or more values. Zero if **`include_reverse_attr`** argument is **`false`** or if there is no current attribute.

The 3-argument **`min::attr_info_of`** function returns **`true`** if any of the 3 counts is non-zero, and **`false`** otherwise. Note that if **`include_reverse_attr`** is **`false`**, the **`reverse_attr_count`** is always zero.

The 4-argument **`min::attr_info_of`** function returns in the **`info`** vector the same information for all attributes of the object pointed at by the attribute pointer that have a non-zero value for at least one of their three counts. The number N of such attributes is returned. Information about the first min(N,**n**) of these attributes is returned in the **info** vector, in arbitrary order. The **min::sort_attr_info** function can be used to sort the vector by **name**. The **min::compare** function, p83, is used to determine the ordering of names. Note that if **include_reverse_attr** is **false**, only information about attributes which have values or flags is returned.

Attributes accessed through the object's attribute vector are not returned by the 4-argument **min::attr_info_of** function unless the **include_attr_vec** argument is **true**. These are just the attributes whose name is a small integer **i** with **0 <= i < m**, where **m** is the size of the object attribute vector. If the **MIN_ALLOW_PARTIAL_ATTR_LABELS** macro is set to **1**, any attribute whose name is a label beginning with such an integer is also not returned.

The 3-argument **min::attr_info_of** function can also take a read-only **min::obj_vec_ptr** pointing at the object instead of a **min::attr_ptr**.

The 2-argument **min::reverse_attr_info_of** function returns information about the current reverse attribute of an attribute pointer (i.e., the attribute whose name was given by the last **min::locate...** call for the attribute pointer and whose reverse name was given by the last subsequent **min::locate_reverse** call). Values stored in the **min::reverse_attr_info** structure members are as follows:

name — **Name** of the current reverse attribute, i.e., the **reverse_name** argument of last **min::locate_reverse** call. **min::NONE()** if there is no such call after the last **min::locate...** call. If a name has only one component that is an *atom*, this is that *atom*; all other names are returned as *labels*.

value — Value of the current reverse attribute if the attribute has a single value. **min::MULTI_VALUED()** if the reverse attribute has more than one value. **min::NONE()** if the reverse attribute has no value (includes cases where **name** is **min::NONE()** or **min::ANY()**).

value_count — Number of values of the current reverse attribute. Zero if **value** member is **min::NONE()**.

The 2-argument **min::reverse_attr_info_of** function returns **true** of **value_count** is non-zero, and **false** otherwise.

The 3-argument **min::reverse_attr_info_of** function returns in the **info** vector the same information for all reverse attributes of the current attribute that have one or more values. The number N of such reverse attributes is returned. Information about the first min(N,**n**) of these reverse attributes is returned in the **info** vector, in arbitrary order. The **min::sort_reverse_attr_info** function can be used to sort the vector by **name**. The **min::compare** function, p83, is used to determine the ordering of names.

5.19.7.1.5 Updatable Attribute Pointers. Updatable attribute pointers can set existing values as well as perform the operations of read-only attribute pointers. The functions for using a **min::attr_updptr** are:

```
            (constructor) min::attr_updptr ap
                              ( min::obj_vec_updptr & vp )
          min::attr_updptr & operator =
                              ( min::attr_updptr & ap,
                                min::obj_vec_updptr & vp )
min::obj_vec_updptr & min::obj_vec_ptr_of
                              ( min::attr_updptr & ap )
```

Locator Functions:

```
void min::locate
              ( min::attr_updptr & ap,
                min::gen name )
void min::locatei
              ( min::attr_updptr & ap,
                int name )
void min::locatei
              ( min::attr_updptr & ap,
                min::unsptr name )
void min::locate
              ( min::attr_updptr & ap,
                min::unsptr & length, min::gen name )

void min::locate_reverse
              ( min::attr_updptr & ap,
                min::gen reverse_name )
void min::relocate
              ( min::attr_updptr & ap )
```

Accessor Functions:

```
min::unsptr min::get
                     ( min::gen * out, min::unsptr n,
                       min::attr_updptr & ap )
   min::gen min::get
                     ( min::attr_updptr & ap )
```

```
unsigned min::get_flags
                  ( min::gen * out, unsigned n,
                    min::attr_updptr & ap )
    bool min::test_flag
                  ( min::attr_updptr & ap,
                    unsigned n )

min::gen min::update
                  ( min::attr_updptr & ap,
                    min::gen v )
```

Information Functions:

```
min::gen min::name_of
                  ( min::attr_updptr & ap )
min::gen min::reverse_name_of
                  ( min::attr_updptr & ap )

bool min::attr_info_of
              ( min::attr_info & info,
                min::attr_updptr & ap,
                bool include_reverse_attr = true )
bool min::reverse_attr_info_of
              ( min::reverse_attr_info & info,
              ( min::attr_updptr & ap )

min::unsptr min::attr_info_of
                     ( min::attr_info * out, min::unsptr n,
                       min::attr_updptr & ap,
                       bool include_reverse_attr = true,
                       bool include_attr_vec = false )
min::unsptr min::reverse_attr_info_of
                     ( min::reverse_attr_info * out, min::unsptr n,
                       min::attr_updptr & ap )
```

Functions defined for read-only attribute pointers are also applicable to updatable attribute pointers with the same results. However, updatable attribute pointers can<u>not</u> be converted to be read-only attribute pointers (unlike the situation with vector pointers).

The **min::update** function requires that the current value multiset have a single value and that it be the value multiset of an attribute, and not a reverse attribute. This function replaces the single value, returning the previous value. It is an error if the value multiset is empty or has more than one value, or if the attribute pointer reverse attribute name is not **min::NONE()**. It is an error if the new value is not a legal attribute value (see p208).

5.19.7.1.6 Insertable Attribute Pointers. Insertable attribute pointers can remove and insert values and set and clear flags, as well as perform the operations of updatable attribute pointers. The functions for using a **min::attr_insptr** are:

```
              (constructor) min::attr_insptr ap
                                  ( min::obj_vec_insptr & vp )
           min::attr_insptr & operator =
                                  ( min::attr_insptr & ap,
                                    min::obj_vec_insptr & vp )
min::obj_vec_insptr & min::obj_vec_ptr_of
                                  ( min::attr_insptr & ap )
```

Locator Functions:

```
void min::locate
            ( min::attr_insptr & ap,
              min::gen name )
void min::locatei
            ( min::attr_insptr & ap,
              int name )
void min::locatei
            ( min::attr_insptr & ap,
              min::unsptr name )
void min::locate
            ( min::attr_insptr & ap,
              min::unsptr & length, min::gen name )

void min::locate_reverse
            ( min::attr_insptr & ap,
              min::gen reverse_name )
void min::relocate
            ( min::attr_insptr & ap )
```

Accessor Functions:

```
min::unsptr min::get
                   ( min::gen * out, min::unsptr n,
                     min::attr_insptr & ap )
   min::gen min::get
                   ( min::attr_insptr & ap )
```

```
unsigned min::get_flags
                  ( min::gen * out, unsigned n,
                    min::attr_insptr & ap )
    bool min::test_flag
                  ( min::attr_insptr & ap,
                    unsigned n )

min::gen min::update
                  ( min::attr_insptr & ap,
                    min::gen v )

void min::set^S
              ( min::attr_insptr & ap,
                const min::gen * in, min::unsptr n )
void min::set^S
              ( min::attr_insptr & ap,
                min::gen v )

void min::add_to_set^S
              ( min::attr_insptr & ap,
                const min::gen * in, min::unsptr n )
void min::add_to_set^S
              ( min::attr_insptr & ap,
                min::gen v )
void min::add_to_multiset^S
              ( min::attr_insptr & ap,
                const min::gen * in, min::unsptr n )
void min::add_to_multiset^S
              ( min::attr_insptr & ap,
                min::gen v )

min::unsptr min::remove_one
                     ( min::attr_insptr & ap,
                       const min::gen * in, min::unsptr n )
min::unsptr min::remove_one
                     ( min::attr_insptr & ap,
                       min::gen v )
min::unsptr min::remove_all
                     ( min::attr_insptr & ap,
                       const min::gen * in, min::unsptr n )
min::unsptr min::remove_all
                     ( min::attr_insptr & ap,
                       min::gen v )
```

```
void min::set_flags^S
             ( min::attr_insptr & ap,
               const min::gen * in, unsigned n )
void min::set_some_flags^S
             ( min::attr_insptr & ap,
               const min::gen * in, unsigned n )
void min::clear_some_flags^S
             ( min::attr_insptr & ap,
               const min::gen * in, unsigned n )
void min::flip_some_flags^S
             ( min::attr_insptr & ap,
               const min::gen * in, unsigned n )

bool min::set_flag^S
             ( min::attr_insptr & ap,
               unsigned n )
bool min::clear_flag^S
             ( min::attr_insptr & ap,
               unsigned n )
bool min::flip_flag^S
             ( min::attr_insptr & ap,
               unsigned n )
```

Information Functions:

```
min::gen min::name_of
                 ( min::attr_insptr & ap )
min::gen min::reverse_name_of
                 ( min::attr_insptr & ap )

bool min::attr_info_of
             ( min::attr_info & info,
               min::attr_insptr & ap,
               bool include_reverse_attr = true )
bool min::reverse_attr_info_of
             ( min::reverse_attr_info & info,
             ( min::attr_insptr & ap )
```

```
min::unsptr min::attr_info_of
                    ( min::attr_info * out, min::unsptr n,
                      min::attr_insptr & ap,
                      bool include_reverse_attr = true,
                      bool include_attr_vec = false )
min::unsptr min::reverse_attr_info_of
                    ( min::reverse_attr_info * out, min::unsptr n,
                      min::attr_insptr & ap )
```

Functions defined for updatable attribute pointers are also applicable to insertable attribute pointers with the same results. However, insertable attribute pointers can<u>not</u> be converted to be updatable or read-only attribute pointers (unlike the situation with vector pointers).

The **min::set** ^{S} function sets all the values in the value multiset the attribute pointer is pointing at, deleting any previous values. For 3-argument **min::set** ^{S} the new values are given in the '**in**' vector, and the number of values is given in **n**. If **n** is zero, all values are deleted. For 2-argument **min::set** ^{S} there is one new value given as an argument, unless **v** is **min::NONE()**, in which case there are no new values and all previous values are simply deleted.

For the **min::set** ^{S} function the pointer's reverse attribute name can be **min::NONE()** but cannot be **min::ANY()**, with the exception of the case where all values are being deleted. If the reverse attribute name is **min:ANY()** and the **min::set** ^{S} function is called to delete all previous values, all double arrows with the current attribute name and any reverse attribute name are deleted.

The **min::add_to_set** ^{S} function adds values to the multiset set of values, but adds each value if and only if the value is not already in the multiset, using **==** to compare values for equality. The **min::add_to_multiset** ^{S} function adds values even if they are already in the multiset. Both functions have a 3-argument version with a vector of new values, and a 2-argument version with a single new value. In the latter case if **v** is **min::NONE()** there is no new value and the function is a no-operation. For these functions the pointer's reverse attribute name can be **min::NONE()** but cannot be **min::ANY()**.

It is an error if an attribute value to be stored by a **min::set**, **min::add_to_set** or **min::add_to_multiset** is not a legal attribute value (see p208) or if the pointer's reverse attribute name is not **min::NONE()** and the value is not a pointer to an object.

The **min::remove_one** function removes values from the multiset set of values, removing only one copy of each value, using **==** to compare values for equality. The **min::remove_all** function removes all copies of each value from the multiset of values (it may be less efficient if the multiset is actually a set). Both functions have a 3-argument version with a vector of values to remove, and a 2-argument version with a single value to remove. In the latter case if **v** is **min::NONE()** there is no value to remove and the function is a no-operation. These functions return the number of values actually removed from the multiset of values. For these functions the pointer's reverse attribute name can be **min::NONE()** but cannot be

min::ANY().

If the attribute pointer is pointing at a multiset of values associated with a reverse attribute name, the values represent double arrows. In this case, let the object whose attribute value is being set or removed be O_1, the pointer attribute name be N, the pointer reverse attribute name be R, and the object being pointed at by the value being set or removed be O_2. Then when the value is set or removed, the corresponding value that is an attribute value of O_2 with name R, reverse name N, and target O_1 will be set or removed. That is, the other end of the double arrow will also be set or removed. In the unusual special case where $O_1 = O_2$ and also $N = R$, only one value is set or removed, so the value will not be duplicated when set and is assumed to be unduplicated when removed.

The **min::set_flags** ^{S} function sets the control codes of the flag set the attribute pointer points at. The new control codes are given in the '**in**' vector which has length **n**: see Attribute Flags, p208, for the format of this vector. The elements of the vector must be control codes, as per **min::is_control_code**. If the previous flag set had more control codes that the new flag set, the extra control codes will be retained but will be zeroed. If **n** is zero, all existing control codes are zeroed. When the object is reorganized, high order zero control codes will be deleted.

The function **min::set_some_flags** ^{S}, **min::clear_some_flags**, and **min::flip_some_flags** can be used to alter flags. If the bit corresponding to the flag is on in the '**in**' vector, it is set, cleared, or flipped, respectively. The elements of the vector must be control codes, as per **min::is_control_code**.

The **min::set_flag** ^{S}, **min::clear_flag**, and **min::flip_flag** functions can be used to alter a single flag whose number is given as the argument **n**. The flag is set, cleared, or flipped, respectively. The previous value of the flag is returned.

5.19.7.2 Object Attribute Short-Cuts. The following functions use vector and attribute pointers internally to access a single attribute. They are inefficient if more than one attribute of an object needs to be accessed, or if the attribute that needs to be accessed needs to be accessed more than once.

These functions execute the same operations and return the same values as the corresponding functions on a **min::attr_insptr** insertable attribute pointer pointing at the **obj** argument and located using the **attr** argument as attribute label (p218), with two differences:

1. When possible a **min::attr_ptr** or **min::attr_updptr** is used instead of a **min::attr_insptr**. This permits functions that do not modify the object to be used on a public object (p178), and also improves efficiency slightly.

2. When the **obj** argument is <u>not</u> a pointer to an object (**min::is_obj** is false), the short-cut 2-argument **min::get** function returns **min::NONE()** and the short-cut 4-argument **min::get** function returns **0**.

With this in mind, the short-cut functions are:

```
    min::gen min::get
                     ( min::gen obj, min::gen attr )
min::unsptr min::get
                     ( min::gen * out, min::unsptr n,
                       min::gen obj, min::gen attr )

     bool min::test_flag
                  ( min::gen obj, min::gen attr,
                    unsigned n )
unsigned min::get_flags
                  ( min::gen * out, unsigned n,
                    min::gen obj, min::gen attr )

min::gen min::update
                  ( min::gen obj, min::gen attr,
                    min::gen v )

void min::set^S
              ( min::gen obj, min::gen attr,
                min::gen v )
void min::set^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, min::unsptr n )

void min::add_to_set^S
              ( min::gen obj, min::gen attr,
                min::gen v )
void min::add_to_set^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, min::unsptr n )
void min::add_to_multiset^S
              ( min::gen obj, min::gen attr,
                min::gen v )
void min::add_to_multiset^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, min::unsptr n )
```

```
min::unsptr min::remove_one
                    ( min::gen obj, min::gen attr,
                      min::gen v )
min::unsptr min::remove_one
                    ( min::gen obj, min::gen attr,
                      const min::gen * in, min::unsptr n )
min::unsptr min::remove_all
                    ( min::gen obj, min::gen attr,
                      min::gen v )
min::unsptr min::remove_all
                    ( min::gen obj, min::gen attr,
                      const min::gen * in, min::unsptr n )
```

```
void min::set_flags^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, unsigned n )
void min::set_some_flags^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, unsigned n )
void min::clear_some_flags^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, unsigned n )
void min::flip_some_flags^S
              ( min::gen obj, min::gen attr,
                const min::gen * in, unsigned n )
```

```
bool min::set_flag^S
              ( min::gen obj, min::gen attr,
                unsigned n )
bool min::clear_flag^S
              ( min::gen obj, min::gen attr,
                unsigned n )
bool min::flip_flag^S
              ( min::gen obj, min::gen attr,
                unsigned n )
```

5.19.8 Graph Typed Objects

A ***graph typed object*** is a pair of objects: the ***graph type*** which provides attribute labels and some attribute values, and the ***context*** which provides variables that give other attribute values. The graph type contains variable values that are **`min::gen`** indices which act as indirect pointers to the variables in the context. The context contains a pointer to the graph type most commonly paired with the context. By abuse of language the context may be called a 'graph typed object'.

Graph typed objects may be introduced in two different ways. In the first method the context is given the **min::OBJ_CONTEXT** flag and its first variable points at the graph type. In the second method the graph typed object is represented by a **min::gen** value pointing at a stub of **min::GTYPED_PTR** type that in turn points at an auxilary stub of **min::GTYPED_PTR_AUX** type that holds two pointers to stubs: one pointing at the graph type and the second pointing at the context. The graph type is pointed at by the control of the auxilary stub, and the context is pointed at by the value of the auxilary stub. An auxilary stub is necessary in order to allow garbage collection of the stubs (the control of the primary stub is reserved for GC use). This second method of introducing graph typed objects allows a context to be viewed with several different graph types.

Typed objects make use of two kinds of indirect pointer. The first is just a single index general value (5.2 ^{p13}) in the graph type that holds the index of a variable vector element. This always refers to a variable in the context. The second is a **min::gen** pointer general value (5.5.2.3 ^{p33}) pointing at a a variable in a context. This second kind of index pointer is only supported when it is stored in context variables.

If an attribute value is an indirect pointer, the indirection is taken to the variable element referred to. If that in turn has an indirect pointer value, that indirection is also taken, to any depth. The first indirection must be an index general value, but the remaining indirections can be either kind of indirect pointer.

Graph type objects and contexts are given the **min::OBJ_PUBLIC** flag (p175), so it is not possible to add attributes to a graph type, context, or graph type object. It is also not allowed for a graph type or graph typed object to have attributes with more than one value or attributes with double arrow values.

Graph types are objects with the **min::OBJ_GTYPE** flag. The first variable of a graph type is an integer that bounds the indices stored in the graph type. That is, this integer is the maximum plus `1` of all the indices stored in the graph type.

A graph type is an object that is the root of an acycle graph of objects, all of which <u>must</u> be graph types. However objects with the **OBJ_PUBLIC** flag but <u>without</u> the **OBJ_GTYPE** flag are treated as graph types that do <u>not</u> contain any index general values (i.e., have no indirect pointers to variables).

5.19.8.1 Creating Graph Typed Objects.

To make a new graph type, first construct the graph making use of **min::new_index_gen** (p21) to reference variables in the context. Then execute the function:

```
min::gen min::new_gtype ( min::gen gtype )
```

The **gtype** object must be the root of an acyclic graph whose leaves are non-objects or objects with the **OBJ_PUBLIC** flag (these last leaves may or may not also have their **OBJ_GTYPE** flag set). First all non-root, non-leaf objects in this graph are made into graph types by recursively calling **min::new_gtype**. Then the **gtype** object is compacted and its **OBJ_PUBLIC** flag is set. If there are any index general values in the acyclic graph rooted at the **gtype** object, the first variable of the **gtype** object is set to one plus the maximum index in the graph, and the **gtype** object's **OBJ_GTYPE** flag is set. The function returns the **gtype** argument if the new graph type contains no errors, and otherwise returns **min::ERROR()** and writes an error message into **min::error_message** (p121). Errors include:

1. The graph rooted at the **gtype** object (with leaves as defined above) contains a cycle.
2. The **gtype** object has an attribute with more than one value.
3. The **gtype** object has an attribute with a double arrow value.
4. The **gtype** object has an attribute with an index general value with index **0** (which would mean that the graph type would point to itself via the context first variable).
5. The **gtype** object has an index general value with index larger than **MIN_CONTEXT_SIZE_LIMIT**.
6. The graph rooted at the **gtype** object (with leaves as defined above) contains an object with has one of the above errors when **min::new_gtype** is called recursively for that object.

Error checking makes use of:

MIN_CONTEXT_SIZE_LIMIT	Maximum number of variables in a context. Defaults to **4096**.

It is also possible to activate an additional feature of **min::new_gtype** by giving it more arguments:

```
min::gen min::new_gtype
                    ( min::gen gtype,
                      min::packed_vec_insptr<min::gen> vartab,
                      min::gen varname = min::standard_varname )
min::gen min::standard_varname
                    // Defaults to min::new_str_gen ( "*VARNAME*" )
```

When this is done any leaf object found in the acyclic graph that is a variable name is specially treated. A ***variable name*** is an object with **.type** attribute equal to the **varname** argument, with no other non-hidden attributes (i.e., no other attributes that do not have an **H** flag), and with a single element that is a symbol or label. This element is looked up in **vartab** and if found at **vartab[i]** the variable name object is replaced by an index general value with index **i**. If not found, the symbol is added to the end of **vartab** and treated as if it were found.

The **vartab** argument must be initialized with **vartab[0]** equal to **min::MISSING()**. Other elements if they exist must be **min::MISSING()** or a symbol or label. This will preassign indices to symbols or labels, or in the case of **min::MISSING()** will prevent the use of an index by a name.

To make a new context with a given graph type, execute the function:

```
min::gen min::new_context ( min::gen gtype )
```

The new context is returned. The new context has the number of variables specified by the graph type, and the first of these is set to point at the graph type. All the other variables are set to **min::UNDEFINED()** (p23).

5.19.8.2 Pointers to Graph Typed Objects.

Attribute pointers to objects that might be graph typed must be created by one of the following:

```
(constructor) min::attr_ptr ap
                        ( min::gen obj )
 min::attr_ptr & operator =
                        ( min::attr_ptr & ap,
                          min::gen obj )
(constructor) min::attr_updptr ap
                        ( min::gen obj )
min::attr_updptr & operator =
                        ( min::attr_updptr & ap,
                          min::gen obj )
```

The argument **obj** may by an object or may be a **GTYPED_PTR** pointer (see above). If it is an object, it must be public, even if it is not a context with a graph type. Creating an attribute pointer this way creates vector pointers inside the attribute pointer. If **obj** is <u>not</u> a graph typed object, the vector pointer created inside the attribute pointer can be accessed by the **min::obj_vec_ptr_of** function (see p209 and p216). If **obj** is a graph typed object, the two vector pointers created inside the attribute pointer can be accessed by:

```
   min::obj_vec_ptr & min::graph_obj_vec_ptr_of
                              ( min::attr_ptr & ap )
min::obj_vec_updptr & min::graph_obj_vec_ptr_of
                              ( min::attr_updptr & ap )
   min::obj_vec_ptr & min::context_obj_vec_ptr_of
                              ( min::attr_ptr & ap )
min::obj_vec_updptr & min::context_obj_vec_ptr_of
                              ( min::attr_updptr & ap )
```

The following functions can be used to determine if **obj** was a graph typed object or not:

```
bool min::is_gtyped ( min::attr_ptr & ap )
bool min::is_gtyped ( min::attr_updptr & ap )
```

The usual attribute pointer creation functions that take a vector pointer as argument (see p209, p216, and p218) cannot be used to create **min::list_xxxptr** or **min::attr_xxxptr** pointers pointing at contexts (i.e., objects with the **OBJ_CONTEXT** flag set). Nor can they be used to create **min::list_updptr** or **min::attr_updptr** pointers pointing at graph types (i.e., objects with the **OBJ_GTYPE** flag set). Nor can they be used to create of **min::xxxx_insptr** pointers pointing at contexts or graph types (i.e., objects with the **OBJ_PUBLIC** flag set).

When used with attribute pointers created with the above functions, single argument **min::get** follows the indirect pointers that are index **min::gen** values, treating these as indices of variables in the graph typed object's contest, and also follows indirect pointers that are **min::gen** values pointing at **min::PTR** stubs provided these are stored in context variables. The **min::update** function follows these same indirect pointers, and requires that the location stored into be a context variable.

The **min::update** function cannot be used to store a **min::PTR** type **min::gen** value. Instead, the single argument **min::get** function stores information in the attribute pointer that can be returned by the following function (provided no other **min::get** function execution on the attribute pointer intervenes):

```
min::ptr<min::gen> min::get_ptr_of
                          ( min::attr_updptr & ap )
```

This function returns a **min::ptr<min::gen>** pointer pointing to the variable whose value the **min::get** returned, provided this value equaled **min::UNDEFINED()**. Otherwise a null **min::ptr<min::gen>** pointer is returned.

5.19.8.3 Graph Typed Object Maintenance. The following functions can be used to read and set the **min::OBJ_GTYPE** and **min::OBJ_CONTEXT** flags.

```
bool min::gtype_flag_of
            ( min::obj_vec_ptr & vp )
bool min::context_flag_of
            ( min::obj_vec_ptr & vp )
void min::set_gtype_flag_of
            ( min::obj_vec_insptr & vp )
void min::set_context_flag_of
            ( min::obj_vec_insptr & vp )
```

The functions that set the flags also set the **min::OBJ_PUBLIC** flag and set the insertable object vector pointer to **min::NULL_STUB**, as an insertable object vector pointer is not allowed to point at a public object.

5.19.9 Printing Object General Values

The format for printing object **min::gen** values is:

```
struct min::obj_format
    {
      min::uns32                    obj_op_flags;

      const min::gen_format *       element_format;
      const min::gen_format *       top_element_format;
      const min::gen_format *       label_format;
      const min::gen_format *       value_format;

      const min::gen_format *       initiator_format;
      const min::gen_format *       separator_format;
      const min::gen_format *       terminator_format;

      min::str_classifier           mark_classifier;


      min::pstring                  obj_empty;

      min::pstring                  obj_bra;
      min::pstring                  obj_braend;
      min::pstring                  obj_ketbegin;
      min::pstring                  obj_ket;

      min::pstring                  obj_sep;

      min::pstring                  obj_attrbegin;
      min::pstring                  obj_attrsep;

      min::pstring                  obj_attreol;

      min::pstring                  obj_attreq;

      min::pstring                  obj_attrneg;


      const min::flag_format *      flag_format;
      const min::uns64              hide_flags;

      min::pstring                  obj_valbegin;
      min::pstring                  obj_valsep;
      min::pstring                  obj_valend;
      min::pstring                  obj_valreq;
```

obj_format.obj_op_flags:

```
const min::uns32 min::FORCE_ID
const min::uns32 min::ENABLE_COMPACT
const min::uns32 min::DEFERRED_ID
const min::uns32 min::ISOLATED_LINE
const min::uns32 min::EMBEDDED_LINE
const min::uns32 min::NO_TRAILING_TYPE
const min::uns32 min::ENABLE_LOGICAL_LINE
const min::uns32 min::ENABLE_INDENTED_PARAGRAPH

struct min::flag_format
    {
      min::pstring                            flag_prefix;
      min::pstring                            flag_postfix;
      min::packed_vec_ptr<min::ustring>    flag_names;
    };

const min::flag_format min::standard_attr_flag_format:
    min::left_square_leading_always_pstring    // flag_prefix   "["
    min::trailing_always_right_square_pstring  // flag_postfix  "]"
    min::standard_attr_flag_names              // flag_names


min::packed_vec_ptr<min::ustring> min::standard_attr_flag_names
    // Names of flags numbered 0, 1, 2, ... 63 are:
    //     * + - / @ & # = $ % < > a ... z A ... Z

const unsigned min::standard_attr_a_flag = 12
const unsigned min::standard_attr_A_flag = 38
const unsigned min::standard_attr_hide_flag = 38 + 'H' - 'A'
 const min::64 min::standard_attr_hide_flags =
                        1ull << min::standard_attr_hide_flag
```

```
struct min::flag_parser
    {
      min::uns32 ( * flag_parser_function )
           ( min::uns32 * flag_numbers,
             char * text_buffer,
             const min::flag_parser * flag_parser );
      // Members beyond this point are used by
      // min::standard_flag_parser.

      const min::uns32 *  flag_map;
      min::uns32          flag_map_length;
    };

const min::uns32 * min::standard_attr_flag_map
  const min::uns32 min::standard_attr_flag_map_length
  const min::uns32 min::NO_FLAG = 0xFFFFFFFF
    // If min::standard_attr_flag_names[F] == "\x01\x01" "C"
    // then min::standard_attr_flag_map[C] == F
    // else min::standard_attr_flag_map[C] == min::NO_FLAG

const min::flag_parser min::standard_attr_flag_parser:
     min::standard_flag_parser            // flag_parser_function
     min::standard_attr_flag_map          // flag_map
     min::standard_attr_flag_map_length   // flag_map_length


const min::obj_format min::compact_obj_format:
     min::ENABLE_COMPACT                        // obj_op_flags

     min::compact_gen_format                    // element_format
     NULL                                       // top_element_format
     min::compact_value_gen_format              // value_format
     min::name_gen_format                       // label_format

     min::leading_always_gen_format             // initiator_format
     min::trailing_always_gen_format            // separator_format
     min::trailing_always_gen_format            // terminator_format

     min::standard_str_classifier               // mark_classifier

     min::left_curly_right_curly_pstring        // obj_empty
```

```
    min::left_curly_leading_pstring          // obj_bra
    min::trailing_vbar_leading_pstring       // obj_braend
    min::trailing_vbar_leading_pstring       // obj_ketbegin
    min::trailing_right_curly_pstring        // obj_ket

    min::space_if_none_pstring               // obj_sep

    min::trailing_always_colon_space_pstring // obj_attrbegin
    min::trailing_always_comma_space_pstring // obj_attrsep

    min::erase_all_space_colon_pstring       // obj_attreol

    min::space_equal_space_pstring           // obj_attreq

    min::no_space_pstring                    // obj_attrneg

    min::standard_attr_flag_format           // flag_format
    min::standard_attr_hide_flags            // hide_flags

    min::left_curly_star_space_pstring       // obj_valbegin
    min::trailing_always_comma_space_pstring // obj_valsep
    min::space_star_right_curly_pstring      // obj_valend
    min::space_equal_space_pstring           // obj_valreq


const min::obj_format min::line_obj_format:
    // Same as min::compact_obj_format except for:

      min::ENABLE_COMPACT                    // obj_op_flags
    + min::ENABLE_LOGICAL_LINE

    min::paragraph_gen_format                // top_element_format


const min::obj_format min::paragraph_obj_format:
    // Same as min::compact_obj_format except for:

      min::ENABLE_COMPACT                    // obj_op_flags
    + min::ENABLE_INDENTED_PARAGRAPH

    min::line_gen_format                     // top_element_format
```

```
const min::obj_format min::compact_id_obj_format:
     // Same as min::compact_obj_format except for:

       min::ENABLE_COMPACT                      // obj_op_flags
     + min::DEFERRED_ID

     min::compact_id_gen_format                 // element_format
     min::compact_id_gen_format                 // value_format

     NULL                                       // obj_attr...

     NULL                                       // flag_format
     0                                          // hide_flags

     NULL                                       // obj_val...

const min::obj_format min::id_obj_format:

       min::FORCE_ID                            // obj_op_flags

     // All other elements are NULL or 0

const min::obj_format min::embedded_line_obj_format:
     // Same as min::compact_obj_format except for:

     min::EMBEDDED_LINE                         // obj_op_flags

     NULL                                       // mark_classifier

     NULL                                       // obj_empty

const min::obj_format min::isolated_line_id_obj_format:
     // Same as min::compact_obj_format except for:

     min::ISOLATED_LINE                         // obj_op_flags

     NULL                                       // mark_classifier

     NULL                                       // obj_empty
```

```
NULL                                    // obj_bra
NULL                                    // obj_braend
NULL                                    // obj_ketbegin
NULL                                    // obj_ket

NULL                                    // obj_attrbegin
NULL                                    // obj_attrsep

min::erase_all_space_colon_pstring      // obj_attreol

0                                       // hide_flags
```

An object is printed as the ID '**@**<*object-id*>' if **printer->id_map** exists and the **FORCE_ID** object format operation flag is on.

Otherwise the object is printed in '***compact format***' if all of the following are true:

1. The **ENABLE_COMPACT** object format operation flag is on.
2. The object has no attributes other than **.type**, **.initiator**, **.terminator**, **.separator**, attributes with a **hide_flag**, and attributes with no values, flags, or reverse attribute (double arrow) values.
3. No attribute other than those with a **hide_flag** has multiple values, flags, or reverse attribute (double arrow) values.
4. A value of any attribute that does not have a **hide_flag** is a string or label whose elements are all strings, except that:
   (a) An **.initiator** value may be **min::LOGICAL_LINE()** if the **ENABLE_LOGICAL_LINE** object format operation flag is on.
   (b) A **.terminator** value may be **min::INDENTED_PARAGRAPH()** if the **ENABLE_INDENTED_PARAGRAPH** object format operation flag is on.
5. If the object has an **.initiator** or **.terminator** attribute, it has both, and has no **.type**.

Otherwise, if the **DEFERRED_ID** object format operation flag is on and **printer->id_map** exists the object is printed as the ID '**@**<*object-id*>'.

Otherwise the object is printed in '***isolated line format***' if the **ISOLATED_LINE** object format operation flag.

Otherwise the object is printed in '***embedded line format***' if the **EMBEDDED_LINE** object format operation flag is on.

Otherwise, when none of the above apply, the object is printed in '***normal format***'.

The compact format has several variants:

1. If the object has a `.type` attribute and no other non-hidden attributes, and if any `disable_mapping` argument to the function being used to print the object (e.g., `min::standard_pgen` or `min::print_obj`) is `false`, and if the `.type` attribute value has a non-`NULL_STUB` entry `df` in the defined format type map (p172), then the defined format function is called. E.g.,

   ```
   min::standard_pgen ( printer, v, gen_format, false )
   ```

   or

   ```
   min::print_obj ( printer, v, gen_format, obj_format, obj_op_flags,
                    false )
   ```

   is translated to

   ```
   (* df->defined_format_function) ( printer, v, gen_format, df )
   ```

2. The '***logical line compact format***' is used if the object has an `.initiator` equal to `min::LOGICAL_LINE()`, the object has no `.separator`, and the `ENABLE_LOGICAL_LINE` object format operation flag is on.

3. Otherwise the '***indented paragraph compact format***' is used if the object has a `.terminator` equal to `min::INDENTED_PARAGRAPH()`, the object has no `.separator`, the `ENABLE_INDENTED_PARAGRAPH` object format operation flag is on, and the object is being printed as an element of a containing object that is being printed in logical line compact format.

4. Otherwise the '***bracketed compact format***' is used if the object has an `.initiator` and `.terminator`.

5. Otherwise the '***empty compact format***' is used if the object has no elements and no `.type`. Note that the object may have a `.separator` which will not be printed.

6. Otherwise the '***default compact format***' is used. In this case the object has a `.type`, no `.initiator`, no `.terminator`, but may have a `.separator`.

Keeping these formats and variants in mind, the members of a `min::obj_format` are:

`obj_op_flags`
This is the logical OR of any of the following flags:

FORCE_ID
ENABLE_COMPACT
DEFERRED_ID
ISOLATED_LINE
EMBEDDED_LINE
ENABLE_LOGICAL_LINE
ENABLE_INDENTED_PARAGRAPH

These flags determine the format the object is printed in. See text above.

NO_TRAILING_TYPE

This flag suppresses printing the type in the closing bracket of a non-**.initiator** compact format object and an embedded line format object. That is, if the type is **T**, this flag replaces **|T}** with **|}**. This flag has no effect on compact format objects printed as per the **mark_classifier** as **{T...T}** without any '**|**'s.

element_format
top_element_format

These are the **min::gen_format**'s used to print the object vector elements. If the object is being printed in logical line compact or indented paragraph compact format, **top_element_format** is used. Otherwise **element_format** is used.

label_format
value_format

These are the **min::gen_format**'s used to print the object attribute labels and attribute values, respectively. **Label_format** is also used to print **.type** values T printed as types, e.g., in the forms '**{**T**|**' and '**|**T**}**', and not as attribute values.

initiator_format
separator_format
terminator_format

These are the **min::gen_format**'s used to print the **.initiator**, **.separator**, and **.terminator** in bracketed compact format, and the **.terminator** in logical line compact format. The **.initiator** in indented paragraph compact format is printed using **terminator_format**, as it is in a terminator position.

mark_classifier

This is a **min::str_classifier** used to classify a **.type** string value **T** to see if a compact object printout of the form '**{T|**...**|}**' may be replaced by '**{T**...**T}**', and to classify the two strings in a **.type** label value **[< T1 T2 >]** to see if a compact object printout of the form '**{[< T1 T2 >]|**...**|}**' may be replaced by '**{T1**...**T2}**', See p240.

obj_empty

This is the **min::pstring** printed in empty compact format to represent an empty object that has no elements and no attributes other than **.position** and **.separator**. E.g., '**{}**'.

obj_bra
obj_braend
obj_ketbegin
obj_ket

These are the **min::pstring**'s printed to bracket object elements and attributes in compact, normal, and embedded line format. E.g., '**{**', '**|**', '**|**', and '**}**'.

obj_sep

This member serves as the object element separator in compact or normal format when the object does not have a **.separator** attribute. E.g., a single space.

obj_attrbegin
obj_attrsep

These are the **min::pstring**'s printed in normal format to introduce an object attribute list (e.g., '**:**') and separate attributes in that list. (e.g., '**,**').

obj_attreol

This is the **min::pstring** printed in embedded line and isolated line formats just after the object elements to introduce the indented paragraph containing the object attributes. E.g., '**:**'.

obj_attreq

This is the **min::pstring** printed to separate an attribute label from its value. E.g., '**=**'.

obj_attrneg

This is the **min::pstring** printed to indicate negation of attributes whose value is **FALSE**. Attributes with value **TRUE** are printed as just the attribute label with no following **obj_attreq** or value. Attributes with value **FALSE** are printed as just **obj_attrneg** followed by the attribute label with no following **obj_attreq** or value. E.g., '**no␣**'.

flag_format

This format is used to print attribute flag names. Its value may <u>not</u> be **NULL**. See p243.

WARNING: **flag_format->flag_names** is <u>not</u> locatable by the ACC, and its value must therefore be stored elsewhere in a locatable variable. It is expected that values of **flag_format->flag_names** will never be garbage collected and will be pointed at by **static min::locatable_var** variables.

hide_flags

If an object attribute has any of these flags, the attribute is ignored and not printed when the object is printed. This member is a bit mask for flags **0** through **63**. It is standardly set so the '**H**' flag hides attributes (such as **.position**).

obj_valbegin

obj_valsep
obj_valend
These are the **min::pstring**'s printed to surround and separate values of an attribute that has more than one value. E.g., '**{***', '**,**', and '***}**'.

obj_valreq
This is a **min::pstring** printed to separate a reverse attribute value or list of reverse attribute values from their reverse attribute label, which follows this **min::pstring**. E.g., '**<=**'.

Whenever an object is printed using an object format, the printer format is first saved, automatic breaks are disabled by **min::no_auto_break**, and a break is set by **min::set_break** before the object is printed. The printer format is restored after the object is printed.

In ***bracketed compact format***, where there are **.initiator** and **.terminator** attributes, the only one of the above **pstring**'s that might be used is **obj_sep**, which is used if there is no **.separator** attribute. If

.initiator	is	**"["**
.separator	is	**","**
.terminator	is	**"]"**

then the object is printed in the format:

$$\texttt{[}\ \ell\ \textit{<element>}\ \{\ \textit{ta}\ \texttt{,}\ \text{␣}\ \textit{<element>}\ \}^{\star}\ t\ \texttt{]}$$

where

[	is	the **.initiator** printed with **initiator_format**
ℓ	is	**min::leading**
ta	is	**min::trailing_always**
,	is	the **.separator** printed with **separator_format**
␣	is	a single space character
t	is	**min::trailing**
]	is	the **.terminator** printed with **terminator_format**
<element>	is	printed by **min::print_gen** using the **element_format**

The indent is saved and set by **min::save_indent** just before printing the first element (and restored by **min::restore_indent** after printing the object), and a break is set by **min::set_break** just before printing every element but the first. If the **.separator** attribute is missing, then '*ta* **,** ␣' is replaced by what is printed by **obj_sep**, which would be a single space character for all the above object formats for which compact printing is enabled.

In ***empty compact format***, where there are are no elements and no attributes other than **`.separator`** and attributes with a **`hide_flag`**, just **`obj_empty`** is printed, which for the above compact enabled object formats is just '**`{}`**'.

In ***logical line compact format***, where there are are no attributes other than **`.initiator`**, **`.terminator`**, and attributes with a **`hide_flag`**, the **`.initiator`** is equal to **`min::LOGICAL_LINE()`**, the object elements are just printed using **`top_element_format`** with elements being separated by a single space. A break is set by **`min::set_break`** just before each element is printed.

If there is a **`.terminator`** not equal to **`"<LF>"`**, that is printed just after the last line element. Specifically, it is preceeded by printing **`min::trailing`** and then printed using the **`terminator_format`**. In this case the line is called a ***terminated line***.

Note that nothing is done before the first element to indent the line and <u>no</u> end of line is output after the line. Lines are elements of other objects whose format is supposed to supply pre-line indentation and post-line end of lines, or they are the ends of top level lines that supply pre-line indentation and post-line end of lines.

In ***indented paragraph compact format***, there are are no attributes other than **`.initiator`**, **`.terminator`**, and attributes with a **`hide_flag`**, the **`.terminator`** is equal to **`min::INDENTED_PARAGRAPH()`**, and the object is being printed as an element of a containing object that is being printed in logical line compact format. First single spaces are erased from the end of the current line by **`min::print_erase_space`**, then **`.initiator`** is printed using **`terminator_format`** (as it is in a terminating position), then **`min::eol`** is printed, and lastly the object elements are printed using **`top_element_format`**.

The paragraph indent is set to the indent of its containing logical line plus 4 if the paragraph is the last and only element of its containing logical line that is printed in indented paragraph compact format. Otherwise it is set to the indent of its containing logical line plus 8, while the indent of non-paragraph element sequences of the containing logical line which separate paragraphs is set to the indent of the containing logical line plus 4.

If a paragraph element is preceded by a paragraph element that is a terminated line (i.e., has a non-line-feed **`.terminator`**) and that is printed in the logical line compact format, the paragraph element is separated from this preceding element by a single space. Otherwise the element is preceded by executing **`min::indent`**, **`min::save_indent`**, and **`min::place_indent(4)`** in order and followed by executing **`min::restore_indent`**. Lastly **`min::bol`** is executed at the end of the paragraph.

In ***default compact format***, where there are are no attributes other than **`.type`**, **`.separator`**, and attributes with a **`hide_flag`**, and none of the above compact formats apply. Then if

```
obj_bra        is  "{"
obj_braend     is  "|"
obj_ketbegin   is  "|"
obj_ket        is  "}"
.type          is  "T"
.separator     is  ","
```

where `T` is a string that is <u>not</u> a mark as described below, then the format is

`{T|` *<element>* { *ta* `,` ␣ *<element>* }* `|T}`

which is as for the bracketed compact format above except that '`[` ℓ' has been replaced by '`{T|`' where '`{`' is what `obj_bra` prints and '`|`' is what `obj_braend` prints, and 't `]`' has been replaced by '`|T}`' where '`|`' is what `obj_ketbegin` prints and '`}`' is what `obj_ket` prints. If any explicit spaces, leading spaces, or trailing spaces are to be printed, these must be encoded in the `obj_`... `pstring`'s.

If in this format if there is no `.type` attribute, it is omitted, as in '`{|...|}`'. If there is a `.type` attribute `"T"` and the `NO_TRAILING_TYPE` object format operation flag on, then `T` is omitted from '`|T}`' which becomes '`|}`'.

If in this format there are no *<element>s*, then '`||T`' is omitted so just '`{T}`' is output, regardless of the setting of the `NO_TRAILING_TYPE` flag.

If instead `mark_classifier` is not `NULL` and `T` is a string whose `mark_classifier` defined string class has the `min::IS_MARK` flag, then the format becomes

`{T` *<element>* { *ta* `,` ␣ *<element>* }* `T}`

which is the same as above but with the `|`'s omitted. E.g., if `"+"` is a `.type` value whose `mark_classifier` string class has the `min::IS_MARK` flag, the format is '`{+`...`+}`'. But if there are no *<element>s*, then just '`{T}`' is output; e.g., as in '`{+}`'. Note that if the '`|`'s (i.e., `obj_braend` and `obj_ketbegin`) are omitted by the action of `mark_classifier`, then the `NO_TRAILING_TYPE` object format operation flag has <u>no</u> effect.

If instead `mark_classifier` is not `NULL` and the `.type` attribute is a two element label created by `min::new_lab_gen("T1","T2")` and `T1` and `T2` are strings whose `mark_classifier` computed string classes either <u>both</u> have the `min::IS_MARK` flag, or `"T1"`'s string class has the `min::IS_LEADING` flag and `"T2"`'s string class has the `min::IS_TRAILING` flag, then the format is

`{T1` *<element>* { *ta* `,` ␣ *<element>* }* `T2}`

in which `T1` is used with the opening bracket and `T2` is used with the closing bracket. E.g., if `min::new_lab_gen("<",">")` is a `.type` value with both `"<"` and `">"` being strings whose

mark_classifier string classes both have the **min::IS_MARK** flag, the format is '**{<**...**>}**'. If there are no *<element>s*, '**{T1 T2}**' is output.

Normal format, in which there are attributes other than **.type**, **.separator**, and **.position** that are to be printed as attributes, is like default compact format except that if:

obj_attrbegin	is	**": "**
obj_attrsep	is	**", "**
obj_attreq	is	**" = "**
obj_attrneg	is	**"no "**
obj_valbegin	is	**"{* "**
obj_valsep	is	**", "**
obj_valend	is	**" *}"**

then '**{T|**' is replace by

{T:␣ *<attribute>* { **,**␣ *<attribute>* }* **|**

where

<attribute>	::=	*attribute-label attribute-flags*$^?$ { ␣**=**␣ *attribute-values* }$^?$
	|	*attribute-label attribute-flags*$^?$ ␣**=**␣ *object-ids* ␣**=**␣ *reverse-attribute-label*
	|	*attribute-label*
	|	**no**␣*attribute-label*
attribute-values	::=	*attribute-value*
	|	**{***␣*attribute-value* { **,**␣*attribute-value* }$^+$ ␣***}**
object-ids	::=	*object-id*
	|	**{***␣*object-id* { **,**␣*object-id* }$^+$ ␣***}**
attribute-label	is	printed by **min::print_gen** using **label_format**
reverse-attribute-label	::=	*attribute-label*
attribute-flags	is	printed using **flags_format** (see p243)
attribute-value	is	printed by **min::print_gen** using **value_format**
object-id	is	printed by **min::print_id**

Here

1. An *<attribute>* that is just '*attribute-label*' is equivalent to '*attribute-label* **= TRUE**'.

2. An *<attribute>* that is just '`no` *attribute-label*' is equivalent to '*attribute-label* `= FALSE`'.

3. An *attribute-label* may appear in more that one *<attribute>*, but at most one without any *reverse-attribute-label*, and in at most one with any particular *reverse-attribute-label*.

4. An *attribute-label* may appear with *attribute-flags* in at most one *<attribute>*.

In normal format, if there are no *<element>'s*, then the '`|`'s and the trailing `T` are omitted, regardless of the setting of the `NO_TRAILING_TYPE` flag. Thus '`{T: level = 4}`' is output instead of '`{T: level = 4||T}`'.

The ***embedded line format*** prints the object in the same way as normal format but with *<attribute>*s after the the *<element>*s and with each *<attribute>* on a line by itself. Also any last attribute and following trailing type is separated by an `obj_attrbegin` (e.g., '`:`'). Thus if

`obj_bra`	is	`"{"`
`obj_braend`	is	`"|"`
`obj_sep`	is	`" "`
`obj_ketbegin`	is	`"|"`
`obj_ket`	is	`"}"`
`.type`	is	`"T"`
`obj_attrbegin`	is	`":"`
`obj_attrsep`	is	`","`

then the object in embedded line format has the form:

`{T|` *<element>* { ␣*<element>* }* `|`
  <attribute>`,`
  ....
  <attribute>`,`
`:  T}`

The object begins as if it were in default compact format, but any `.separator` attribute is <u>not</u> used to separate elements (`obj_sep` is used), and the list of elements is terminated by `obj_ketbegin` (`|` here) which introduces the attributes, one per line, with each attribute but the last followed by `obj_attrsep` ('`,`' here) on its line. If attributes are followed by a type, the type is separated from the last attribute by `obj_attrbegin` (`:` here). The object printout begins with a `min::indent` operation and ends with a `min::eol` operation. If there are no attributes other than `.type`, in particular no `.separator` attribute, the embedded line format is identical to normal format except for the `min::indent` and `min::eol` operations.

The ***isolated line format*** prints the object on separate lines, and is indented for use when the object is isolated and <u>not</u> inside a containing object, as when the object is printed when flushing an identifier map (5.18.6 *p169*). If

obj_sep	is	**" "**
obj_attreol	is	**":"**

then the object in isolated line format has the form:

$$<\mathit{element}>\ \{\ {\sqcup}<\mathit{element}>\ \}^{\star}\texttt{:}$$
$$\quad<\mathit{attribute}>$$
$$\quad\cdots\cdots\cdots$$
$$\quad<\mathit{attribute}>$$

Here every attribute, including **.type** and **.separator**, is output on its own indented sub-paragraph line. The output ends with a **min::eol** operation, but does not begin with any special printer operation. If there are no attributes, the indented paragraph and its introducing '**:**' are omitted, but the output ends with **min::eol**. If there are no attributes and no elements, nothing is output but the **min::eol**.

Attribute flags are printed immediately after the attribute's label if the attribute has any flags, according to the **flag_format**. If:

flag_format->flag_prefix	is	**"["**
flag_format->flag_postfix	is	**"]"**
flag_format->flag_names	is	**min::standard_attr_flag_names**

then the attribute label and flags are printed in the format:

attribute-label-and-flags	::=	*attribute-label attribute-flags*$^?$
attribute-flags	::=	**[***attribute-flag-name****]**
attribute-flag-name	::=	***** | **+** | **-** | **/** | **@** | **&** | **#** | **=** | **$** | **%** | **<** | **>**
	|	**a** | ... | **z** | **A** | ... | **Z**
	|	**,***flag-number*

Here *attribute-flag-names* are listed in the order of their flag number, where the correspondence given by **min::standard_attr_flag_names** is

***** ... **>** correspond to flag numbers **0** ... **11**
a ... **z** correspond to flag numbers **12** ... **37**
A ... **Z** correspond to flag numbers **38** ... **63**

Note these names are chosen so that any sequence of them forms a string whose **min::standard_str_classifier** string class has the **min::IS_GRAPHIC** flag and does not have the **min::NEEDS_QUOTES** flag.

Character strings that are sequences of *attribute-flag-names* can be parsed by applying a ***flag parser*** using the function:

```
min::uns32 min::parse_flags
                    ( min::uns32 * flag_numbers,
                      char * text_buffer,
                      const min::flag_parser * flag_parser )
```

This function takes its input character string from the **text_buffer** argument, outputs the flag numbers whose names are in this character string in the **flag_numbers** vector, and returns the length of this vector (i.e., number of flag numbers output). The **flag_numbers** vector must have a length at least as great as the length of (number of characters in) the input **text_buffer**, as the parser may output one flag number per character input, but cannot output more. If there are no errors, the function sets **text_buffer** to the empty string, but if there are errors, it sets it to a string of the form

error-character-sequence { **,** *error-character-sequence* }*

which is made from the input **text_buffer** by replacing every string of correctly interpreted characters by a single comma, and then deleting any beginning or ending commas and any commas following other commas, leaving sequences of incomprehensible characters separated by commas. Incomprehensible characters are ignored by the parser.

Flag parsers accept sequences of digits as flag numbers, provided commas are used to separate these from each other and from other flag names. Commas used to separate flag names are ignored. Digits and commas cannot be used in a flag names.

The first element of a **min::flag_parser** (p231) is a function that executes **min::parse_flags**, so a **min::flag_parser** is in effect a closure. The usual first element is the address of the function:

```
min::uns32 min::standard_flag_parser
                    ( min::uns32 * flag_numbers,
                      char * text_buffer,
                      const min::flag_parser * flag_parser )
```

This function assumes that flag names are of the form **"\x01\x01" "C"** for some single UNICODE character **C**, and uses

```
flag_parser->flag_map[C]
```

as the flag number associated with **C**. The value **min::NO_FLAG** is used to indicate there is no flag number associated with **C**. Also, if

```
C >= flag_parser->flag_map_length
```

then there is no flag number associated with **C**.

The **min::standard_attr_flag_parser** (p231) parses character strings that contain flag names taken from **min::standard_attr_flag_names**. Thus if this parser is given the **text_**

buffer input '***a(b)_c34,200,5d**' the flag numbers output would be 0, 12, 13, 14, 200, 15 and the **text_buffer** output would be '**(,)_,34,5**'.

The following functions can be used to print object general values:

```
min::printer min::print_obj
                    ( min::printer printer,
                      min::gen obj,
                      const min::gen_format * gen_format )
min::printer min::print_obj
                    ( min::printer printer,
                      min::gen obj,
                      const min::gen_format * gen_format,
                      const min::obj_format * obj_format )
min::printer min::print_obj
                    ( min::printer printer,
                      min::gen obj,
                      const min::gen_format * gen_format,
                      const min::obj_format * obj_format,
                      min::uns32 obj_op_flags,
                      bool disable_mapping = false )
```

The second function is equivalent to the first but with the **obj_format** argument replacing **gen_format->obj_format**. The third function is equivalent to the second but with the **obj_op_flags** argument replacing **obj_format->obj_op_flags** and additionally an option to set **disable_mapping** to **true** (see p235 for **disable_mapping** details).

The expression '**printer << min::pgen(v)**' is equivalent to

```
min::print_obj(printer,v)
```

if **v** is an object.

Appendices

A C/C++ Interface

Unless otherwise noted, this interface is defined by `min.h`.

Abbreviations:

These are to be included in user's code, and are not in `min....h` files.

Compilation Macros:

These are in `min_parameters.h`.

Assert Macros and Functions:

These are in `min_parameters.h`.

Stub Type Codes:

Stub Related Functions:

Gen Value Protected Functions:

Gen Value Unprotected Functions:

Gen Value Subtype Codes:

Special Values:

min::gen min:: MISSING() p23
min::gen min:: NONE() p23
min::gen min:: DISABLED() p23
min::gen min:: ENABLED() p23
min::gen min:: ANY() p23
min::gen min:: MULTI_VALUED() p23
min::gen min:: UNDEFINED() p23
min::gen min:: UNUSED() p24
min::gen min:: SUCCESS() p24
min::gen min:: FAILURE() p24
min::gen min:: ERROR() p24
min::gen min:: LOGICAL_LINE() p24
min::gen min:: INDENTED_PARAGRAPH() p24

Body References:

min:: ref<T> MUP:: new_ref
(const min::stub * s,
T & const location) p29
min:: ref<T> MUP:: new_ref<T>
(const min::stub * s,
min::unsptr offset) p29
min:: ref<T> min:: new_ref
(T & location)
where location is not relocatable p29
const min::stub * MUP:: ZERO_STUB p29
const min::stub * const r .s p29
min::unsptr const r .offset p29
min::ref<T> const & operator =
(min::ref<T> const & r, T const & value)
.... p29
min::ref<T> const & operator =
(min::ref<T> const & r,
min::ref<T> const & r2) p29
T operator T
(min::ref<T> const & r) p29
T operator ->
(min::ref<T> const & r) p29
T & operator ~
(min::ref<T> const & r) [unprotected] p29

Body Pointers:

T * operator ~
(min::ptr<T> const & p) [unprotected] p31
min::ptr<T> ptr<T>
(min::ptr<S> const & p) [unprotected] p31
bool operator <
(min::ptr<T> const & p1,
min::ptr<T> const & p2) p33
min::ptr<T> operator ++ [postfix ++]
(min::ptr<T> & p, int) p33
min::ptr<T> operator -- [prefix --]
(min::ptr<T> & p) p33
bool operator ==
(min::ptr<T> const & p1, min::ptr<T> const & p2) ... p33
bool operator !=
(min::ptr<T> const & p1, min::ptr<T> const & p2) ... p33

Body Copy Macros:

MIN_STACK_COPY (type, name, length, source) p34

ACC Locatable Types:

class min:: locatable_var<T> : public T
use min::stub_ptr for T instead of const min::stub * p35

typedef min::locatable_var<min::gen>
min:: locatable_gen p35
typedef min::locatable_var<min::stub_ptr>
min:: locatable_stub_ptr ... p35
In compact implementation:
typedef min::locatable_gen min:: locatable_num_gen p35
In loose implementation:
typedef min::gen min:: locatable_num_gen p35
(constructor) min:: locatable_var<T>
var (void) p35
(constructor) min:: locatable_var<T>
var (min::locatable_var<T> const & var) .. p35
(constructor) min:: locatable_var<T>
var (T const & value) p35
min::locatable_var<T> & operator =
(min::locatable_var<T> & var,
min::locatable_var<T> const & var2) .. p36
min::locatable_var<T> & operator =
(min::locatable_var<T> & var,
T const & value) p36

Control Flags:

Unprotected Body Allocation Functions:

UNICODE Characters:

```
     const min::uns32 min::  IS_DIGIT   ........ p58
     const min::uns32 min::  IS_NATURAL ........ p58
     const min::uns32 min::  IS_LETTER  ........ p58
     const min::uns32 min::  IS_MARK    ........ p58
     const min::uns32 min::  IS_ASCII   ........ p58
     const min::uns32 min::  IS_LATIN1  ........ p58
```

UNICODE Support Control:

```
    inline min::uns32 min::  char_flags
                               ( const min::uns32 * char_flags,
                                 min::support_control sc,
                                 min::Uchar c ) ........ p64

           struct     min::  support_control  ........ p64
           {
                min::uns32   support_mask
                min::uns32   unsupported_char_flags
           }
 const min::uns32 min::  ALL_CHARS   = 0xFFFFFFFF ........ p64
 const min::support_control min::  ascii_support_control   =
      { min::IS_ASCII, min::IS_UNSUPPORTED }; ........ p64
 const min::support_control min::  latin1_support_control   =
      { min::IS_LATIN1 + min::IS_ASCII, min::IS_UNSUPPORTED }; ........ p64
 const min::support_control min::  support_all_support_control   =
      { min::ALL_CHARS, min::IS_UNSUPPORTED }; ........ p64
```

String Classifiers:

```
 typedef min::uns32 ( * min::  str_classifier   )
                                ( const min::uns32 * char_flags,
                                  min::support_control sc,
                                  min::unsptr n,
                                  min::ptr<const min::Uchar> p ) ... p65
       const min::uns32 min::  IS_BREAKABLE  ........ p65
 const min::str_classifier
           min::  standard_str_classifier  ........ p65
 const min::str_classifier
           min::  quote_separator_str_classifier  ........ p65
 const min::str_classifier
           min::  quote_value_str_classifier  ........ p65
 const min::str_classifier
           min::  quote_all_str_classifier  ........ p65
 const min::str_classifier
           min::  null_str_classifier  ........ p65
```

bool min:: is_num (min::gen v) p69

min::gen min:: new_num_gen^R (int v) p69
min::gen min:: new_num_gen^R (min::unsptr v) p69
min::gen min:: new_num_gen^R (min::float64 v) p69

min::int64 min:: int_of (min::gen v) p69
min::float64 min:: float_of (min::gen v) p69

min::uns32 min:: numhash (min::gen v) p69
min::uns32 min:: floathash (min::float64 f) p70

Number Unprotected Functions:

min::float64 MUP:: float_of (min::gen v) p70

String Protected Functions:

min::unsptr min:: max_id_strlen [default 32] p71

min::gen min:: new_str_gen^R (const char * p) p72
min::gen min:: new_str_gen^R
(const char * p, min::unsptr n) p72
min::gen min:: new_str_gen^R (min::ptr<const char> p) p72
min::gen min:: new_str_gen^R
(min::ptr<const char> p, min::unsptr n) p72
min::gen min:: new_str_gen^R (min::ptr<char> p) p72
min::gen min:: new_str_gen^R
(min::ptr<char> p, min::unsptr n) p72
min::gen min:: new_str_gen^R
(const min::Uchar * p, min::unsptr n) p72
min::gen min:: new_str_gen^R
(min::ptr<const min::Uchar> p,
min::unsptr n) p72
min::gen min:: new_str_gen^R
(min::ptr<min::Uchar> p, min::unsptr n) p72
int min:: is_str (min::gen v) p73
bool min:: is_id_str (min::gen v) p73
bool min:: is_non_id_str (min::gen v) p73

String Pointers:

Names:

Packed Structures:

```
min::packed_struct_updptr<S> &   operator  =
                              ( min::packed_struct_updptr<S> & psup,
                                min::gen v ) ........................... p89
min::packed_struct_updptr<S> &   operator  =
                              ( min::packed_struct_updptr<S> & psup,
                                const min::stub * s ) ................. p89
 min::ptr<S>   operator  ->
                 ( min::packed_struct_updptr<S> const & psup ) ...... p89
 min::ref<S>   operator  *
                 ( min::packed_struct_updptr<S> const & psup ) ...... p89
```

Packed Vectors:

```
       struct min::  packed_vec_header<L>
                       {
                         const min::uns32 control;
                         const L length;
                         const L max_length;
                       }; ............................................. p92
(constructor) min::  packed_vec
                             <E,H=min::packed_vec_header<min::uns32>,
                                L=min::uns32>
                             pvtype
                       ( const char * name,
                         const min::uns32 * element_gen_disp = NULL,
                         const min::uns32 * element_stub_disp
                                                  = NULL,
                         const min::uns32 * header_gen_disp = NULL,
                         const min::uns32 * header_stub_disp = NULL ) .. p92
(constructor) min::  packed_vec_with_base<E,H,B,L>   pvtype
                       ( const char * name,
                         const min::uns32 * element_gen_disp = NULL,
                         const min::uns32 * element_stub_disp
                                                  = NULL,
                         const min::uns32 * header_gen_disp = NULL,
                         const min::uns32 * header_stub_disp = NULL ) .. p92
```

Files:

```
void min::  init_input_string^S
              ( min::ref<min::file> file,
                min::ptr<const char> string,
                const min::line_format * line_format = NULL,
                min::uns32 spool_lines = min::ALL_LINES ) ...... p110
void min::  init_input_string^S
              ( min::ref<min::file> file,
                min::ptr<char> string,
                const min::line_format * line_format = NULL,
                min::uns32 spool_lines = min::ALL_LINES ) ...... p110
void min::  init_input_string^S
              ( min::ref<min::file> file,
                const char * string,
                const min::line_format * line_format = NULL,
                min::uns32 spool_lines = min::ALL_LINES ) ...... p110
void min::  init_input_named_file^S
              ( min::ref<min::file> file,
                min::gen file_name,
                const min::line_format * line_format = NULL,
                min::uns32 spool_lines = min::ALL_LINES ) ...... p109

  const min::uns32 min::  NO_LINE  ............................ p112

 min::uns32 min::  next_line^S  ( min::file file ) ............. p112

 min::uns32 min::  remaining_offset  ( min::file file ) ........ p113
 min::uns32 min::  remaining_length  ( min::file file ) ........ p113
 min::uns32 min::  skip_remaining  ( min::file file ) .......... p113
 min::uns32 min::  partial_length  ( min::file file ) .......... p113
 min::uns32 min::  partial_offset  ( min::file file ) .......... p113
 min::uns32 min::  file_is_complete  ( min::file file ) ........ p112
 min::uns32 min::  line
                     ( min::file file,
                       min::uns32 line_number ) ................. p114
 min::uns32 min::  flush_spool^S
                     ( min::file file,
                       min::uns32 line_number =
                           min::NO_LINE ) ....................... p115
 min::uns32 min::  rewind
                     ( min::file file,
                       min::uns32 line_number = 0 ) ............. p115
       void min::  end_line^S  ( min::file file ) ............... p110
       void min::  complete_file  ( min::file file ) ........... p111
```

```
           void min::  end_line^S
                         ( min::file file,
                           min::uns32 offset ) ........................ p114
     min::uns32 min::  flush_file^S
                         ( min::file file,
                           bool copy_completion = true ) ............ p115
     min::uns32 min::  flush_line^S
                         ( min::file file,
                           min::uns32 offset ) ........................ p116
     min::uns32 min::  flush_remaining^S  ( min::file file ) ....... p116
      std::ostream &   operator <<
                         ( std::ostream & out,
                           min::file file ) ........................... p116
           min::file   operator <<^S
                         ( min::file ofile,
                           min::file ifile ) ........................ p116
        min::printer   operator <<^S
                         ( min::printer printer,
                           min::file file ) ........................... p116
```

Identifier Maps:

```
    typedef min::packed_vec_ptr
                       < min::gen,
                         min::id_map_header<min::uns32> >
                   min::  id_map  ..................................... p117
  min::id_map min::  init^R  ( min::ref<min::id_map> map ) .......... p117

          const min::uns32 map->  length  ........................... p117
            const min::gen map[id] .................................. p117
           min::uns32 map->  next  .................................. p118
           min::uns32 map->  occupied  .............................. p118

  min::uns32 min::  find^R
                     ( min::id_map map,
                       min::gen g ) ................................... p117
  min::uns32 min::  find_or_add^R
                     ( min::id_map map,
                       min::gen g ) ................................... p117
```

```
               min::gen map->  ID_prefix
                               // Default: MIN string "!" ................ p118
             min::Uchar map->  ID_character
                               // Default: U'@' .......................... p118
               min::gen map->  ID_assign
                               // Default: MIN string ":=" ............... p118
      min::gen_format * map->  id_gen_format
                               // Default: min::id_map_gen_format ...... p118

    min::gen min::  map_get
                      ( min::id_map map,
                        min::uns32 id ) ............................... p119
        void min::  map_set^R
                      ( min::id_map map,
                        min::uns32 id,
                        min::gen g ) .................................. p119
        void min::  map_clear
                      ( min::id_map map,
                        min::uns32 id ) ............................... p119

    min::gen min::  map_get
                      ( min::id_map map,
                        min::gen symbol ) ............................. p120
        void min::  map_set^R
                      ( min::id_map map,
                        min::gen symbol,
                        min::gen g ) .................................. p120
        void min::  map_clear
                      ( min::id_map map,
                        min::gen symbol ) ............................. p120
        void min::  map_clear_input
                      ( min::id_map map ) ............................. p120
```

Printers:

```
   min::locatable_var<min::printer> min::  error_message   ........... p121

      typedef min::packed_struct_updptr<min::printer_struct>
                            min::  printer   ....................... p121

   min::printer min::  init^R
                         ( min::ref<min::printer> printer,
                           min::file file = min::NULL_STUB ) ....... p121
   min::printer min::  init_ostream^S
                         ( min::ref<min::printer> printer,
                           std::ostream & ostream ) ................. p121
```

```
        min::file printer->  file  ........................ p121
  std::ostream * printer->  ostream  ...................... p121

      struct   min::  line_break  ......................... p122
      {
          min::uns32   offset
          min::uns32   column
          min::uns32   line_length
          min::uns32   indent
      }

               const min::uns32 printer->  column  ........ p122
          const min::line_break printer->  line_break  .... p122
const min::packed_vec_insptr<min::line_break>
                                printer->  line_break_stack   p122

 const min::line_break min::  default_line_break  ......... p122

    struct                 min::  print_format  ........... p123
    {
                  min::uns32      op_flags
                  min::uns32 *    char_flags
           min::support_control   support_control
           min::display_control   display_control
           min::display_control   quoted_display_control
             min::break_control   break_control
        min::char_name_format *   char_name_format
              min::gen_format *   gen_format
                     min::uns32   max_depth
    }

    const min::print_format printer->  print_format  ...... p123
     const min::packed_vec_insptr
             <min::print_format>
                           printer->  print_format_stack  . p123

  const min::print_format min::  default_print_format  ... p123

          min::file printer->  id_map  ................... p123
        min::uns32 printer->  id_map->next  .............. p123
        min::uns32 printer->  depth  ..................... p123
```

`print_format.op_flags`:

```
const min::uns32 min::  EXPAND_HT .............................. p124
const min::uns32 min::  DISPLAY_EOL ............................ p124
const min::uns32 min::  DISPLAY_PICTURE ........................ p124
const min::uns32 min::  DISPLAY_NON_GRAPHIC .................... p124
const min::uns32 min::  FLUSH_ON_EOL ........................... p124
const min::uns32 min::  FLUSH_ID_MAP_ON_EOM .................... p124
const min::uns32 min::  FORCE_SPACE ............................ p124
const min::uns32 min::  DISABLE_STR_BREAKS ..................... p124
const min::uns32 min::  FORCE_PGEN ............................. p124
```

print_format.support_control: p64

print_format.display_control:

```
struct     min::  display_control ................. p124
{
     min::uns32   display_char
     min::uns32   display_suppress
}
const min::display_control min::  graphic_and_sp_display_control   =
      { min::IS_SP + min::IS_GRAPHIC, 0 }; .......................... p124
const min::display_control min:: graphic_and_hspace_display_control =
      { min::IS_HSPACE  + min::IS_GRAPHIC, 0 }; ..................... p124
const min::display_control min:: graphic_only_display_control =
      { min::IS_GRAPHIC, 0 }; ....................................... p124
const min::display_control min:: graphic_and_vhspace_display_control =
      { min::IS_VHSPACE + min::IS_GRAPHIC, 0 }; ..................... p124
const min::display_control min:: display_all_display_control =
      { min::ALL_CHARS, 0 }; ........................................ p124
```

print_format.break_control:

```
struct     min::  break_control ........................ p124
{
     min::uns32   break_before
     min::uns32   break_after
     min::uns32   conditional_break
     min::uns32   conditional_columms
}
```

```
const min::break_control min::  no_auto_break_break_control   =
          { 0, 0, 0, 0 }; ........................................... p125
const min::break_control min::  break_after_space_break_control   =
          { min::IS_BHSPACE, 0, 0, 0 }; ............................. p125
const min::break_control min::  break_before_all_break_control   =
          { 0, min::ALL_CHARS, 0, 0 }; .............................. p125
const min::break_control min::  break_after_hyphens_break_control   =
          { min::IS_BHSPACE, 0, min::CONDITIONAL_BREAK, 4 }; ...... p125
```

print_format.char_name_format:

```
      struct              min::  char_name_format   ................ p125
      {
                   min::ustring   char_name_prefix
                   min::ustring   char_name_postfix
      }
const min::char_name_format min::  standard_char_name_format   =
          { (min::ustring) "\x01\x01" "<",
            (min::ustring) "\x01\x01" ">" }; ....................... p125
```

print_format.line_format:

```
      struct              min::  line_format   ..................... p125
      {
                   min::uns32     op_flags
                   char           mark
                   const char *   blank_line
                   const char *   end_of_file
                   const char *   unavailable_line
                   const char *   line_class
                   const char *   line_number_class
      }
const min::line_format * min::  standard_line_format  : ............ p126
   min::DISABLE_LINE_BREAKS   // op_flags
       + min::EXPAND_HT,
   '^' ,                      // mark
   NULL,                      // blank_line
   "<END-OF-FILE>",           // end_of_file
   "<UNAVAILABLE-LINE>",      // unavailable_line
   "MIN-LINE",                // line_class
   "MIN-LINE-NUMBER"          // line_number_class
const min::line_format * min::  marked_line_format  : .............. p126
   // Same as min::standard_line_format except for:
   '^' ,                      // mark
   "MIN-LINE-MARK"            // line_mark_class
```

```
const min::line_format * min::  picture_line_format  :
   // Same as min::standard_line_format except for:
   min::DISABLE_LINE_BREAKS   // op_flags
       + min::EXPAND_HT
       + min::DISPLAY_PICTURE
   "<BLANK-LINE>"             // blank_line
```
.... p126

```
const min::line_format * min::  non_graphic_line_format  :
   // Same as min::standard_line_format except for:
   min::DISABLE_LINE_BREAKS   // op_flags
       + min::EXPAND_HT
       + min::DISPLAY_PICTURE
       + min::DISPLAY_NON_GRAPHIC
   "<BLANK-LINE>"             // blank_line
```
.... p126

```
const min::line_format * min::  eol_line_format  :
   // Same as min::standard_line_format except for:
   min::DISABLE_LINE_BREAKS   // op_flags
       + min::EXPAND_HT
       + min::DISPLAY_PICTURE
       + min::DISPLAY_NON_GRAPHIC
       + min::DISPLAY_EOL
   "<BLANK-LINE>"             // blank_line
```
.... p126

Adjusting Printer Parameters:

Printing File Lines and Phrases:

```
    min::uns32 min::  print_line^S
                        ( min::printer printer,
                          min::file file,
                          min::uns32 line_number,
                          const min::line_format * line_format =
                                                   NULL ) .......... p151
  (constructor) min::  pline_numbers
                        ( min::file file,
                          min::uns32 first, min::uns32 last ) ...... p152

 min::printer   operator  <<^S
                  ( min::printer printer,
                    min::pline_numbers const & pline_numbers ) ...... p152

  struct min::  position
                  {
                    min::uns32 line;
                    min::uns32 offset;
                  }; .................................................. p152
const min::position min::  MISSING_POSITION
                             = { 0xFFFFFFFF, 0xFFFFFFFF } .......... p153
        bool   operator  ==
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  !=
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  <
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  <=
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  >
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  >=
                  ( const min::position & p1,
                    const min::position & p2 ) ....................... p153
        bool   operator  (bool)
                  ( const min::position & p ) ....................... p153
```

Printing General Values:

```
       min::op min::  pgen  ( min::gen v ) ............................ p157
       min::printer   operator  <<^S
                        ( min::printer printer, min::gen v ) ......... p157
       min::op min::  pgen
                        ( min::gen v,
                          const min::gen_format * gen_format ) ...... p157
       min::op min::  set_gen_format
                        ( const min::gen_format * gen_format ) ...... p157
       min::op min::  pgen_name  ( min::gen v ) ...................... p157
       min::op min::  pgen_quote  ( min::gen v ) ..................... p157
       min::op min::  pgen_never_quote  ( min::gen v ) ............... p157
  min::printer min::  print_gen
                        ( min::printer printer,
                          min::gen v ) ............................... p157
  min::printer min::  print_gen
                        ( min::printer printer,
                          min::gen v,
                          const min::gen_format * f,
                          bool disable_mapping = false ) ............. p157

  typedef min::printer ( * min::  pgen_function  ) ................... p158
                    ( min::printer printer,
                      min::gen v,
                      const min::gen_format * gen_format,
                      bool disable_mapping )
  struct min::  gen_format  .......................................... p159
                    {
                      min::pgen_function pgen;

                      // The following are used by min::standard_pgen:
                      //
                      const min::num_format *  num_format;
                      const min::str_format *  str_format;
                      const min::lab_format *  lab_format;
                      const min::special_format *  special_format;
                      const min::obj_format *  obj_format;

                      // The following is NOT used by min::standard_pgen:
                      //
                      const void *  non_standard;
                    };
```

```
const min::gen_format * min::  compact_gen_format  : ............... p159
        & min::standard_pgen                    // pgen
        min::long_num_format                    // num_format
        min::quote_separator_str_format         // str_format
        min::bracket_lab_format                 // lab_format
        min::bracket_special_format             // special_format
        min::compact_obj_format                 // obj_format
        NULL                                    // non_standard

const min::gen_format * min::  line_gen_format  : ................... p159
        // Same as min::compact_gen_format except for:
        min::line_obj_format                    // obj_format

const min::gen_format * min::  paragraph_gen_format  : ............. p159
        // Same as min::compact_gen_format except for:
        min::paragraph_obj_format               // obj_format

const min::gen_format * min::  compact_value_gen_format  : ........ p159
        // Same as min::compact_gen_format except for:
        min::quote_value_str_format             // str_format

const min::gen_format * min::  compact_id_gen_format  : ............ p159
        // Same as min::compact_gen_format except for:
        min::quote_value_id_str_format          // str_format
        min::compact_id_obj_format              // obj_format

const min::gen_format * min::  id_gen_format  : .................... p160
        // Same as min::compact_gen_format except for:
        min::quote_value_id_str_format          // str_format
        min::id_obj_format                      // obj_format

const min::gen_format * min::  id_map_gen_format  : ................. p160
        // Same as min::compact_gen_format except for:
        min::quote_value_str_format             // str_format
        min::isolated_line_id_obj_format        // obj_format

const min::gen_format * min::  name_gen_format  : ................... p160
        // Same as min::compact_gen_format except for:
        min::quote_value_str_format             // str_format
        min::name_lab_format                    // lab_format

const min::gen_format * min::  leading_always_gen_format  : ....... p160
        // Same as min::compact_gen_format except for:
        min::standard_str_format                // str_format
        min::leading_always_lab_format          // lab_format
```

Printing Numeric General Values:

```
"%.0f"                  // int_printf_format
1e15                    // non_float_bound
"%.15g                  // float_printf_format
NULL                    // fraction_divisors
0                       // fraction_accuracy
```

const min::uns32 * min:: `standard_divisors` :

```
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
16, 32, 64, 128, 256, 512, 1024, 0
```



```
min::printer min::  print_num
                   ( min::printer printer,
                     min::float64 value,
                     const min::num_format *
                           num_format = NULL )
```


Printing String General Values:


```
struct min::  str_format
                   {
                     min::str_classifier   quote_control;
                     min::ustring          str_break_begin;
                     min::ustring          str_break_end;
                     min::quote_format     quote_format;
                     min::uns32            id_strlen;
                   };
```



```
min::standard_str_classifier          // str_classifier
(min::ustring) "\x01\x01" "#"         // str_break_begin
(min::ustring) "\x01\x01" "#"         // str_break_end
min::standard_quote_format            // quote_format
0xFFFFFFFF                            // id_strlen
```



```
// Same as min::standard_str_format except for:
min::quote_separator_str_classifier    // str_classifier
```



```
// Same as min::standard_str_format except for:
min::quote_value_str_classifier // str_classifier
```

```
const min::str_format * min::  quote_all_str_format  : .................. p164
    // Same as min::standard_str_format except for:
    min::quote_all_str_classifier                // str_classifier
    0                                            // id_strlen
```

```
struct min::  quote_format  ........................................... p164
                    {
                      min::ustring    str_prefix;
                      min::ustring    str_postfix;
                      min::ustring    str_postfix_name;
                    };
```

```
const min::quote_format min::  standard_quote_format  : ............ p164
        (min::ustring) "\x01\x01" "\""   // str_prefix
        (min::ustring) "\x01\x01" "\""   // str_postfix
        (min::ustring) "\x03\x03" "<Q>"  // str_postfix_name
```

```
min::printer min::  print_unicode
                     ( min::printer printer,
                       min::unsptr n,
                       min::ptr<const min::Uchar> p,
                       const min::str_format *
                             str_format = NULL ) .................... p164
min::printer min::  print_quoted_unicode
                     ( min::printer printer,
                       min::unsptr n,
                       min::ptr<const min::Uchar> p,
                       const min::str_format * str_format ) ...... p165
min::printer min::  print_breakable_unicode
                     ( min::printer printer,
                       min::unsptr n,
                       min::ptr<const min::Uchar> p,
                       const min::str_format * str_format ) ...... p165
min::printer min::  print_cstring
                     ( min::printer printer,
                       const char * str,
                       const min::str_format *
                             str_format = NULL ) .................... p166
min::printer min::  print_str
                     ( min::printer printer,
                       min:gen str,
                       const min::str_format *
                             str_format = NULL ) .................... p166
```

Printing Label General Values:

```
struct min:: lab_format ............................................. p167
                   {
                     min::pstring         lab_prefix;
                     min::pstring         lab_separator;
                     min::pstring         lab_postfix;
                   };
```

```
const min::lab_format min:: name_lab_format : ..................... p167
   NULL                                    // lab_prefix
   min::space_if_none_pstring              // lab_separator " "
   NULL                                    // lab_postfix
```

```
const min::lab_format min:: leading_always_lab_format : ......... p167
   NULL                                    // lab_prefix
   min::leading_always_pstring             // lab_separator
        // Equivalent to printer << min::leading
   NULL                                    // lab_postfix
```

```
const min::lab_format min:: trailing_always_lab_format : ........ p167
   NULL                                    // lab_prefix
   min::trailing_always_pstring            // lab_separator
        // Equivalent to printer << min::trailing
   NULL                                    // lab_postfix
```

```
const min::lab_format min:: bracket_lab_format : .................. p167
   min::left_square_angle_space_pstring    // lab_prefix    "[< "
   min::space_if_none_pstring              // lab_separator " "
   min::space_angle_right_square_pstring   // lab_postfix   " >]"
```

Printing Special General Values:

```
struct min:: special_format ......................................... p168
                {
                  min::pstring                         special_prefix;
                  min::pstring                         special_postfix;
                  min::packed_vec_ptr<min::ustring>    special_names;
                };
```

```
const min::special_format min::  name_special_format  : ............ p168
   NULL                                      // special_prefix
   NULL                                      // special_postfix
   min::standard_special_names               // special_names


const min::special_format min::  bracket_special_format  : ........ p168
   min::left_square_dollar_space_pstring    // special_prefix   "[$ "
   min::space_dollar_right_square_pstring   // special_postfix  " $]"
   min::standard_special_names              // special_names


min::packed_vec_ptr<min::ustring> min::  standard_special_names  .. p168
   // Names of specials 0x1000000 - 0 .. 0x1000000 - 27 are:
   //      first (does not exist): NULL
   //      next 13 (unused): SPECIAL -1 ... SPECIAL -13
   //      MULTI_VALUED ANY  NONE
   //      MISSING  DISABLED  ENABLED
   //      UNDEFINED  UNUSED  SUCCESS  FAILURE  ERROR
   //      LOGICAL_LINE INDENTED_PARAGRAPH
```

Printing Using An Identifier Map:

```
  min::printer min::  print_id
                        ( min::printer printer,
                          min::gen v ) .............................. p169
   min::id_map min::  set_id_map^S
                        ( <min::printer printer,
                          min::id_map map = min::NULL_STUB ) ...... p170

min::op min::  flush_one_id   ......................................... p170
min::op min::  flush_id_map   ......................................... p170

 min::printer min::  print_one_id  ................................ p170
                      ( min::printer printer,
                        min::id_map id_map = min::NULL_STUB,
                        const min::gen_format * gen_format = NULL )
 min::printer min::  print_id_map  ................................ p170
                      ( min::printer printer,
                        min::id_map id_map = min::NULL_STUB,
                        const min::gen_format * gen_format = NULL )
```

```
min::printer min::  print_mapped_id  ................................ p171
                      ( min::printer printer,
                        min::uns32 ID,
                        min::id_map id_map = min::NULL_STUB,
                        const min::gen_format * gen_format = NULL )
min::printer min::  print_mapped  ................................... p171
                      ( min::printer printer,
                        min::gen v,
                        min::id_map id_map = min::NULL_STUB,
                        const min::gen_format * gen_format = NULL )
```

Printing Using Defined Formats:

```
typedef min::printer ( * min::   defined_format_function   ) ....... p171
                             ( min::printer printer,
                               min::gen v,
                               const min::gen_format * gen_format,
                               min::defined_format defined_format )

struct min::  defined_format_header  ................................. p172
{
                        const min::uns32   control
                        const min::uns32   length
                        const min::uns32   max_length
     const min::defined_format_function   defined_format_function
}

min::defined_format_insptr min::   new_defined_format R
                             ( min::defined_format_function f,
                               min::uns32_number of_arguments ) .... p172

 void min::  map_packed_subtype R
               ( min::uns32 subtype,
                 min::defined_format defined_format ) ............... p172
 void min::  map_type R
               ( min::gen type,
                 min::defined_format defined_format ) ............... p172
```

Object Creation:

Protected Object Vector Level:

```
  min::ref<min::gen>   var
                         ( min::obj_vec_updptr & vp,
                           min::unsptr index ) ........................ p184
  min::ref<min::gen>   hash
                         ( min::obj_vec_updptr & vp,
                           min::unsptr index ) ........................ p184
  min::ref<min::gen>   attr
                         ( min::obj_vec_updptr & vp,
                           min::unsptr index ) ........................ p184
  min::ref<min::gen>   aux
                         ( min::obj_vec_updptr & vp,
                           min::unsptr index ) ........................ p184
  min::ref<min::gen>   operator  [ ]
                         ( min::obj_vec_updptr const & vp,
                           min::unsptr index ) ........................ p184
    min::ptr<min::gen>   operator  +
                           ( min::obj_vec_updptr const & vp,
                             min::unsptr index ) ...................... p184
min::ptr<min::gen> min::  begin_ptr_of
                           ( min::obj_vec_updptr & vp ) ............ p184
min::ptr<min::gen> min::  end_ptr_of
                           ( min::obj_vec_updptr & vp ) ............ p184

  (constructor) min::  obj_vec_insptr   vp ( min::gen v ) ........... p185
  (constructor) min::  obj_vec_insptr   vp ( min::stub * s ) ........ p185
  (constructor) min::  obj_vec_insptr   vp ( void ) ................. p185

                       operator  const min::stub *
                         ( min::obj_vec_insptr const & vp ) ........ p185
min::obj_vec_insptr &   operator  =
                         ( min::obj_vec_insptr & vp,
                           min::gen v ) ............................. p185
min::obj_vec_insptr &   operator  =
                         ( min::obj_vec_insptr & vp,
                           const min::stub * s ) .................... p185

 void min::ref<min::gen> min::  attr_push
                                  ( min::obj_vec_insptr & vp ) ..... p185
 void min::ref<min::gen> min::  aux_push
                                  ( min::obj_vec_insptr & vp ) ..... p185
```

Vector Level Object Maintenance:

Unprotected Object Vector Level:

Object List Level:

Read-Only Attribute Pointer

```
      (constructor) min::  attr_ptr   ap
                             ( min::obj_vec_ptr & vp ) ........... p209
     min::attr_ptr &       operator  =
                             ( min::attr_ptr & ap,
                               min::obj_vec_ptr & vp ) ........... p209
 min::obj_vec_ptr & min::  obj_vec_ptr_of
                             ( min::attr_ptr & ap ) .............. p209
```

Locator Functions:

```
void min::  locate
               ( min::attr_ptr & ap,
                 min::gen name ) ..................................... p210
void min::  locatei   ( min::attr_ptr & ap, int name ) ............. p210
void min::  locatei
               ( min::attr_ptr & ap,
                 min::unsptr name ) .................................. p210
void min::  locate
               ( min::attr_ptr & ap,
                 min::unsptr & length, min::gen name ) .............. p210
void min::  locate_reverse
               ( min::attr_ptr & ap,
                 min::gen reverse_name ) ............................. p210
void min::  relocate   ( min::attr_ptr & ap ) ........................ p210
```

Accessor Functions:

```
 min::unsptr min::  get
                      ( min::gen * out, min::unsptr n,
                        min::attr_ptr & ap ) ....................... p210
    min::gen min::  get
                      ( min::attr_ptr & ap ) ....................... p210
    unsigned min::  get_flags
                      ( min::gen * out, unsigned n,
                        min::attr_ptr & ap ) ....................... p210
        bool min::  test_flag
                      ( min::attr_ptr & ap,
                        unsigned n ) ............................... p210
```

Information Functions:

```
struct min::  attr_info
                {
                  min::gen    name;
                  min::gen    value;
                  min::uns64  flags;
                  min::unsptr value_count;
                  min::unsptr flag_count;
                  min::unsptr reverse_attr_count;
                }; ............................................ p210

struct min::  reverse_attr_info
                {
                  min::gen    name;
                  min::gen    value;
                  min::unsptr value_count;
                }; ............................................ p211

   min::gen min::  name_of
                     ( min::attr_ptr & ap ) ....................... p211
   min::gen min::  reverse_name_of
                     ( min::attr_ptr & ap ) ....................... p211

       bool min::  attr_info_of
                     ( min::attr_info & info,
                       min::attr_ptr & ap,
                       bool include_reverse_attr = true ) ...... p211
       bool min::  reverse_attr_info_of
                     ( min::reverse_attr_info & info,
                     ( min::attr_ptr & ap ) ....................... p211

min::unsptr min::  attr_info_of
                     ( min::attr_info * out, min::unsptr n,
                       min::attr_ptr & ap,
                       bool include_reverse_attr = true,
                       bool include_attr_vec = false ) .......... p211
       void min::  sort_attr_info
                     ( min::attr_info * out, min::unsptr n ) ... p211
min::unsptr min::  attr_info_of
                     ( min::attr_info * out, min::unsptr n,
                       min::obj_vec_ptr & vp,
                       bool include_reverse_attr = true,
                       bool include_attr_vec = false ) .......... p211
```

```
  unsigned min::  get_flags
                    ( min::gen * out, unsigned n,
                      min::attr_updptr & ap ) .................. p217
      bool min::  test_flag
                    ( min::attr_updptr & ap,
                      unsigned n ) .............................. p217
  min::gen min::  update
                    ( min::attr_updptr & ap,
                      min::gen v ) .............................. p217
```

Information Functions:

```
  min::gen min::  name_of
                    ( min::attr_updptr & ap ) .................. p217
  min::gen min::  reverse_name_of
                    ( min::attr_updptr & ap ) .................. p217

      bool min::  attr_info_of
                    ( min::attr_info & info,
                      min::attr_updptr & ap,
                      bool include_reverse_attr = true ) ...... p217
      bool min::  reverse_attr_info_of
                    ( min::reverse_attr_info & info,
                    ( min::attr_updptr & ap ) .................. p217

min::unsptr min::  attr_info_of
                    ( min::attr_info * out, min::unsptr n,
                      min::attr_updptr & ap,
                      bool include_reverse_attr = true,
                      bool include_attr_vec = false ) .......... p217
min::unsptr min::  reverse_attr_info_of
                    ( min::reverse_attr_info * out,
                      min::unsptr n,
                      min::attr_updptr & ap ) .................. p217
```

Insertable Attribute Pointer

```
        (constructor) min::  attr_insptr   ap
                               ( min::obj_vec_insptr & vp ) ....... p218
         min::attr_insptr &    operator  =
                               ( min::attr_insptr & ap,
                                 min::obj_vec_insptr & vp ) ....... p218
 min::obj_vec_insptr & min::  obj_vec_ptr_of
                               ( min::attr_insptr & ap ) ........... p218
```

Locator Functions:

Object Attribute Short-Cuts:

Printing Object General Values:


```
{
  min::uns32                          obj_op_flags;
  const min::gen_format *             element_format;
  const min::gen_format *             top_element_format;
  const min::gen_format *             label_format;
  const min::gen_format *             value_format;
  const min::gen_format *             initiator_format;
  const min::gen_format *             separator_format;
  const min::gen_format *             terminator_format;
  min::str_classifier                 mark_classifier;
  min::pstring                        obj_empty;
  min::pstring                        obj_bra;
  min::pstring                        obj_braend;
  min::pstring                        obj_ketbegin;
  min::pstring                        obj_ket;
  min::pstring                        obj_sep;
  min::pstring                        obj_attrbegin;
  min::pstring                        obj_attrsep;
  min::pstring                        obj_attreol;
  min::pstring                        obj_attreq;
  min::pstring                        obj_attrneg;
  const min::flag_format *            flag_format;
  const min::uns64                    hide_flags;
  min::pstring                        obj_valbegin;
  min::pstring                        obj_valsep;
  min::pstring                        obj_valend;
  min::pstring                        obj_valreq;
```

obj_format.obj_op_flags:

```
struct min::  flag_format  ........................................ p230
          {
            min::pstring                         flag_prefix;
            min::pstring                         flag_postfix;
            min::packed_vec_ptr<min::ustring>    flag_names;
          };

const min::flag_format min::  standard_attr_flag_format  : ........ p230
   min::left_square_leading_always_pstring    // flag_prefix   "["
   min::trailing_always_right_square_pstring  // flag_postfix  "]"
   min::standard_attr_flag_names              // flag_names

min::packed_vec_ptr<min::ustring>
     min::   standard_attr_flag_names  ............................ p230
              // Names of flags numbered 0, 1, 2, ... 63 are:
              //     * + - / @ & # = $ % < > a ... z A ... Z

const unsigned min::  standard_attr_a_flag   = 12 ................. p230
const unsigned min::  standard_attr_A_flag   = 38 ................. p230
const unsigned min::  standard_attr_hide_flag   = 38 + 'H' - 'A' ... p230
 const min::64 min::  standard_attr_hide_flags   =
                        1ull << min::standard_attr_hide_flag ...... p230

struct min::  flag_parser  ........................................ p231
      {
        min::uns32 ( * flag_parser_function )
             ( min::uns32 * flag_numbers,
               char * text_buffer,
               const min::flag_parser * flag_parser );
        // Members beyond this point are used by
        // min::standard_flag_parser.

        const min::uns32 *  flag_map;
        min::uns32          flag_map_length;
      };

   const min::uns32 * min::  standard_attr_flag_map   .............. p231
     const min::uns32 min::  standard_attr_flag_map_length   ...... p231
     const min::uns32 min::  NO_FLAG   = 0xFFFFFFFF ................ p231
    // If min::standard_attr_flag_names[F] == "\x01\x01" "C"
    // then min::standard_attr_flag_map[C] == F
    // else min::standard_attr_flag_map[C] == min::NO_FLAG
```

```
const min::flag_parser min::  standard_attr_flag_parser  : ........ p231
   min::standard_flag_parser           // flag_parser_function
   min::standard_attr_flag_map         // flag_map
   min::standard_attr_flag_map_length  // flag_map_length


const min::obj_format min::  compact_obj_format  : ................. p231
     min::ENABLE_COMPACT                   // obj_op_flags

   min::compact_gen_format                 // element_format
   NULL                                    // top_element_format
   min::compact_value_gen_format           // value_format
   min::name_gen_format                    // label_format

   min::leading_always_gen_format          // initiator_format
   min::trailing_always_gen_format         // separator_format
   min::trailing_always_gen_format         // terminator_format

   min::standard_str_classifier            // mark_classifier

   min::left_curly_right_curly_pstring     // obj_empty

   min::left_curly_leading_pstring         // obj_bra
   min::trailing_vbar_leading_pstring      // obj_braend
   min::trailing_vbar_leading_pstring      // obj_ketbegin
   min::trailing_right_curly_pstring       // obj_ket

   min::space_if_none_pstring              // obj_sep

   min::trailing_always_colon_space_pstring // obj_attrbegin
   min::trailing_always_comma_space_pstring // obj_attrsep

   min::erase_all_space_colon_pstring      // obj_attreol

   min::space_equal_space_pstring          // obj_attreq

   min::no_space_pstring                   // obj_attrneg

   min::standard_attr_flag_format          // flag_format
   min::standard_attr_hide_flags           // hide_flags
```

```
   min::left_curly_star_space_pstring      // obj_valbegin
   min::trailing_always_comma_space_pstring // obj_valsep
   min::space_star_right_curly_pstring     // obj_valend
   min::space_equal_space_pstring          // obj_valreq
```

const min::obj_format min:: line_obj_format : p232

```
   // Same as min::compact_obj_format except for:
     min::ENABLE_COMPACT                   // obj_op_flags
   + min::ENABLE_LOGICAL_LINE
   min::paragraph_gen_format               // top_element_format
```

const min::obj_format min:: paragraph_obj_format : p232

```
   // Same as min::compact_obj_format except for:
     min::ENABLE_COMPACT                   // obj_op_flags
   + min::ENABLE_INDENTED_PARAGRAPH
   min::line_gen_format                    // top_element_format
```

const min::obj_format min:: compact_id_obj_format : p233

```
   // Same as min::compact_obj_format except for:
     min::ENABLE_COMPACT                   // obj_op_flags
   + min::DEFERRED_ID
   min::compact_id_gen_format              // element_format
   min::compact_id_gen_format              // value_format
   NULL                                    // obj_attr...
   NULL                                    // flag_format
   0                                       // hide_flags
   NULL                                    // obj_val...
```

const min::obj_format min:: id_obj_format : p233

```
     min::FORCE_ID                         // obj_op_flags
   // All other elements are NULL or 0
```

const min::obj_format min:: embedded_line_obj_format : p233

```
   // Same as min::compact_obj_format except for:
   min::EMBEDDED_LINE                      // obj_op_flags
   min::null_str_classifier                // mark_classifier
   NULL                                    // obj_empty
```

B Operating System Interface

The interface between MIN implementation code and the operating system consists of standard C++/C functions available on all operating systems, such as `iostreams`, plus **`min::os`** namespace interface functions that are declared in `min_os.h` and defined in `src/min_os.cc`. This code division is intended to make it easy to port MIN to different operating systems by placing all the code that must be changed in the small `src/min_os.cc` file. `min::os` is commonly abbreviated to '**`MOS`**' by including the following definition in code that accesses the interface:

```
#define  MOS  min::os
```

Details of the `min::os` interface are in `min_os.h`. The following is an overview.

B.1 Configuration Parameters

In UNIX the **`MIN_CONFIG`** environment variable value consists of whitespace separated entries of the form '*name*=*value*' that specify configuration parameters, most of which control the Allocator/Collector/Compactor.

B.2 Memory Pools

A memory pool is a contiguous block of pages of <u>virtual</u> memory. Memory pools may be allocated, and it is possible to specify that a pool being allocated has its starting address in a particular range (e.g., so stub addresses can be limited to 44 bits). Segments of memory pools may be freed, may be made into inaccessible virtual memory, and may be made reaccessible. Segments of memory pools may be moved by copying page table entries, which is faster than copying bytes.

C Allocator/Collector/Compactor

The ***Allocator/Collector/Compactor***, or **`acc`**, is a replaceable component of the MIN code, which should not normally be accessed directly. It can be controlled by parameters passed to the program (see Configuration Parameters, p320), or by defaults for these provided at compile time in `min_acc_parameters.h`. Documentation for these parameters is in this last file. The acc code is in the `min::acc` namespace which is abbreviated to '**`MACC`**' by including the following definition in acc code:

```
#define  MACC  min::acc
```

Details of the acc code are in `min_acc.h`. The following is an overview.

TBD

Index