The C_ Programming Dialect

Dedicated to programmer
`Terry Davis`
for the development of
`TempleOS` *and* `HolyC`

Contents

Introduction

The C_ dialect provides a set of abstractions for the C programming language. The purpose of this document is to specify the syntax, constraints, and semantics of various features in C_, without describing the implementation details. Programmers are free to write their own implementation of C_ that conforms to or refines the abstract semantics described in this document. Source codes for the C_ reference implementation and sample examples are available at the repository `<https://github.com/cHaR-shinigami/c_>`, released under the terms of GNU Lesser General Public License (LGPL 2.1 or later), without any warranty of merchantability or fitness for some purpose.

The reference implementation of C_ is a proof of concept that is intended to conform with ISO/IEC 9899:2024, which is the current revision of the C standard; it is commonly known to programmers as C23, and this is the name we shall use throughout the rest of this document. Some features of C_ are implemented using non-standard extensions, which are compiler-specific, and therefore, not fully portable; these have been marked with an asterisk (*) when they are introduced in later chapters. Other implementations of C_ can provide these features in a standard-conforming way, and can also give well-defined behavior to constructs beyond the scope of this document.

Design of the reference implementation is based on a header-only architecture, making it open source by nature; it does not require a separate front-end for C compilers or installation of any additional software. At a high level, the reference implementation is essentially a preprocessing-based transpiler which converts C_ code to C code; the idea is similar to Cfront, the original translator for "*C with Classes*" (later renamed to C++), which transpiled the source code to C code that could be compiled with a native C compiler. Since it is possible to emulate almost the entirety of C_ using C (C23), we do not refer to C_ as a new programming language, but consider it a dialect of C.

The intent of documenting C_ in terms of its abstract semantics is two-fold: firstly, readers need not get bogged down to implementation details, which can often be an unnecessary distraction. The second reason is more important: this approach isolates the abstract behavior from any particular concrete implementation, which allows a future scope of providing more efficient translators for C_. We shall digress a little to the development timeline: C_ started out as a modest collection of macros, and as it started to mature, the necessity of a formal documentation was realized, which commenced after finalizing the reference implementation. Any serious discrepancy between the reference implementation and the contents of this document is unintentional, and may be considered as a "bug".

C_ was created with an aim to simplify programming, while also making it harder for things to go wrong; the latter is a more ambitious goal, and the extent of its fulfillment largely rests on the programmer. C_ facilitates "abstraction-oriented programming", which blends several concepts from functional programming and object-oriented programming paradigms. Many features of C_ are based on similar constructs found in other programming languages, such as C++, Python, Java, which are themselves influenced by C (either directly or indirectly); the semantics of `defer_` have been borrowed from Zig. Proper use of the right abstractions reduces the chances of bugs.

A knowledge of basic concepts and standard terminology in C is a prerequisite, and this document assumes familiarity with the general concepts of programming. The current C standard is quite sophisticated in its entirety, and readers are not required to have a complete mastery over all its intricacies; a deep understanding of C is surely beneficial to understand how the reference implementation works, but that is not necessary to get started with C_.

0.1 Example

Let us start our journey with a non-trivial example that illustrates some of the basic features of C_. The following program reads a list of prices and discounts (in %), and then computes the final price after applying discounts.

```
#include <c._>

Int_ main(Int argc, Ptr(Char_) argv[const])
begin
    guard_(argc == 2)
    Auto_ count_ = 0U;
    guard_(input__(argv[1], count_) == 1)
    Var prices = new__(Float_ [count_]);
    guard_(prices)
    Var discounts = new__(Float_ [count_]);
    guard_(discounts)
    loop_(0, count_ - 1)
        print_("Enter price and discount (in %) for item", _i_ + 1);
        guard_(scan__((*prices)[_i_], (*discounts)[_i_]) == 2, 1)
    end
    Auto_ price_ = 0.f;
    op_(price_, +, prices)
    print_("Total price is", price_);
    op_(discounts, *, prices)
    op_(discounts, /, 100)
    loop_(0, count_ - 1)
        print_("Discount on item", _i_ + 1, "is", (*discounts)[_i_]);
    end
    Auto_ discount_ = .0f;
    op_(discount_, +, discounts)
    print_("Total discount is", discount_);
    print_("Final price is", price_ - discount_);
    print_("Have a nice day");
end
```

The first line `#include <c._>` includes the contents of a header file named `c._`, which itself aggregates and configures several macros, enumeration constants, type definitions, and inline functions, from other header files organized in a multilevel hierarchy. Type names in C_ start with an uppercase letter, and are modifiable if and only if the name ends with an underscore; for example, `Int argc` means the parameter `argc` is non-modifiable.

Recall that in C, a parameter of array type is adjusted to pointer type, so `Ptr(Char_) argv[const]` means "non-modifiable pointer to a pointer to `Char_`". Note that `Char_` means that the dereferenced lvalue is modifiable, and `Ptr(Char_)` means that the pointer itself is non-modifiable. Readers may intuit that `Ptr_(Char_)` makes the pointer modifiable as well, whereas `Ptr_(Char)` means the pointer is modifiable, but the dereferenced lvalue is not.

Blocks or compound statements are started with `begin` and completed with `end`, which offer the benefit of early exit: if some condition is (un)met, one may skip rest of the code within the innermost block that supports early exit, and continue execution just after the end of that block. This feature is useful to flatten out "arrow-shaped code", which simplifies the overall design for the programmer, and reduces cognitive burden for the reader: it can be cumbersome to remember and match braces in a deeply nested code to figure out where each nested block ends.

This is exemplified in the next line `guard_(argc == 2)`, which is a guard clause that says exactly two command-line arguments should be passed at runtime; if the check fails, it skips rest of the code. Notice the absence of a semicolon at the end of the guard clause: this is because `guard_` is syntactically not an expression, but a statement, so a semicolon is redundant and only serves as null statement. The definition `Auto_ count_ = 0U;` creates a modifiable counter variable initialized to zero, whose type is inferred as `ULong_` due to the initializer `0U` (`ULong_` is a synonym for modifiable `unsigned long`; `ULong` is similar but non-modifiable).

Recall that the first command-line argument is conventionally the name by which the executable is invoked, which is available to the program as the first array element `argv[0]`. For this example, we need to pass the number of items as the second argument, which is available as the next element `argv[1]`. The call `input__(argv[1], count_)` reads this string for a valid unsigned number, and on success, assigns that value to `count_`. It returns the number of input items scanned and assigned, which should be 1 in our example, so this check has been expressed as a guard clause, which fails if the second command-line argument does not start with a valid unsigned number of items.

The call `new__(Float_ [count_])` allocates a modifiable array of `float` in the heap memory segment, having a capacity of `count_` elements. On success, it returns pointer to a `Float_` array of length `count_`, which is inferred as the type of the non-modifiable variable `prices`. Note that we could have used `Auto` instead of `Var`; the minor distinction between these two is explained in the next chapter. We know that memory allocation can fail at runtime, in which case `new__` returns a null pointer; we have enforced this check with the guard statement `guard_(prices)`.

The next two lines do a similar allocation and check for `discounts`, whose type is also `Ptr(Float_ [count_])`, which is read as "non-modifiable pointer to a modifiable array of `Float_`, having a capacity of `count_` elements". The primary advantage of using the type "pointer to array" is that a complete array type encodes length information, which is utilized later in the program. Recall that the lifetime of memory objects allocated on the heap is managed by the programmer; however, for the sake of brevity, we have omitted an explicit deallocation of `prices` and `discounts` by calling `free` (for programs executing in a hosted environment, the operating system typically takes care of deallocating the heap memory once the process terminates or aborts, among other cleanup activities).

`loop_(0, count_ - 1)` iterates from zero through `count_ - 1`, and the current iteration number is stored in the modifiable "candle" variable `_i_`, whose scope and lifetime are limited to the loop block till `end`. In each iteration, a prompt message is displayed before reading the values for price and discount entered by the user. Note that these variables are of type "pointer to array", so we first obtain the array by dereferencing the pointers, and then access the element at index `_i_`. The parentheses are necessary around the dereference due to its lower precedence than the array subscript operator; as an alternative, it also can be written as `prices[0][_i_]` or `discounts[0][_i_]`.

The `print_` syntax is modeled after the `print` function of Python 3, which prints a space between arguments and also appends a trailing newline. `scan__` reads from `stdin` (the standard input stream), and assigns the values to its arguments in the same order. Similar to `input__`, `scan__` also returns the number of input items scanned and assigned, which can be less than the number of arguments if the user enters an invalid value outside the domain of an argument's type, or if the input stream is exhausted and a subsequent read returns `EOF` (end of file).

Notice that the `guard_` statement syntax is different from before: this time there is an additional argument `1` just after the guard clause. This is because `guard_` is actually a "polymorphic" statement, which can accept either one or two arguments. In the first form, if the guard clause fails, then rest of the statements are skipped. However, if we use this form within a `loop_` block, then it will only skip the remaining iterations if the guard clause fails, and continue execution after the loop. However, this is not desirable: if a value cannot be read, the program should not proceed any further. In the second form of the `guard_` statement, if the clause is unsatisfied, then the function returns the value specified by the additional argument. We have used `1` in this example, because the value returned by `main` is also exit status of the process, which is conventionally zero on success and non-zero to indicate failure.

After reading the prices and discounts, we define a variable `price_` to aggregate the total price; the initializer `0.f` makes its type `Float_`. The statement `op_(price_, +, prices)` adds each price of the `prices` array one by one to the variable `price_`; it was initialized to zero earlier, so its result is the sum of all prices, which is printed.

`op_` is a versatile statement that unifies a rich variety of operations with both arrays and non-arrays under a common syntax. The statement `op_(discounts, *, prices)` multiplies each element of the `discounts` array with the corresponding element in the `prices` array. The results are stored in the `discounts` array, but as the discount values are entered as percentage, we also need to divide each discount by 100: this is done by the next statement `op_(discounts, /, 100)`. `op_` is modeled after `op__`; their precise semantics are described in chapters 4 and 5.

Next we run a loop to print the discount amount applied to each item. This is followed by storing the total discount amount in the variable `discount_`, which is done by the statement `op_(discount_, +, discounts)`. We print this amount and also the final price after subtracting the total discount. Since the C99 revision of ISO C, reaching the end of `main` returns zero to the environment, so an explicit `return 0;` statement has been omitted.

0.2 Compilation

To compile our first example, we need a concrete implementation of C_. The reference implementation can be freely downloaded from the repository `https://github.com/cHaR-shinigami/c_`. It contains a directory `examples/`, which can be considered as a project directory that contains source codes of examples discussed in this document. It contains three sub-directories: `.include/`, `include/`, and `compile/`. `.include/` contains header files which form the core implementation of C_; these files need not be modified by most programmers and their contents are likely to change over time, mostly for bug fixes and occasionally to accommodate new features in existing headers. The directory `include/` contains the header `<c._>`, which we saw in the previous example, along with several other header files that will be discussed in later chapters. Files within the directory `compile/` are given to the compiler as translation units. Both `include/` and `compile/` files are meant to be modified by programmers. By convention, C_ header files have `._` as filename extension; C_ translation units are analogous to `.c` files, and have `.c_` as extension.

0.2.1 Requirements

The `compile/` directory has a file named `discount.c_`, which contains the previous example. It has another file named `lib.c_`, which provides external definitions for several helper functions. These definitions are required by the linker, so instead of recompiling this file `lib.c_` every time, we can generate an object file to be linked later. The project directory `examples/` has a shell script named `build.sh`, which automates this task and also generates object files for all the translation units in the `compile/` directory. This script is intended for POSIX-compatible shells (such as `bash`), and needs to be executed only once. The commands of this script are discussed in appendix E, based on which similar scripts can be written for other environments (such as classic Windows `cmd` or `PowerShell`).

The reference implementation requires a compiler that supports C23. Full C23 support is not necessary; only a few improved language rules and new features are required to emulate the behavior of C_ using C. The precise set of requirements can be found in the companion document that explains the full architecture of the reference implementation; many of these dependencies have been available for a long time, so these are not entirely new, but "prior art" which may be available in compilers that do not fully support C23 yet.

The source codes presented in this document have been tested with two compilers installed on a 32-bit variant of Ubuntu operating system: `gcc` (version 14) and `clang` (version 19), whose precompiled binaries for i386 architecture (IA-32) were downloaded from the official software repository of Ubuntu. The most important task is to make sure that our compiler is able to locate the header files; both `gcc` and `clang` have similar options for this. For demonstration, if we assume that `examples/` is present in the home directory, then the following can be used:

```
gcc/clang -xc -std=c23 -iprefix "$HOME"/examples/.include
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library
-iprefix "$HOME"/examples/include -iwithprefix/.
```

`gcc/clang` needs to be at least `gcc-14` or `clang-19`. The option `-xc` is required due to the filename extension `.c_`: it tells the complier to consider them as C files; `-std=c23` is required just in case the default language version is set to an older C standard. `-iprefix "$HOME"/examples/.include` sets the path relative to which the subsequent `-iwithprefix` locations are considered, up to the next `-iprefix` which changes the relative path to `include/` directory. The sub-directories within `.include/` collectively contain nearly a hundred header files that are logically structured into multilevel hierarchies. As a small note, `bash` supports "tilde expansion" for the home directory, so `"$HOME"` can be replaced with the more terse ∼ symbol.

With that, we're all set to compile and execute our first C_ program. Later in this chapter, we shall add several more compilation options to enable rigorous diagnostics; however, the overall command becomes rather clunky when typed out in its entirety. A much cleaner workaround is to set a "command alias", which is done in `bash` as:

```
alias cc_="gcc -xc -std=c23 -iprefix '$HOME'/examples/.include \
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library \
-iprefix '$HOME'/examples/include -iwithprefix/."
```

Note that the trailing backslash in first two lines is meant for line splicing. Most modern shells will have some mechanism for setting a command alias. By default, `bash` tries to read the contents of `"$HOME"/.bash_aliases` (if the file exists), so a better approach is to put our alias in that file (create the file if needed); it comes into effect at the next invocation of `bash`, or by "sourcing" it as `. "$HOME"/.bash_aliases` in the current invocation itself. Finally, if the `examples/` directory is put elsewhere, then users will have to replace `"$HOME"` with the correct path.

0.2.2 Execution

We are now at a point to see some tangible outcome: after changing the current directory to `examples/compile/`, compile as `cc_ lib.c_ discount.c_` (sequence doesn't matter), and assuming things go well, run the file `a.out`.

```
$ cc_ discount.c_ lib.c_
$ ./a.out 2
Enter price and discount (in %) for item 1
100 10
Enter price and discount (in %) for item 2
200 20
Total price is 300
Discount on item 1 is 10
Discount on item 2 is 40
Total discount is 50
Final price is 250
Have a nice day
$
```

We had set the `cc_` alias to use `gcc` as the compiler command. For `clang` (version 19 or later), the current set of options generates quite a few warnings; these are harmless, and can be suppressed with an additional option `-Wno-pointer-arith`, so a minimal `cc_` alias with `clang` as the compiler would look like:

```
alias cc_="clang -xc -std=c23 -iprefix '$HOME'/examples/.include \
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library \
-iprefix '$HOME'/examples/include -iwithprefix/. -Wno-pointer-arith"
```

Recall a brief mention of the shell script `build.sh`; it creates a directory `object/` within `examples/` for storing precompiled object files that can be linked later. We can link the binary `object/lib.o` which is generated by compiling `lib.c_`; as an alternative to running the build script, we can also manually compile `lib.c_` with the compile-only option `-c`, which produces an object file named `lib.o` instead of the executable `a.out`. Now we can modify the alias as shown below (`gcc` can be replaced by `clang`, in conjunction with `-Wno-pointer-arith`).

```
alias cc_="gcc -std=c23 -iprefix '$HOME'/examples/.include \
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library \
-iprefix '$HOME'/examples/include -iwithprefix/. \
'$HOME'/examples/object/lib.o -xc"
```

Now we don't need to recompile `lib.c_` for testing every example, as we can just write `cc_ discount.c_`.

0.3 Diagnostics

Modern software development is incomplete without enabling compile-time diagnostics of suspicious code constructs, which are usually reported as "compiler warnings". Most professional C developers will use a customized set of options aligned with the coding standards and business requirements of their production environment. For compiling examples in later chapters, we shall enable several warning options that are considered suitable for most purposes.

NOTE The reference implementation provides the starred features (*) using non-standard extensions that need to be compiled with `cc_ -Wno-pedantic` (this option disables `-Wpedantic` which is part of our `cc_` command alias).

0.3.1 gcc

The following options have been used with `gcc` to test the reference implementation and all sample examples in this document; same set of options is also used by the build script to generate object files within `object/` directory.

```
alias cc_="gcc -std=c23 -O3 -s -ftrack-macro-expansion=0 \
-iprefix '$HOME'/examples/.include \
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library \
-iprefix '$HOME'/examples/include \
-iwithprefix/. -iwithprefix/class -iwithprefix/interface \
-Werror -Wall -Wextra -Wpedantic \
-Wcast-align  -Wcast-qual -Wswitch-enum -Wwrite-strings \
-Wduplicated-branches     -Winit-self -Wshift-overflow=2 \
-Wduplicated-cond  -Wnull-dereference -Wstrict-overflow=2 \
-Wno-override-init -Wno-missing-field-initializers \
-Wno-parentheses   -Wno-tautological-compare -Wno-type-limits \
'$HOME'/examples/object/lib.o -xc"
```

`-O3` is an optimization flag that we have chosen not just to improve runtime efficiency, but also because some diagnostic options come into effect only when certain optimizations are enabled in conjunction with warning flags. This ensures a more rigorous diagnosis of our examples, which is all the more crucial for something as new as C_. `-s` is used to reduce size of the executable by stripping non-essential data pertaining to symbol table and relocation of object code; this option is not suitable for use with debugging tools. It should also not be used in conjunction with the `-c` option, as it removes crucial information required by the linker, making the object file(s) unlinkable. Header files placed within the `class/` and `interface/` directories will be required later in chapters 7 and 8.

The reference implementation relies heavily on macros; in fact, it is essentially a preprocessing-based transpiler, and a simple typographic error in a small code can trigger an avalanche of error messages, which can be overwhelming for beginners, and mildly annoying for seasoned programmers. The option `-ftrack-macro-expansion=0` limits each error message to the macro that is used in the program, skipping any nested expansion of helper macros that actually produce the erroneous code. Not using this option is mostly useful for debugging expansions of low-level iterated composition and metaprogramming macros from the ellipsis framework, all of which is introduced in chapter 4.

`-Werror` turns all warnings into errors, excluding any exceptions specified with the `-Wno-error=` option. This sounds reasonable in a textbook setting, but may not always be suitable in a production environment where certain warnings are tolerable, but should not be outright disabled as they are informative to be programmer; for example, use of non-portable features may be acceptable with a particular compiler, but it should be reported as a warning. To quickly get back to the main agenda, we shall skip a thorough explanation of specific warning options; these are summarized in appendix E. The last few options starting with `-Wno-` disable the specified warning; these warn about suspected bugs, but are otherwise perfectly legal in ISO C, and false positives for the reference implementation.

0.3.2 clang

Most options of `clang` are similar to those of `gcc`; for our purpose, we shall use the following set of options.

```
alias cc_="clang -std=c23 -O3 -s -fmacro-backtrace-limit=1 \
-iprefix '$HOME'/examples/.include \
-iwithprefix/ellipsis -iwithprefix/dialect -iwithprefix/library \
-iprefix '$HOME'/examples/include \
-iwithprefix/. -iwithprefix/class -iwithprefix/interface \
-Werror -Wall -Wextra -Wpedantic \
-Wcast-align  -Wcast-qual -Wswitch-enum -Wwrite-strings \
-Wassign-enum -Wshift-sign-overflow -Wunreachable-code-aggressive \
-Wno-override-init -Wno-missing-field-initializers \
-Wno-pointer-arith -Wno-unused-command-line-argument \
'$HOME'/examples/object/lib.o -xc"
```

The setting `-fmacro-backtrace-limit=1` works similar to the `gcc` option `-ftrack-macro-expansion=0`. As before, a few warnings need to be disabled: in particular, the exception `-Wno-error=unused-command-line-argument` allows us to look at preprocessed code with the `-E` option, which does not require linker, so `lib.o` remains unused.

0.4 Debugging

C_ offers out of the box debugging support without the use of any additional software. Programs can be compiled in debugging mode simply by defining the macro `DEBUG`; this setting can be applicable to a whole project, or even toggled selectively for specific source files. Several C_ headers are configurable, which means that the features implemented by them behave differently depending on whether the macro `DEBUG` was defined at the time the header was included; more precisely, these headers are re-configurable, so if we change the `defined` state of the `DEBUG` macro and then include the header again, it will be automatically re-configured. For standard headers of C_, the current configuration is internally recorded in an object-like macro, whose name is same as the header name entirely in uppercase, and the dot replaced by an underscore; as an example, `<assert._>` is a re-configurable header whose current configuration is given by the macro `ASSERT__`. Such a macro shall expand to either 0 or 1, where 0 indicates that the `DEBUG` macro was not defined when the header was last included, so debugging is disabled; whereas 1 means the latest inclusion of that header was preceded by an active definition of `DEBUG`, so debugging is currently enabled.

`<c._>` is a top-level header that needs to be included in C_ source files (either directly or via another header). Recall that this file is placed in `include/` directory, whose contents are meant to be updated by programmers as per their needs. As this file will be our companion through the rest of this journey, its worth taking a look inside.

```
//
#ifndef C__

#define  DEBUG
#undef  NDEBUG
#define TODO

/*/
#ifndef C__

#undef   DEBUG
#define NDEBUG
#undef  TODO

/**/

#define C__

#define LOGGER

#include <class._>
#include <defer._>
#include <environment._>
#include <interface._>

#endif

#include <array._>
#include <bits._>
#include <call._>
#include <io._>
#include <iterators._>
#include <logger._>
#include <method._>
#include <pointer._>
#include <range._>
#include <shift._>
```

Besides aggregating contents of several other headers, `<c._>` also sets a global configuration for the entire project, which can changed locally within a source file, either for specific headers or the entirety of C_. The uncommented configuration is for "debugging mode"; any replacement text of the `DEBUG` macro is ignored. We also define another macro `TODO`, whose purpose is quite trivial: it is used to mark incomplete sections of code, such as temporary stubs (`TODO` is often highlighted by code editors); the identifier `TODO` should not be used to declare any lexical element.

The section after `/*/` configures "production mode", where debugging is disabled and (non-comment) occurrences of `TODO` are not erased by the preprocessor, causing compilation errors. Debugging mode increases code size and the program incurs some runtime overhead, which can cause a noticeable difference in performance for large projects. Once a codebase has been thoroughly tested, it may be desirable to disable "defensive" checks. `NDEBUG` affects the `assert` macro on a subsequent inclusion of `<assert.h>`; this should be used with caution, as the expression passed to `assert` is not evaluated, which can cause unexpected "Heisenbugs" due to unevaluated side effects.

The headers included before `#endif` are non-configurable; those included afterwards are re-configurable. When a newly created module is used with a well tested project, it may be desirable to enable debugging only for that module, while compiling rest of the project in production mode. As global configurations are within header guard, they are effective only the first time `<c._>` is included, so debugging can be enforced locally by including `<c._>` twice.

```
#include <c._>
#define DEBUG
#include <c._>
```

It may also be desirable to enforce debugging for certain features of C_, irrespective of the global configuration in `<c._>`. This can be done by re-including only the specific header(s) that provide these features; for instance:

```
#include <c._>
#define DEBUG
#include <array._>
```

The above example shows how one can locally enforce debugging for features provided by the `<array._>` header. Note that this approach re-configures only the selected headers, not their dependencies. For example, `<array._>` depends on some non-configurable components provided by `<pointer._>`; the latter is also a re-configurable header, but its existing configuration is preserved, and to change that one needs to explicitly include it after defining `DEBUG`. Similarly, one can also selectively disable debugging: simply undefine `DEBUG` and include the appropriate headers.

The `LOGGER` macro enables logging facilities provided by `<logger._>`; we have chosen to always enable logging, so it is kept outside the mode configuration. Note that `<c._>` configures several headers, but does not provide any debuggable feature by itself. As such, the macro `C__` does not record the `defined` state of `DEBUG` to introspect the current configuration; it would not be of much use because the actual providers can be individually re-configured.

The extent of debugging support depends largely on the feature under consideration; the basic checks are for common sources of bugs, such as null pointer dereference or accessing an array out of bounds. A major feature of debugging mode is that for methods (discussed in chapter 6), the `call__` expression invokes protocols, which are used to specify pre-conditions and post-conditions. A protocol can be bypassed if and only if both the "provider" and the "consumer" (caller) agree that debugging is not required, i.e. they are both compiled in production mode. In other words, if either of them enables debugging, then the transformation logic (process) is invoked via protocol.

Finally, one important observation is that the global configuration can be altered simply by changing the second character of the first line; the mechanism relies on a preprocessing rule: comments do not nest. In other words, a comment started by `/*` is ended by the first occurrence of `*/`, disregarding any instance of `/*` within the comment. If we change the second character inside `<c._>` from `/` to `*`, then the first line becomes `/*`, starting a comment that ends at the line `/*/`, and what follows is the uncommented configuration for production mode. With `//` as the first line, the line `/*/` acts differently: instead of ending an ongoing comment, it actually starts a new comment, which consumes the production mode configuration and ends at the line `/**/`. In summary, the global configuration can be toggled with minimum effort just by changing the second character in the file: `/` means debugging mode; `*` means production mode. Additional macros can be defined in both configuration segments as per project requirements.

0.5 Conventions

The reference implementation operates with "pseudo-namespaces" that use name mangling to emulate the behavior of proper namespaces: members in different namespaces can be designated by the same identifier. For a proper functioning of these so-called pseudo-namespaces, the reference implementation "trusts the programmer" to abide by a few simple naming restrictions, in addition to the existing identifier reservations imposed by the C standard.

Firstly, macros defined by the reference implementation shall not remain undefined, and the programmer shall not provide a non-equivalent redefinition of these macros. Secondly, non-macro names defined in `C_` shall not be defined as non-trivial object-like macros; however, it is permitted to define them as trivial object-like macros, or even function-like macros. A trivial object-like macro is one whose expansion does not alter the source code. In other words, an object-like macro is trivial if its replacement text expands to the macro name itself; for example:

```
#define identifier identifier
```

One advantage of function-like macros is that their expansion can be suppressed, if the first non-whitespace character after an occurrence of the macro name is not the opening parenthesis; for example, standard library functions may be additionally provided as macros, and a common trick to suppress macro expansion and call the actual function is to parenthesize the name, such as the call `(putc)('\n')`. On the contrary, the expansion of an object-like macro cannot be suppressed, which can be problematic if an existing non-macro identifier is defined as a non-trivial object-like macro. These restrictions are not specific to `C_`, but applicable to C in general; for instance, the standard header `<stdlib.h>` defines the structure type `div_t` with two members `quot` and `rem`, so none of these names can be defined as non-trivial object-like macros; however, the following macro definitions are unlikely to cause any problem:

```
#define quot quot
#define rem(...)
```

We can also define the notion of a trivial function-like macro in a similar fashion: a function-like macro is considered to be trivial is its expansion is successful and does not modify any text of the source code; for example:

```
#define abs(...) abs(__VA_ARGS__)
```

Despite the simplicity of its definition, there are certain caveats in the design of trivial function-like macros. `abs` is a standard library function declared in the header `<stdlib.h>`: it takes a single argument of type `int` (`Int_` in `C_`) and returns its absolute value. A naive implementation of a corresponding function-like macro might look like:

```
#define abs(n) abs(n)
```

However, the above macro is actually non-trivial; in the absence of this macro, the call `abs(*(int []){0,})` works correctly, but the preprocessor does not understand compound literals, so a macro invocation actually reads it as two arguments: `*(int []){0` and `}`. This may be a contrived example, but is nonetheless perfectly legal C code that is rendered invalid by our second definition of `abs` with a single parameter, making the macro non-trivial.

Note that our definition of "triviality" only requires that the resulting expansion of the macro should not alter the source text; however, it does not imply that a trivial macro can be freely undefined, which can impact a subsequent `#if` preprocessing directive that checks whether a macro name is defined or not. Another naming restriction of `C_` is that user-defined identifiers should not have two consecutive underscores anywhere; this is to avoid name collisions with several internal identifiers with double underscores that are formed as a result of name mangling.

0.5.1 Naming

The underscore character plays a pivotal role in the naming scheme designed for this dialect, which justifies the name C_; it is believed that liberal use of trailing underscores in C_ also reduces the chances of name collision with identifiers in existing libraries. We are already familiar with a couple of rules from the first example: a type name ending with an underscore means that it is modifiable. Readers may have already observed that the same convention is also followed for identifiers of variables: they do not start with an uppercase letter, and a trailing underscore means the object is modifiable. Furthermore, type names begin with an uppercase letter and contain at least one lowercase letter; the latter requirement is to differentiate it from object-like macros, which are traditionally named entirely in uppercase. Except for a few keyword-like features, C_ continues the same tradition for naming object-like macros, with an additional requirement: except for header guards, they should not end with an underscore.

Names of function-like macros end with underscore to differentiate them from function names; the latter should not end with an underscore. One design principle of C_ is that a function-like macro, whose documented semantics are required to be similar to those of a proper function call, should mimic the behavior of an actual function as closely as possible. However, there is a fundamental limitation that cannot be overcome: a macro does not have an address, so it is impossible to obtain a function pointer for a macro. A trailing underscore acts as a form of self-documentation for such limitations: it makes the programmer aware that instead of an actual function, some other form of implementation may be used, most likely a function-like macro. Such names will be encountered frequently in later chapters, as most features of C_ are provided as function-like macros by the reference implementation.

The abstract semantics of C_ does not mandate that its features need to be implemented with macros; thus implementors are not bound to rely on the preprocessor, and they can write a separate front-end pass that transpiles C_ code to C code. For the latter approach, it is strongly recommended that the names for non-macro implementations should additionally be defined as trivial object-like macros; it is hoped that any code that works correctly with the reference implementation should also behave identically when compiled with other implementations of C_.

One undesirable aspect of function-like macros is that the replacement text can expand some argument more than once; this can lead to unfortunate outcomes if the argument contains side effects and more than one expansion gets evaluated. C_ follows the convention of naming such macros with two trailing underscores, which signifies that the implementation is permitted, but not obliged, to evaluate some argument multiple times. Thus any C_ feature whose name ends with two underscores is permitted to be implemented as a function-like macro that may possibly evaluate some argument more than once; the documentation clearly specifies which exact argument(s) may undergo multiple evaluation, and under what circumstances. For some features, multiple evaluation of arguments can itself be part of the semantics; for example, the macro `repeat__(frequency, ...)` from the ellipsis framework replicates the argument sequence given by `__VA_ARGS__`, for the number of times specified by the `frequency` argument.

On the contrary, a feature whose name ends with a single underscore has more stringent requirements: it can be implemented as a function-like macro, but it is not permitted to evaluate any argument more than once. However, a macro implementation is allowed to expand an argument more than once, as long as at most one such expansion is evaluated at runtime; one example is the use of a macro argument in multiple associations of a `_Generic` selection. The phrase "at most" implies that such macros may ignore some argument or perhaps expand it in a context where it does not get evaluated under certain conditions. As before, there may be cases where this behavior is required by the semantics; for example, the conditional expression `test_(`*check*`,` *yes*`,` *no*`)` expands all three arguments, but only one of the sub-expressions *yes* or *no* gets evaluated, depending on whether the scalar expression *check* is true (non-zero) or false (zero). Recall that in C, a "scalar expression" is of arithmetic type or pointer type; equivalently, it is an expression that can be assigned to or cast as boolean type (`_Bool` in C is `Bool_` in C_).

A function-like macro whose name does not have any lowercase letter and ends with a single underscore indicates that a valid invocation expands to a list of constants. The expanded list can be a singleton; for example, the macro `COUNT_(...)` from the ellipsis framework counts the number of arguments passed to it (more precisely, it returns one more than the number of unparenthesized commas in the expanded argument sequence).

Header guards are defined as object-like macros having the same name as the filename entirely in uppercase, with the dot being replaced by an underscore; this assumes that header filenames should not start with a digit and not contain any non-identifier characters. We have already encountered one such example before: `C__` acts as the guard for the header <c._>. Another example is the macro `ASSERT__`, which records whether the header <assert._> was last configured in debugging mode; in addition to that, it also doubles as the header guard. The same practice has been adopted for other configurable headers; for a header file named `header._`, an object-like macro `HEADER__` serves a dual purpose: besides acting as a header guard to avoid redefinitions, it also records the `defined` state of `DEBUG` macro every time `header._` is included. One common use of this convention is to check whether a header has already been included or not, which helps to avoid any unintended re-configuration of dependency headers.

Throughout this document, we have used <angle bracket> syntax for including the configuration header <c._> and C_ standard headers; reference implementation of the standard headers are placed within `.include/` directory. The example headers used for illustrative purposes are not part of C_; these have been included with the "double quotes" syntax, to differentiate them from standard headers that are to be provided by all implementations of C_.

0.5.2 Reservations

In addition to identifiers reserved for the implementation of C itself, names ending with `_C` or `_c` followed by any number of underscores (possibly none) are reserved for use by the implementation of C_ dialect; header filenames ending with `_c._` are reserved as well. Also, any name starting with an underscore is reserved for use as internal identifiers. These names should not be used for declarations even in block scope; one such example is `_site`, which is a non-modifiable parameter of protocols that provides call-site identification details for diagnostic purposes.

0.5.3 Spacing

The source codes presented in this document follow a certain style of spacing between operands and operators: if an operand is placed between two operators, then we put the operand "closer" to the operator that uses this operand; for example, the evenly spaced expression `a + b + c` is written as `a+b + c` to emphasize left associativity of addition. Similarly, we would prefer writing `a * b + c` as `a*b + c` to indicate that multiplication happens first.

Conversely, if an operator is placed between two operands, we put the operator closer to the operand that is evaluated first; an equal spacing on either side indicates that the evaluation of operands is unsequenced, so there is no specific order of evaluation. For example, `l & r` indicates that either operand may be evaluated first, but `p&& q` emphasizes the "short-circuit" behavior of logical operators: in this case, the left expression `p` is evaluated first; `q` is evaluated if and only if `p` turns out to be zero. Similarly, we prefer uneven spacing for a conditional expression `c? y : n` or a comma expression `(x, y)`; the latter aligns with the conventional writing style, but note that other uses of comma, such as for separating function call arguments, do not specify any particular order of evaluation.

It is worth stressing that judicious spacing only clarifies that the programmer is well aware of the precedence and associativity rules of operators; it has no impact on the meaning of the expression, and explicit parentheses can often be a cleaner alternative. As a third alternative, sometimes the use of a different operator may convey the intent more clearly; for example, `*ptr_++` may be confusing to beginners: which one gets incremented, the modifiable pointer `ptr_`, or the dereferenced lvalue? Experienced developers will be aware of right associativity for unary operators, but a simple space should make things somewhat less confusing: `* ptr_++`. Still, the subscript operator can make things more readable; the equivalent `ptr_++[0]` clearly indicates that the pointer gets incremented. Unlike the previous naming conventions, the unusual use of spaces is more of an idiosyncrasy that is not imposed upon programmers, and ritually observing this practice can lead to awkward gaps if lots of operators are involved; while a recommended practice is to break such large expressions into smaller sub-expressions, it is hoped that readers will be forbearing if they perceive that this advice may not have been diligently followed in some of the examples.

0.6 Polymorphism

C_ is strongly influenced by functional programming and object-oriented programming languages; a common feature of the latter is polymorphism, which means “many forms”. Several function-like features of C_ are polymorphic in nature, which means their behavior depends on the number of arguments passed, and sometimes on the type of arguments as well; most often, additional arguments extend the basic functionality by allowing for more fine-grained specification. We are already familiar with one such feature: the `guard_` statement discussed in the first example.

Polymorphic features follow a particular naming convention: if the argument sequence to a polymorphic feature named `multiform` expands to `n` arguments (equivalently, `n - 1` unparenthesized commas), then a feature named `multiform_n` gets invoked; some features may also provide an additional form named `multiform_0` that is used if `multiform_n` is not defined. In our first example, `guard_(prices)` is resolved to `guard_1_(prices)`, whereas `guard_(prices, 0)` would be equivalent to `guard_2_(prices, 0)`. Most polymorphic features are provided as function-like macros by the reference implementation, so as per our conventions, their names end with underscore. For documenting the syntax, optional arguments are surrounded by square brackets (such as [`,` *opt*]), and the notation [`,` *arg*=`value`] indicates that if the optional argument *arg* is omitted, then `value` acts as a default argument.

If one of the forms is permitted to evaluate some argument more than once, then the polymorphic feature is named with two trailing underscores. The multiple forms themselves have trailing underscores depending on whether they are permitted multiple evaluation of arguments; for example, `output__` is a polymorphic feature that has four forms: `output__0__`, `output__1_`, `output__2__`, and `output__3__`. Only the form `output__1_` is not permitted to evaluate its argument more than once; rest of the forms may possibly evaluate only the first argument multiple times, and only if it has a variably modified type. In summary, the naming convention for multiple argument expansion dominates: if at least one of the forms is permitted to evaluate some argument more than once, then the polymorphic macro itself is named with two trailing underscores, which also appear before the argument count in the names of individual forms. `output__` and the complementary `input__` are examples of dual polymorphism, whose behavior depends on both number and type of arguments; their precise semantics are discussed in chapter 3.

This section only provides a brief introduction to static polymorphism, which means that the exact form is resolved at translation time itself; as the reference implementation provides these features as function-like macros, their name resolution is done by the preprocessor. We shall encounter several polymorphic features in the next three chapters, before chapter 4 provides a complete picture that also demonstrates how such features can be implemented as function-like macros with the ellipsis framework; polymorphism based on argument type is commonly implemented with `_Generic` selections. C_ also supports a form dynamic polymorphism with runtime binding of method calls, which is used with extension or “inheritance” of classes and interfaces (discussed in chapter 7 and 8).

There are certain advantages of directly using a specific form instead of its polymorphic generalization: theoretically, the most obvious benefit is that the name resolution is skipped, which improves translation time. Readers who are interested in the reference implementation will find that it does not invoke any of the polymorphic macros; the specific forms are directly used instead, which leads to an appreciable decrease in preprocessing time.

0.7 Limitations

Emulating an entire dialect with the preprocessor has some inherent drawbacks. Readers who were able to successfully compile the first example may have observed some slowness in translation; a large fraction of this time is spent transpiling C_ code to C code, which involves highly resource intensive computations. Another unimpressive aspect of the reference implementation is that diagnostic messages (errors and warnings) can be cryptic, and deciphering the precise cause of an error may require familiarity with inner workings of how some feature has been implemented. Most features rely on helper macros or functions whose names end with `_C` or `_c` (possibly followed by underscores); these are not part of C_ and should not be directly used by programmers, as their availability is not guaranteed.

It is hoped that future implementors of the C_ dialect will provide alternatives that have faster compilation time and more informative diagnostics, possibly in conjunction with better code generation; this optimism is the primary motivation for documenting C_ in terms of abstract semantics rather than concrete implementation.

The reference implementation provides almost everything as macros, and the underlying preprocessor may impose a translation limit on the maximum number of parameters for function-like macros, in which case the same limit should also be applicable to the number of arguments for macro invocations. The C_ standard requires that any such limit shall be at least 127, which is the default value of `PP_MAX` used by the ellipsis framework. Portable code should not exceed this limit, which is respected by the reference implementation and sample examples.

It should be noted that the preprocessor is a separate component that does not understand most rules of the C language; one serious limitation in calling function-like macros is that brace-enclosed commas that are not parenthesized get interpreted as separate arguments, which can cause issues with unparenthesized compound literals having multiple elements in its initializer (or even a single element with a redundant trailing comma). We have already seen an example with a wrapper around the `abs` function, during the discussion on trivial function-like macros; unfortunately, that workaround is often limited to only the last argument. As an example, the library function `putc` can be additionally provided as a macro in `<stdio.h>`; for discussion, let us consider it as a plain wrapper over `fputc`.

```
#define putc(c, stream) fputc(c, stream)
```

As we saw earlier, macros like this suffer from subtle problems with compound literals; a contrived example is `putc('\n', *(Stream []){stdout,})` (`Stream` is a synonym for non-modifiable `File_` pointer, and `File_` itself means a modifiable `FILE` object). To deal with such unusual arguments, one should use ellipsis (`...`) instead.

```
#define putc(c, ...) fputc(c, __VA_ARGS__)
```

This is a small example where the first parameter `c` may be omitted, as it can be part of the ellipsis itself; however, macros may need to isolate arguments for additional logic (such as `_Generic` selection). A general recommendation is that compound literal arguments having commas should be parenthesized, which neatly avoids all these complications; a couple of alternatives are available for non-polymorphic features: a comma that does not separate arguments can be written as `COMMA`, or an entire argument with unparenthesized commas can be wrapped within `echo_`, a basic primitive of ellipsis framework. A small relaxation is that the last argument of a non-polymorphic feature allows unparenthesized commas, without requiring any workaround. As another artificial example, `guard__1_(prices, 0)` can be written as `guard__1_(prices, *(Int []){0,})`, but `guard_(prices, *(Int []){0,})` does not work with the reference implementation, as it counts three arguments: `prices`, `*(Int []){0`, and `}`; consequently, the invocation gets resolved to `guard_3_`, which is not available in C_ and causes an error.

Before moving ahead, it should be stated that this section is only meant to inform readers of some inherent drawbacks of the reference implementation, that are mostly due to dependency on the preprocessor. This is not an exhaustive list of limitations which mostly apply to pathological corner-cases in contrived code snippets; from a practical perspective, such artificial counter-examples should be of no concern to most programmers. It is also worth stressing that these weaknesses are specific to the reference implementation that serves a proof of concept for C_, and the dialect itself is not hindered by such drawbacks; however, for some features, the behavior of certain cases has been documented as implementation-defined, and implementations of C_ are also permitted to disallow such cases entirely. In particular, the reference implementation currently does not support most of these scenarios.

NOTE The choice of writing a preprocessing-based implementation is best explained by quoting Donald Knuth: "*One rather curious thing I've noticed about aesthetic satisfaction is that our pleasure is significantly enhanced when we accomplish something with limited tools.*"

At the very least, the reference implementation is a testament to metaprogramming capabilities of preprocessor.

0.8 Roadmap

Besides being the *de facto* documentation of C_, this manuscript is also intended to serve as a learning resource; the writing and presentation follow a textbook style, with the contents being organized into chapters, supplemented with a good number of illustrative examples. Features marked with an asterisk (*) currently do not have a fully portable implementation; they should be used judiciously due to their dependency on compiler-specific extensions.

Chapter 1 covers the basic types of C_, which are required to be synonymous with the corresponding standard types in C; we shall also see how such synonyms can be created with `typedef_`, the C_ extension to `typedef`.

Chapter 2 discusses various kinds of statements, such as blocks supporting early exit, branching, and iteration statements. An interesting statement is `defer_`, whose semantics have been borrowed from Zig. Unlike statements, many features can be used as sub-expressions within a larger expression; these are covered separately in chapter 3.

Chapter 4 is dedicated to the rich set of C99-compatible metaprogramming facilities provided by the ellipsis framework, which serves as the backbone of the reference implementation. Perhaps the most important contribution of C_ is a constructive proof that the C preprocessor can be used as a general model of computation, which may be of interest in the study of programming languages. Many features are modeled after basic CPU instructions, and the overall design is influenced by microprogramming architecture used in control units. This modular approach to hardware design has been integrated with the help of "deferred expansion", a consequence of an obscure preprocessing rule; iterated composition via deferred expansion is arguably the most powerful feature of C_, which opens the way for structured programming purely with macros, such as branching, iterations, and even nested recursions.

Chapter 5 is focused on arrays; unlike traditional pedagogy, the discussion of arrays has been isolated three chapters apart from pointers; this is done to stress the fact that arrays and pointers are not the same, and conflating them is a common misconception among beginners. This chapter advocates the use of "pointer to array", which offers an advantage that is leveraged by highly versatile iterators; the `op_` statement is a prime example of this.

Chapter 6 introduces C_ "methods", whose primary objective is to "isolate behavior from implementation"; the structure of this entire documentation follows the same spirit, and an architectural separation of "*what* to do" from "*how* to do" is the most important design technique in C_. Another advantage of methods is default arguments: they are useful for augmenting a method with extra parameters, without breaking existing programs that invoke the method with fewer arguments. We shall design verifiers using Floyd-Hoare logic in protocols, with the help of pre-conditions and post-conditions; for aid in diagnostics, the former requires call-site details available in a special parameter `_site` of type `Site`, which is a structure type used for storing "source code coordinates".

Chapters 7 and 8 cover dynamic typing with classes and interfaces influenced by object-oriented programming; the major features supported include encapsulation, multilevel inheritance, Liskov substitution, and runtime polymorphism. Classes are used for creating concrete data types, whereas an interface is basically an abstract data type, whose instantiation requires a concrete implementation by some class. The interplay of abstract and concrete types requires some care, but it can greatly reduce redundancy through code reuse; less code is easier to maintain.

Chapter 9 concludes the main documentation by listing C_ extensions that make the C standard library more consistent with the rest of this dialect. For every standard header in C23, a corresponding C_ header is available, which includes the standard header and provides additional enhancements, such as C_ style synonyms for types defined in the header and wrappers over standard functions; the purpose of the latter is to refine certain semantics that are left undefined by the C standard, thereby filling some missing gaps and providing diagnostic support.

Appendix A is a collection of all examples on classes and interfaces. Appendix B describes the name mangling schemes used by the reference implementation. Appendix C provides the steps to modify the reference implementation for extending the domain of integers supported by the ellipsis framework. Appendix D proposes a quantitative method for benchmarking the preprocessing overhead of function-like macros. Appendix E describes the build script provided with the reference implementation, along with a summary of diagnostic options for `gcc` and `clang`. Appendix F contains references for ISO C standard. The index is alphabetical list of features and headers in C_.

Chapter 1

Types

C_ provides synonyms for the standard data types in C; the reference implementation defines them in the header <types._>. C_ uses a type system we shall call "twin typing", where each type comes as a pair: the non-modifiable variant does not end with an underscore, and its modifiable "twin" has the same name suffixed with an underscore. By convention, C_ type names begin with an uppercase letter, and contain at least one lowercase letter.

The concept of "immutability" originates from functional programming, and is increasingly being adopted by new programming languages. C_ follows the same trend, as it can prevent accidental modifications due to typographic errors; one common source of bugs is writing the equality operator `==` as `=`, which can cause silent assignment if the left operand happens to be a modifiable lvalue. Somewhat ironically to the name C_, we advocate the use of types without underscore; modifiable types should be used judiciously, only when necessary. Besides reducing chances of unintended mutations, it can also aid in optimizations, and lesser use of underscore improves visual appeal of code.

For consistency, we also follow the same convention for variables; a trailing underscore is easy to spot, making one realize that the variable is modifiable. Variable names start with a lowercase letter to differentiate them from type names; they should not start with an underscore, which is reserved for special purposes. We start by listing the basic types with their minimum bit-width, without repeating their modifiable twins whose existence is implied.

1.1 Integer

Character, signed, unsigned, and user-defined enumeration types (`enum`) are collectively grouped as integer types.

1.1.1 Character

`Char` is the basic character type, which is also an integer type with implementation-defined signedness. By definition, `sizeof (Char)` is 1 and all bits are value bits (at least 8); character encoding scheme is implementation-defined.

NOTE Character constants are always signed as they are of type `Int_`, not `Char_`, so for example `typeof_('\n')` is `Int_`; however, string literals are of type "array of characters", so the type `typeof_(*"\n")` is `Char_` as expected.

1.1.2 Signed

Signed types can represent positive and negative integers within a nearly-symmetric range. Their representation contains the necessary value bits to support the range, one sign bit, and optional padding bits (not allowed in `Byte`). C23 requires two's complement representation, but C_ does not forbid one's complement and sign-magnitude forms.

The standard signed types are listed in non-decreasing order of their actual width and increasing order of rank; the range of a lower ranked type is contained within the range of a higher ranked type. In arithmetic expressions, `Byte` and `Short` are promoted to `Int_`, and if required by the operator, a lower ranked operand is promoted to the type of a higher ranked operand, which is also the type of the result (overflow behavior is implementation-defined).

Type	Minimum width
`Byte`	8
`Short`	16
`Int`	16
`Long`	32
`LLong`	64

NOTE Width includes sign and precision bits. `Byte_` is a synonym for `signed char`, so `sizeof (Byte)` is 1. For two's complement representation, a signed type cannot represent the negation of its most negative value.

1.1.3 Unsigned

For every signed type, there is a corresponding unsigned type with the same name prefixed with `U`: these are `UByte`, `UShort`, `UInt`, `ULong`, and `ULLong`. Unsigned types cannot represent negative numbers, and the range of each unsigned type is a superset of the non-negative range of the corresponding signed type; furthermore, the intersecting range has identical representation in both. There is no overflow with unsigned arithmetic, and integers that cannot be represented by an unsigned type are mapped to its range under the rules of modular wraparound: numbering restarts from zero after the maximum value ($2^w - 1$, where w is the bit-width, or number of value bits); negative integers are counted backwards, so for an unsigned type, -1 is mapped to its largest representable value.

NOTE `UByte` and `UShort` are usually promoted to `Int_`, which can represent all values of the former two. As a consequence, operations are performed with signed arithmetic rules; in particular, implicit modular wraparound may not apply to results outside the range of `UByte` and `UShort`. For example, `(UInt)-1 + 1` is always zero, but `(UShort)-1 + 1` would be zero iff `Int` cannot represent all values of `UShort`, in which case it is promoted to `UInt_`.

1.1.3.1 Boolean

`Bool` is an unsigned type having one value bit, and an implementation-defined number of padding bits (at least 7); the single value bit can represent only two values: 0 and 1. Any scalar expression can be converted to `Bool` and the converted result is 1 if and only if the expression is non-zero (for pointers, only null pointer compares equal to 0).

NOTE Unlike other unsigned types, there is no corresponding signed type for `Bool`; the optionally supported `BitInt (2)` is the closest substitute. The result of equality and relational operators is of a boolean nature, but for historical reasons, its type is actually `Int_`; also, a `Bool` operand is promoted to `Int_` in arithmetic expressions.

1.1.4 `False` and `True`

C_ introduces two additional integer types `False` and `True`, which are not compatible with any other type. Many features of C_ allow compile-time introspection, and for some of them the outcome is of a boolean nature; for example, `is_pointer_` takes a scalar expression and outputs 1 if the expression is of pointer type, and 0 otherwise.

To permit the use of introspective features as the controlling expression of a `_Generic` selection, the outcomes must be of incompatible types: if outcome is 0, its type is `False_`; else its type is `True_`. For instance, the macro `NULL` can expand to `0`, `((void *)0)`, or any form of null pointer constant; we can introspect if it is of pointer type.

EXAMPLE `_Generic (is_pointer_(NULL), False_ : "not pointer", True_ : "is pointer")`

NOTE If `cc_` uses `gcc`, this code has to be compiled with `cc_ -Wno-pointer-arith` (not required for `clang`).

1.2 Floating

The floating-point types are used for representing rational numbers; irrational numbers (such as π) are approximated to the nearest representable value (margin of error can be negative or positive). There are three basic floating-point types, listed in non-decreasing order of precision and increasing order of rank: `Float`, `Double`, and `LDouble`. As with integer types, any value representable with a lower ranked type is also representable with a higher ranked type. However, an information-theoretic impossibility is that a floating-point type implemented with n bits cannot "exactly" represent all values of an integer type with n value bits, when it also supports fractional values in between.

EXAMPLE Floating-point representation is usually based upon IEEE 754 or ISO/IEC 60559, and many integers are approximated to a "close enough" value. For a concrete demonstration of information loss, `Float` typically uses 32 bits, and the following program finds integers in the range $[1, 2^{32})$ that cannot be exactly represented as `Float`. The code uses `continue_` statements, which cause an immediate skip to next iteration if the condition is satisfied.

```
#include <c._>

Int_ main()
begin
    Var_  count_ = 0UL ;
    Var_ maxabs_ = 0UL ;
    Var_ maxerr_ = 0L  ;
    Var_  worst_ = 0UL ;
    Var_    sqr_ = 0ULL;
    Var_    sum_ = 0ULL;
    Var limit  =  (1ULL << 32) - 1;
    loop_(1, limit)
        Float approx = _i_;
        Var err   =  (LLong)approx - (LLong)_i_;
        continue_(!err)
        count_++;
        sqr_ += err*err;
        Var abs   =  err<0 ?  -err : err;
        sum_ += abs;
        continue_(abs <= maxabs_)
        maxabs_ = abs;
        maxerr_ = err;
         worst_ = _i_;
    end
    print_("Number of approximations is", count_, "out of", limit);
    print_("Maximum error is", maxerr_, "for", worst_);
    Var arv   =  (Float)sum_ / limit;
    print_("Average rectified value of error is", arv);
    Double_ sqrt(Double);
    Var rms   =  sqrt((Float)sqr_ / limit);
    print_("Root mean square of error is", rms);
    Var per   =  100.0f*count_ / limit;
    print_("Percentage of approximations is", per,"%");
end
```

Due to the use of `sqrt` function, the math library has to be linked separately with `-lm`, meaning "link `libm.so`".

```
$ cc_ approx.c_ -lm && ./a.out
Number of approximations is 4211081216 out of 4294967295
Maximum error is -128 for 2147483776
Average rectified value of error is 42.666016
Root mean square of error is 55.865264
Percentage of approximations is 98.046875 %
```

1.3 Optional

This section describes optional types based on conditional features in C which may not be supported by all compilers.

1.3.1 Bit-precise

A bit-precise type specifies an integer type with a particular width. `BitInt` (w) specifies a non-modifiable signed type with one sign bit and $w-1$ value bits; similarly, `UBitInt` (w) specifies a non-modifiable unsigned type with w value bits. As usual, their modifiable twins are specified as `BitInt_`(w) and `UBitInt_`(w). There will be padding bits if w is not a multiple of `width_(Char)`; padding bits may also be present for alignment with a word boundary. In type promotions, the rank of a bit-precise type is higher than the rank of another integer type with lesser width.

Bit-precise types are standardized in C23, and if they are supported by the compiler, then the header `<limits.h>` defines the macro `BITINT_MAXWIDTH` that expands to the maximum permissible width; it is at least `width_(LLong)`. C23 mandates two's complement, so range of `UBitInt` (w) is $[0, 2^w)$, and range of `BitInt` (w) is $[-2^{w-1}, 2^{w-1})$.

NOTE `<limits._>` is included by `<c._>`, and bit-precise types are available iff `BITINT_MAXWIDTH` is defined. Width w must be a positive integer not exceeding `BITINT_MAXWIDTH`; for signed bit-precise types, w cannot be 1.

1.3.2 Complex

For each standard floating-point type, a corresponding complex type can be optionally available: these are `Fcomplex`, `Dcomplex`, and `LDcomplex`, in non-decreasing order of precision of increasing order of rank. Values for these types have two parts: real and imaginary; as always, a higher ranked type can represent all values of a lower ranked type.

Complex types were introduced in C99 as a core part of the language; however, the C11 revision relegated them to conditional feature, whose support is available iff an implementation does not define the macro `__STDC_NO_COMPLEX__`.

NOTE `<types._>` includes the C_ library header `<complex._>` iff complex types are supported by the compiler.

1.3.2.1 Imaginary

If complex types are available, then an implementation can optionally provide the corresponding imaginary types: `Fimaginary`, `Dimaginary`, and `LDimaginary`. If complex types are present, the implementation provides the header `<complex.h>`; the macro `imaginary` is defined in `<complex.h>` if and only if imaginary types are available as well.

1.3.3 Decimal

The optional decimal floating-point types are `Decimal32`, `Decimal64`, and `Decimal128`, which if available, conform to the ISO/IEC 60559 formats *decimal32*, *decimal64*, and *decimal128* (respectively). These are listed in increasing order of rank, with `Decimal32` having a higher rank than `LDouble`. Being a conditionally supported feature introduced in C23, these types are provided iff the macro `__STDC_IEC_60559_DFP__` is defined by the implementation.

1.4 Void

`Void` is an "incomplete type", which means `sizeof (Void)` is disallowed, and we cannot declare an object of `Void` type, or an array of `Void`; however, pointer to `Void` is a complete object type. A function with return type `Void_` does not return any value, and a return statement in its definition must not have an expression. A function that does not accept any argument can specify `Void_` in its parameter list; C23 makes it equivalent to an empty parameter list. Also, many compilers do not warn about "unused result" if it is discarded by casting the expression to `Void`.

1.5 Synonyms

Except `False` and `True`, the types discussed so far are synonyms for C types, which can be created with `typedef_`.

Syntax

`typedef_ (` *Synonym* `,` *type-name* `)`

Constraints

Synonym shall be an identifier; if that identifier is redeclared within the same scope, then the other declaration shall also be a synonym for the same type. *type-name* shall be an object type; within a single scope, it shall not redefine a tagged enumeration type, and shall not provide an incompatible redefinition for a tagged aggregate type.

Semantics

`typedef_` is syntactically a declaration. It creates two synonyms for *type-name*: `Synonym` is non-modifiable, whereas `Synonym_` is modifiable. *type-name* can specify any object type, including array or function pointer, such as `Char [BUFSIZ]` or `Void_ (*)(Void_)`; it can also be an incomplete type. If *type-name* contains the definition of an enumeration or aggregate type (`struct` or `union`), then that type itself gets defined in the current scope. If *type-name* is a variably modified type, then it can be evaluated to determine sizes of variable length arrays (VLA).

Recommended practice

For consistency with our naming conventions, the identifier *Synonym* should begin with an uppercase letter, contain at least one lowercase letter, and not end with an underscore; it should also not end with `_C` or `_c`. Also, if *type-name* is a variably modified type, then side effects should be avoided, which can be hoisted outside `typedef_`.

1.6 typeof_

Syntax

`typeof_ (` *expression* `)`
`typeof_ (` *type-name* `)`

Constraints

expression shall not have and *type-name* shall not specify an incomplete aggregate type; if they contain a type definition, then it shall not provide an incompatible redefinition for a tagged type already defined in the same scope.

Semantics

`typeof_` specifies the resulting type of an expression with the same type as the argument to `typeof_`, after performing lvalue conversion, array to pointer decay, and function to function pointer conversion. As with `typedef_`, if *type-name* specifies definition of an enumeration or aggregate type, then that type is defined in the same scope. `typeof_` can evaluate its argument only if it is of a variably modified type, in order to determine the sizes of VLAs.

Recommended practice

Side effects are discouraged, and code should not rely on evaluation of an argument with variably modified type.

NOTE `typeof_` is similar to `typeof_unqual`, but they are not the same, particularly with arrays and functions.

EXAMPLE `typeof_("str")` is pointer type `Char *`, whereas `typeof_unqual ("str")` is array type `Char_ [4]`.

1.7 Unaliasable

A variable designates a memory object commonly known as lvalue; if the lvalue can be accessed by an expression other than the variable, then that expression is an alias: the simplest form of aliasing is a pointer to the variable.

EXAMPLE Non-modifiable types only disallow updating the variable, but the lvalue can still be modifiable.

```
Int capacity = BUFSIZ;
/* ++capacity; // constraint violation */
++*(Int_ *)&capacity; // can compile
```

To make a variable truly immutable, we need to enforce non-aliasing, which can be done by declaring with `let`.

Syntax

`let` *variable-declaration*

Constraints

variable-declaration shall not declare a function, but it can declare a function pointer.

Semantics

For a block scope declaration that does not specify any storage-class, `let` disallows aliasing, so a pointer cannot be obtained to the lvalue designated by the variable; this also applies to function parameters. The behavior is implementation-defined if `let` is used in conjunction with any storage-class specifier, or with external declarations.

EXAMPLE In the earlier code, `&capacity` will cause an error if we declare it as `let Int capacity = BUFSIZ;`

Recommended practice

The reference implementation provides `let` as the `register` keyword, so it cannot be used with external declarations, or with other storage-class specifiers; other implementations can provide `let` as a separate built-in feature.

NOTE `let` can be used with `auto` only if the latter is meant for type inference, which is standardized in C23. The use of `let` is encouraged, as disabling aliasing can sometimes provide an opportunity for better optimizations.

1.8 Inference

Syntax

`Auto` *identifier* `=` *initializer* `;`
`Auto_` *identifier_* `=` *initializer* `;`
`Var` *identifier* `=` *initializer* `;`
`Var_` *identifier_* `=` *initializer* `;`

Constraints

initializer shall be an expression, and a comma expression shall be parenthesized.

Semantics

`Auto` defines a non-modifiable variable named *identifier* whose type is inferred from resulting type of *initializer* expression, after performing lvalue conversion, array to pointer decay, and function to function pointer conversion; the expression is evaluated and the variable is initialized to its value. The behavior is implementation-defined if the *initializer* expression contains any type definition, or if multiple variables are defined in a single declaration.

`Auto_` is the modifiable twin of `Auto`, and defines a variable which can be updated thereafter. `Var` and `Var_` are respectively equivalent to `Auto` and `Auto_` preceded by `let`; in other words, they make the variable unaliasable.

Recommended practice

Variable names should start with a lowercase letter to differentiate them from type names. As a convention, *identifier* in `Auto` and `Var` should not end with underscore; *identifier_* in `Auto_` and `Var_` should end with underscore.

NOTE Type inference cannot be used to define arrays or functions, which get converted to pointer types.

1.9 Properties

This section describes features that can be used to inspect certain properties of the execution environment for arithmetic types, excluding complex types; these function-like features are: `has_sign_`, `isNBO_`, `max_`, `min_`, `width_`, and `precision_`. All of them require a type name and support integer types, with `max_` and `min_` also supporting real floating-point types. The outcome of `has_sign_` is required to be an integer constant expression, that can be used with enumerators, C11 static assertions, or C23 `constexpr`. This requirement is not imposed upon rest of the features, though it is desirable that other implementations can provide them as compile time constants; the reference implementation only supports this partially for `max_`, `min_`, `width_`, and `precision_`. None of these are required to be suitable for use with the `#if` preprocessing directive; additionally, `isNBO_`, `max_`, `min_`, `width_`, and `precision_` need not work reliably with type inference, even though all of their outcomes have well-defined types.

1.9.1 Signedness

Syntax

`has_sign_ (` *type-name* `)`

Constraints

type-name shall specify an integer type.

Semantics

`has_sign_` returns an integer constant expression which equals zero if *type-name* specifies an unsigned type, and one for signed types; the outcome is of type `Bool_`.

1.9.2 Endianness

Syntax

`isNBO_ (` *type-name* `)`

Constraints

type-name shall specify an integer type.

Semantics

`isNBO_` returns one if *type-name* specifies a multibyte type that uses network byte ordering or big-endian, and it returns zero if the type uses little-endian representation; in either case, the outcome is of type `Bool_`. The outcome is zero for the single-byte types `UByte`, `Char`, `Byte`, and it is implementation-defined for any other byte ordering.

NOTE Endianness refers to the ordering of bytes for types wider than `Byte`: big-endian means a byte with more significant value is stored at lower address; little-endian is the reverse ordering, where the least significant byte comes first, followed by more significant bytes. Many networking protocols use big-endian representation for data transmission, which can require a conversion (reversal of bytes) if the native byte ordering for a host machine is little-endian. Very rarely, exotic architectures can support some other permutation of bytes, which are collectively grouped as middle-endian or mixed-endian: one historical example is the PDP-endian ordering. For the reference implementation, `isNBO_` only checks the starting byte value when the integer 1 is represented using the given type.

EXAMPLE Most common architectures natively support `width_(Byte) == 8` and `sizeof (Short) == 2`. If `Short` uses big-endian, `256` (`0x0100` in hexadecimal) would be stored as `0x01 0x00` (`0x01` at starting address, followed by `0x00`); on the other hand, it would be stored as `0x00 0x01` in little-endian form.

1.9.3 Maximum

Syntax

`max_ (` *type-name* `)`

Constraints

type-name shall specify an arithmetic type, but not a complex type.

Semantics

`max_` returns the maximum value that can be represented with the type given by *type-name*; for floating-point types, it is the largest finite value. The outcome is of the same type as *type-name*.

1.9.4 Minimum

Syntax

`min_ (` *type-name* `)`

Constraints

type-name shall specify an arithmetic type, but not a complex type.

Semantics

`min_` returns the minimum value that is representable with the type given by *type-name*; it is zero for unsigned types, and for floating-point types, it is the smallest absolute value. The outcome is of the same type as *type-name*.

1.9.5 Width

Syntax

`width_ (` *type-name* `)`

Constraints

type-name shall specify an integer type.

Semantics

`width_` returns the number of useful bits for the type specified by *type-name*: for unsigned types, it counts the number of value bits, and for signed types, it also counts the sign bit along with value bits; padding bits are ignored.

EXAMPLE `width_(Bool)` is always 1; both `width_(UBitInt (`w`))` and `width_(BitInt (`w`))` are equal to w.

1.9.5.1 Precision

Syntax

`precision_ (` *type-name* `)`

Constraints

type-name shall specify an integer type.

Semantics

`precision_` returns the number of value bits for the type specified by *type-name*. The only difference between `width_` and `precision_` is for signed types: the former counts the sign bit as well, but the latter does not.

EXAMPLE `precision_(Bool)` is 1; `precision_(UBitInt (`w`))` is w, but `precision_(BitInt (`w`))` is $w - 1$.

NOTE For the reference implementation, the outcome of `max_`, `min_`, `width_`, and `precision_` is a constant expression if *type-name* specifies a basic type, or an extended integer type that is also available as a standard synonym in `<stdint.h>` header; in particular, the result is not a constant expression for bit-precise integer types.

1.10 Pointers

We start this section with a quote by Donald Knuth, from his article *Structured Programming with go to Statements*:

"*I do consider assignment statements and pointer variables to be among computer science's 'most valuable treasures'.*"

Pointers are ubiquitous in C, and their basic use is to refer to an lvalue by its memory address. The address of an lvalue expression (often just a variable) can be obtained with the unary operator `&` (its binary form is used for bitwise AND). The pointed-to object can be accessed by dereferencing the pointer with unary operator `*` (its binary form denotes multiplication); the dereferenced expression denotes an lvalue, so it can be subjected to the `&` operator, in which case the dereference is not done at runtime. Using pointers, the same object can be referred to in multiple places without creating separate replicas that increase memory footprint; this approach works best if the shared object is immutable, so we can have multiple readers but no writers: one classic example is string literals.

As an analogy, consider a realtor who advertises the address of a house to potential buyers, all of whom can come and view the house, but none of them can alter it (until they buy it of course). The idea of physically replicating an entire house for each interested buyer is absurd. On a similar reasoning, aggregate types (structures and unions) can have large sizes, and passing them by value can be a significant function call overhead; this can be easily avoided by passing a pointer to the data, which has a small fixed size (commonly 4 bytes or 8 bytes, and sometimes even 2 bytes on memory-constrained devices). Number of usable bits in a pointer determines size of virtual address space.

Another common use of pointers is to modify an lvalue outside its lexical scope, but within the object's lifetime; the traditional example is swapping the values of two variables with a function call, which we shall discuss soon. Dynamic memory allocators work on the same principle: a function such as `malloc` returns the pointer to an untyped object of the required size, and the object can be subsequently accessed outside the lexical scope where it was allocated, as its lifetime continues until a `free` call. A pointer can itself be a memory object, and therefore an lvalue, so we can have pointer to pointer, which is sometimes called a "double pointer" (not to be confused with "pointer to `double`"); similarly, we can have higher levels of indirection that will require additional dereferences.

One underappreciated use of pointers is for working with opaque data types, where the structural representation of some type is hidden by the interface, which only names the type. In C, these can be implemented with incomplete structures and unions, where only the tag is declared, but the type definition is not available. Pointers to incomplete types cannot be dereferenced, as the type information is unavailable; instead, a set of functions acts as an interface that provide controlled access to various attributes of each object. We shall defer an elaborate discussion on data hiding and encapsulation to chapters 7 and 8, where we reach a middle ground called "translucent object", where the dynamic type information is available, but the structural details are abstracted away from the programmer.

NOTE Pointers are not integers, even though their representation is typically an unsigned integer that denotes a memory address; on modern hosted environments, this is a virtual address. Pointers are not even arithmetic types, even though we can add a valid integer offset to a pointer (such that the resulting pointer points to some part of the same object), or subtract two pointers that both point to parts of the same object. Being able to obtain a memory address from a pointer is only its bare minimum requirement; additional bits may be utilized to store extra information known as "provenance"; however, such possibilities have not yet been explored in C_.

1.10.1 Declarations

The design of C is characterized by economy of syntax, which can make some of its rules less intuitive for beginners;

EXAMPLE C uses the same syntax for declaring pointer variables and dereferencing pointer expressions.

```
Char *
p_ = 0,
ch = 0;
```

The previous declaration is badly misleading: it can create a wrong impression that both `p_` and `ch` are pointers of type `Char *`. However, the `*` is part of the variable and not the type, so `p_` is declared as a modifiable pointer to `Char`, whereas `ch` is a non-modifiable variable of type `Char`. The same declaration can be better written as:

```
Char
*p_ = 0,
 ch = 0;
```

The rewritten declaration conveys the intent more clearly: both `*p_` and `ch` are of type `Char`, so `p_` itself must be a pointer to `Char`. But now there is another subtlety: an assignment expression `*p_ = 0` means the dereferenced lvalue `*p_` is set to zero, but within a definition, the initialization `*p_ = 0` means that `p_` is set to the null pointer.

C_ takes a different approach: we can avoid such confusion entirely by declaring pointers with `Ptr` or `Ptr_`.

Syntax

`Ptr` (*type-name*)
`Ptr_` (*type-name*)

Constraints

type-name shall not provide an incompatible redefinition of an aggregate type that has already been defined in the current scope, and it shall not redefine an enumeration type that was earlier defined in the same scope.

Semantics

`Ptr` specifies the type "non-modifiable pointer to *type-name*", whereas `Ptr_` specifies the type "modifiable pointer to *type-name*". If *type-name* contains a type definition, then that type also gets defined in the current scope.

EXAMPLE In the following declaration, both `p_` and `q_` are modifiable pointers to `Char`.

```
Ptr_(Char)
p_ = 0 ,
q_ = p_;
```

NOTE `Ptr` and `Ptr_` denote modifiability of the pointer variable, which does not affect modifiability of the dereferenced lvalue; the latter is decided solely by *type-name*.

EXAMPLE Parameters of the `swap` function are non-modifiable, but the dereferenced lvalue can be updated.

```
#include <c._>

Void_ swap
(   let Ptr (Char_) this ,
    let Ptr (Char_) that
)
begin
    let Char temp = *this;
    *this = *that;
    *that =  temp;
end
```

1.10.2 Pointer type

`Pointer` is the generic object pointer type, which can be assigned to and from any object pointer without type cast.

NOTE `Pointer_` is a synonym for `volatile Void *`; it may not be suitable for representing function pointers.

1.10.3 `is_pointer_`

`is_pointer_` can be used to statically check if the type of a scalar expression is pointer to a complete object type.

Syntax

`is_pointer_ (` *expression* `)`

Constraints

expression shall have scalar type.

Semantics

For a scalar expression that is not a pointer to an incomplete type or function type, `is_pointer_` returns one if the expression evaluates to a pointer after array to pointer decay (if applicable); otherwise it returns zero. The outcome is an integer constant expression, whose type is `False_` for zero, and `True_` for one.

EXAMPLE `is_pointer_(0)` is false; `is_pointer_((Char *)0)` and `is_pointer_("array decay")` are true.

NOTE `is_pointer_` provided by the reference implementation can portably differentiate between arithmetic type expressions and pointers to complete types; other implementations can support a broader class of expressions.

1.10.4 Null pointer

Null pointer is a special pointer value that indicates a non-existent object or function. The exact value is implementation-defined, but it is most commonly represented with all zero bits; moreover, the null pointer representation can be different for object pointer types and function pointer types. At source code level, a null pointer is denoted by the null pointer constant, which is an integer constant expression that evaluates to zero, or such an expression cast as pointer to `Void`; in other words, whenever the compiler sees 0 (possibly cast as `Void_ *`) assigned to or compared with a pointer type, it substitutes an implementation-defined representation of the null pointer. C23 also allows `nullptr` as another form of null pointer constant; `nullptr` is a predefined constant of type `nullptr_t`.

NOTE Portable programs should not use `memset` to initialize some memory with null pointer(s), as zeroing out a pointer object is not guaranteed to result in a null pointer; the proper way is to assign the null pointer constant.

1.10.4.1 `notnull__`

The `notnull__` and `notnull_` family are useful for runtime diagnosis of null pointers: when compiled in debugging mode, detecting a null pointer prints a customizable diagnostic message and then terminates the process with exit status as one; when compiled in production mode, a null pointer is replaced with an optional default pointer.

Syntax

`# include <pointer._>`

`notnull__ (` *ptr* [, *def*`=NULL` [, *text*`="(`*ptr*`) != NULL"` [, *site*`=SITE` [, *sink*`=stderr` [, *DEBUG*`=POINTER__`]]]]]`)`
`notnull__1__ (` *ptr* `)`
`notnull__2__ (` *ptr* `,` *def* `)`
`notnull__3__ (` *ptr* `,` *def* `,` *text* `)`
`notnull__4__ (` *ptr* `,` *def* `,` *text* `,` *site* `)`
`notnull__5__ (` *ptr* `,` *def* `,` *text* `,` *site* `,` *sink* `)`
`notnull__6__ (` *ptr* `,` *def* `,` *text* `,` *site* `,` *sink* `,` *DEBUG* `)`

Constraints

ptr shall be a pointer expression, and *def* shall be a pointer compatible with the type of *ptr*; more precisely, it shall be possible to assign *def* to a variable with the same type as that of *ptr*, without requiring a type cast. *text* shall be a string. *site* shall be of type `Site`. *sink* shall be of type `Stream`. *DEBUG* shall expand to either `0` or `1`.

Semantics

`notnull__` invokes `notnull__`*n*`__` if the expanded argument sequence contains n arguments. Outcome of the expression is of the same type as *ptr*. If *ptr* is not null, then that is the value of the outcome. When compiled with *DEBUG* expanding to `0`, if *ptr* is null, then value of the outcome is *def*, but having the same type as that of *ptr*.

When compiled with *DEBUG* expanding to `1`, if *ptr* is null, the process prints a diagnostic message of the following form, and then terminates by calling `exit(1)`.

`Assertion failed: (`*ptr*`) != NULL, function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

The text `(`*ptr*`) != NULL` is the default, which can be customized with the argument *text*; an empty string is used if *text* is null. *function-identifier*, *file-name*, and *line-number* collectively form the "source code coordinates", which indicate the location where the assertion failed; the default argument `SITE` is an expression that obtains these coordinates from the predefined identifier `__func__`, the `__FILE__` macro, and the `__LINE__` macro (respectively). The structure type `Site` is discussed in chapter 9; for now, we can provide the *site* argument with a compound literal, as shown below (if any of the first two members is null, then an empty string is used as the corresponding text).

`( ( Site ) { "`*function-identifier*`" , "`*file-name*`" ,` *line-number* `} )`

The message is written to the standard error stream `stderr` by default, which can be changed with the *sink*. `stderr` is also the fallback if *sink* is null; the behavior of writing to *sink* is undefined if it is not an output stream.

The default value of the *DEBUG* argument is `POINTER__`, which is an object-like macro that records the `defined` state of the `DEBUG` macro when the header `<pointer._>` was last included. If the latest inclusion of `<pointer._>` was preceded by an active definition of the macro `DEBUG`, then `POINTER__` expands to `1`; otherwise it expands to `0`. The first argument *ptr* can be evaluated more than once only if it is pointer to a variably modified type; otherwise it is evaluated once. Rest of the arguments are always evaluated only once, regardless of whether *DEBUG* is `0` or `1`.

NOTE Readers may observe that when *DEBUG* is `1`, the argument *def* is not utilized; conversely, when *DEBUG* is `0`, the arguments *text*, *site*, and *sink* are redundant. Nevertheless, if these arguments are explicitly supplied by the programmer, then they are always evaluated, even if their results remain unused. This ensures a consistent behavior when a non-essential argument involves side effects, which take place anyways to avoid unexpected surprises.

EXAMPLE `notnull__` can be used to write debugging wrappers for functions that have undefined behavior with null pointer arguments. The `echo_` macro is used as a safeguard in case `str` contains unparenthesized commas.

```
#define sscanf(str, ...) sscanf(notnull__(echo_(str), ""), __VA_ARGS__)
```

1.10.4.2 `notnull_*`

Syntax

`# include <pointer._>`

`notnull_ (` *ptr* [`,` *def*`=NULL` [`,` *text*`="(`*ptr*`) != NULL"` [`,` *site*`=SITE` [`,` *sink*`=stderr` [`,` *DEBUG*`=POINTER__`]]]]]`)`
`notnull_1_ (` *ptr* `)`
`notnull_2_ (` *ptr* `,` *def* `)`
`notnull_3_ (` *ptr* `,` *def* `,` *text* `)`
`notnull_4_ (` *ptr* `,` *def* `,` *text* `,` *site* `)`
`notnull_5_ (` *ptr* `,` *def* `,` *text* `,` *site* `,` *sink* `)`
`notnull_6_ (` *ptr* `,` *def* `,` *text* `,` *site* `,` *sink* `,` *DEBUG* `)`

Constraints

The `notnull_` family shall have precisely the same constraints as those applicable for the `notnull__` family.

Semantics

`notnull_` family evaluates each expression exactly once; rest of the semantics are identical to `notnull__` family.

NOTE The reference implementation provides these features in the header `<pointer._>`. Other implementations can provide them elsewhere or even as built-in features, but `<pointer._>` will still be relevant as its inclusion configures the behavior of these features with the macro `POINTER__`. So `<pointer._>` is part of the requirements.

1.10.5 Dereference

The dereference operator `*` can be applied only for a pointer to complete object type or function type, and only if it points to a valid object whose lifetime has not ended, or to a function of the right type. Dereferencing an invalid pointer causes undefined behavior; however, there are some exceptions where the dereference operation is not performed, such as applying the address-of operator `&` to the dereferenced expression, the controlling expression of a `_Generic` selection, `sizeof` and `typeof` operators without a variably modified type, and pointer to an array.

1.10.5.1 `fetch__`

Syntax

```
# include <pointer._>

fetch__    ( pointer [, default-value] )
fetch__1__ ( pointer )
fetch__2__ ( pointer  , default-value  )
```

Constraints

The first argument shall be pointer to a complete object type. If *pointer* is an expression of type “pointer to `T`”, then it shall be possible to assign *default-value* to a variable of type `T`, without requiring an explicit type cast.

Semantics

`fetch__` invokes `fetch__`n`__` if the expanded argument sequence has n arguments. If *pointer* is not null, then outcome is the lvalue obtained on dereferencing the pointer. When `POINTER__` expands to `1`, if *pointer* is null, then the behavior is same as if `notnull__(`*pointer*`)` is invoked. When `POINTER__` expands to `0`, a null pointer is not dereferenced, in which case the outcome is *default-value*, but with the same type as of dereferencing the pointer.

If *pointer* is of type `T *`, then the outcome is an lvalue of type `T`. When `POINTER__` expands to `0`, if *default-value* is used as the outcome, then the dereferenced lvalue is a temporary object with automatic storage duration, whose lifetime ends after the innermost enclosing block. *pointer* can be evaluated more than once only if it points to an object with variably modified type; *default-value* is always evaluated once, even if its outcome remains unused.

NOTE As the outcome is an lvalue, the dereference can be suppressed by applying address-of operator `&` to the result, which does not impact null pointer diagnosis in debugging mode compilation (`POINTER__` expands to `1`).

1.10.5.2 `fetch_*`

Syntax

```
# include <pointer._>

fetch_   ( pointer [, default-value] )
fetch_1_ ( pointer )
fetch_2_ ( pointer  , default-value  )
```

Constraints

The `fetch_` family shall have precisely the same constraints as those applicable for the `fetch__` family.

Semantics

The `fetch_` family evaluates each expression exactly once; rest of the semantics are identical to `fetch__` family.

1.10.6 Compatibility

Pointer to a modifiable type can be converted as pointer to the corresponding non-modifiable type, but the other way around should be avoided. For example, an expression of type `Ptr (Char_)` can be assigned to a variable of type `Ptr (Char)`, but attempting the converse will trigger a warning from most compilers; in particular, our use of the option `-Werror` in the `cc_` command alias will turn this into a compilation error. C allows casting away the non-modifiability, but our use of the option `-Wcast-qual` disallows that as well; the intent is to diagnose a potentially erroneous type cast, where the type should actually be non-modifiable. Explicitly discarding non-modifiability is strongly discouraged, but in some cases this can be unavoidable; such a need may arise during the implementation of legacy APIs, where changing some declaration can break existing code. The standard library has some notable examples, such as the function `strchr` declared in the header `<string.h>`: it accepts a pointer to `Char` as one of its arguments and returns a pointer to `Char_`, which points to a part of the same object as pointed to by the argument. An alternative design is to return an offset relative to the argument pointer, but most library functions are based on "prior art" from the early days of C, when non-modifiability was not as prioritized as it is today. Nevertheless, in recognition of the occasional necessity for discarding non-modifiability, C_ provides `unqual__` and `unqual_`*.

Syntax

`unqual__ (` *pointer* `)`
`unqual_ (` *pointer* `)`

Constraints

The argument shall be pointer to an object type.

Semantics

Outcome of both `unqual__` and `unqual_` has the same value as *pointer*, but its type is pointer to the corresponding modifiable type, without any type qualifier; more precisely, if the argument is pointer to a possibly qualified type `T`, then the result is the same pointer, but typed as `typeof_unqual (T) *`. `unqual__` can evaluate the argument more than once only if it is pointer to a variably modified type; `unqual_` evaluates the argument exactly once.

EXAMPLE The following code exemplifies a need to forgo non-modifiability with an implementation of `strchr`.

```
#include <c._>

Char_ *c_strchr
(   let Ptr_(Char) str_,
    let Int chr
)
begin
    guard_(str_, NULL)
    do stop_(*str_ == chr, unqual__(str_))
    while  (* str_++);
    return NULL;
end
```

NOTE Strong type safety is an integral aspect of C, and it is also respected by C_. Concerning the potential abuse of `unqual__` and `unqual_`, C_ trusts the programmer to use them judiciously, and it is the responsibility of API designers to declare prototypes in such a way that implementors are not compelled to discard non-modifiability.

Chapter 2

Statements

This chapter discusses various types of statements in C_. The most common form of statements is expression statement, which is an expression followed by a semicolon; the semicolon acts as a sequence point, which means that pending side effects before the semicolon are completed before moving on to the next statement. For example, assuming the return values have arithmetic types in the expression `f() * g() + h()`, the multiplication is done before addition due to precedence rules, but the order of evaluation of arguments is unsequenced, so there are six possible orderings in which these functions can get called. If we want to enforce a specific order of calling these functions, we need to separate them out as statements, and store their return values in temporary variables.

```
Var a = f();
Var b = g();
Var c = h();
Var sum  =  a*b + c;
```

There are several other forms of statements, which are categorized into selection or branching statements, iteration statements, jump statements, and compound statements; the latter is a sequence of declarations and statements. It is important to note that C_ declarations and statements do not require a terminating semicolon, which is harmless in most cases, but should be avoided as a general practice. This is because an extra semicolon acts as a null statement by itself, which can be problematic in certain contexts (these are syntax errors, not bugs).

EXAMPLE The following code fails to compile as `stop_` is itself a statement, and the next semicolon is a null statement outside the `if` statement, which creates a syntactic isolation of the subsequent `else` statement.

```
Void_ queue(Int);
Int_  flush();
Var_  chr = 0;
if ((chr = getchar()) == EOF)
    stop_(flush() == EOF);
else
    queue(chr);
```

Same problem occurs when an external C_ declaration is followed by a semicolon; the latter acts acts a null statement, which is disallowed outside functions. For these reasons, an unnecessary semicolon should be avoided.

2.1 Declarations

Syntactically, declarations are not statements; they are described here as the amount of content does not justify a separate chapter. Over the years, ISO C revisions have made two major amendments in the rules of declarations.

- Firstly, mixing declarations and statements is permitted since C99; earlier a declaration was not allowed to follow a statement, and the workaround was to enclose a subsequent declaration within an inner block.
- Secondly, C23 permits a declaration to immediately follow a label; it was earlier forbidden due to a grammatical restriction, and a simple workaround was to add a null statement (a semicolon) just after the label.

Some features of C_ are in fact declarations; we have already seen one of them in the previous chapter: `typedef_` is a declaration that creates a pair of type synonyms. Here we shall discuss two other kinds of declarations.

2.1.1 Static assertions

Static assertions are used to specify pre-conditions for compiling a program. Unlike runtime assertions which are expressions, static assertions are syntactically classified as declarations, so they can be placed anywhere a declaration can occur. Static assertions are checked during compilation itself, and they are evaluated in the lexical order as they appear in the source code, including those within function definitions. Static assertions are not part of the object code generated by the compiler, and if they are not satisfied during translation, it causes a constraint violation.

Syntax

`static_assert_` (*constant-expression* [, *string-literal*="*constant-expression*"])
`static_assert_1_` (*constant-expression*)
`static_assert_2_` (*constant-expression* , *string-literal*)

Constraints

The first argument shall be a non-zero integer constant expression.

Semantics

`static_assert_` invokes `static_assert_`*n*`_` if the expanded argument sequence contains n separate arguments. *constant-expression* is evaluated during compilation: if the result is zero, the constraint violation is reported with a diagnostic message. The optional *string-literal* is part of the message; if it is omitted, then the text of *constant-expression* is used as the default value. Translation continues only if the expression is non-zero, and if it contains any type definition, then that type is also created in the current scope, so there cannot be any incompatible redefinition.

NOTE Static assertions are based on C11 `_Static_assert`, which requires both the arguments; C23 makes *string-literal* optional, but the reference implementation does not depend on this for providing `static_assert_1_`.

EXAMPLE `static_assert_(width_(Byte) == 8, "POSIX requires exactly eight bits in a byte")`

2.1.2 Spare variables

Many compilers have an option for warning about unused variables. Even though such warnings can be disabled altogether, it is a good idea to enable them by default, and selectively suppress them for certain variables by adding a dummy expression with a `Void` cast; this is a common trick for suppressing warnings about unused parameters.

The only downside of the approach is that expression statements cannot be used outside functions (or any kind of statement for that matter), so if there are unused "private" variables (internal linkage) created for future use, then this approach requires an additional function where we can add dummy expressions that use such variables.

C_ offers an alternative with `spares_`, which is syntactically a declaration, so it can also occur outside functions.

Syntax

`spares_` (*identifier-list*)

Constraint

identifier-list shall be a comma separated list of identifiers, and each identifier shall refer to a variable or a function. At least one identifier shall be provided and no identifier shall be repeated; an identifier that has already been used in a `spares_` declaration shall not be reused in a subsequent `spares_` declaration within the same scope.

Semantics

`spares_` declares that variables or functions specified in *identifier-list* are potentially unused; the list should not end with a trailing comma, as implementations are not required to support it, and can be treated as a syntax error.

NOTE C23 adds the standard attribute `maybe_unused` that should be preferred if supported by the compiler.

2.2 Blocks

A block is a sequence of declarations and statements, which are executed in their lexical order until a branch or jump statement is encountered. Syntactically, a block as a whole acts as one single statement, which is why it is also called a compound statement. A C_ block is started with `begin`, and its lexical scope extends till the occurrence of a matching `end`. C_ blocks can be nested like ordinary blocks: an inner block ends before the one that encloses it.

Unlike traditional blocks enclosed within curly braces, C_ blocks also support guard clauses, which are discussed in the next section. The primary advantage is that if some condition is (un)satisfied, then it is possible to bail out early by skipping rest of the statements within the nearest enclosing block that supports jumping directly to its end. We refer to this feature as "early exit" out of a C_ block; the term does not imply exiting the process itself.

2.3 Branching

Branching statements are also called selection statements, which are used to conditionally change the sequential flow of control; in other words, branching statements are used to execute some code depending on whether a condition is satisfied or not. The next few subsections describe the syntax and semantics of branching statements in C_.

2.3.1 Guard clauses

Guard clauses check if some condition is satisfied or not: if yes, then normal control flow is not interrupted; otherwise, subsequent statements are skipped till the end of the nearest enclosing block that supports the use of `break` statement, and control jumps directly to the end of that block. Guard clauses are supported by iteration blocks, switch blocks, and any C_ block that can be lexically closed with `end` or `refed` (discussed later).

2.3.1.1 `guard_`

Syntax

`guard_` (*condition* [, *return-value*$_{opt}$])
`guard_1_` (*condition*)
`guard_2_` (*condition* , *return-value*$_{opt}$)

Constraints

condition shall be a scalar expression, and the optional *return-value* shall be an expression that can be returned by the function without a type cast. If the function return type is `Void_`, then *return-value* shall be blank.

Semantics

`guard_` invokes `guard_`n`_` if the expanded argument sequence has n arguments. If *condition* does not compare equal to zero, then control flow is not altered and *return-value* is not evaluated; otherwise the behavior of `guard_1_` is same as that of `break` statement, and `guard_2_` has the effect of a return statement with *return-value*.

2.3.1.2 `stop_`

Syntax

`stop_` (*condition* [, *return-value*$_{opt}$])
`stop_1_` (*condition*)
`stop_2_` (*condition* , *return-value*$_{opt}$)

Constraints

The `stop_` family shall have precisely the same constraints as those applicable for the `guard_` family.

Semantics

Behavior of the `stop_` family is equivalent to that of the `guard_` family with *condition* being logically negated.

NOTE Early exit happens with `stop_` family if *condition* is non-zero; `guard_` family does it otherwise.

2.3.1.3 Flattening arrow code

Arrow-shaped code is a lexically deep nesting of blocks that visually resembles an arrowhead if indented properly.

EXAMPLE The following artificial code snippet highlights the arrow (anti-)pattern.

```
if (a)
{   s();
    if (r)
    {   h();
        if (r)
        {   a();
            if (o)
            {   p();
                if (w)
                {   e();
                }
            }
        }
    }
}
```

Non-trivial arrow code can be hard to read and maintain in large projects, which is why many programmers consider it to be an anti-pattern that should be avoided; such code can be easily flattened in C_ with guard clauses.

EXAMPLE The arrow-shaped code contrived earlier can be restructured to eliminate nesting, as done below.

```
begin
    guard_(a)
    s();
    guard_(r)
    h();
    guard_(r)
    a();
    guard_(o)
    p();
    guard_(w)
    e();
end
```

2.3.2 `elif`

Syntax

`elif (` *condition* `)`

Constraints

condition shall be a scalar expression, and `elif` shall be syntactically connected to a preceding `if` statement.

Semantics

`elif (` *condition* `)` is equivalent to `else if (` *condition* `)`.

2.3.3 `if_`

Syntax

`if_ (` *condition* `)`
    declarations-and-statements$_{opt}$
`end`

Constraints

condition shall be a scalar expression, and an `if_` block shall be lexically closed by one of `end`, `e1se`, or `elif_`.

Semantics

Code within an `if_` block is executed only if *condition* is non-zero; otherwise the entire block is skipped.

2.3.4 `e1se`

Syntax

`e1se`
    declarations-and-statements$_{opt}$
`end`

Constraints

`e1se` shall be syntactically connected to a preceding `if_` block, whose lexical scope is ended by `e1se`.

Semantics

`e1se` is equivalent to `end else begin`. `e1se` block is executed only if *scalar-expression* of the preceding `if_` block equals zero; in other words they are mutually exclusive: either `if_` block or `e1se` block is executed, not both.

2.3.5 `elif_`

Syntax

`elif_ (` *condition* `)`

Constraints

condition shall be a scalar expression, and `elif_` shall lexically close a preceding `if_` block or an `elif_` block.

Semantics

`elif_ (` *condition* `)` is equivalent to `end else if_ (` *condition* `)`.

2.3.6 `switch_`

Syntax

`switch_ (` *selection* `)`
    declarations-and-statements$_{opt}$
`end`

Constraints

selection shall be an integer expression.

Semantics

`switch_` blocks allow case labels of the following forms, where each *constant-expression* shall be a distinct integer.

`case` *constant-expression* `:` *statement*
`CASE` *constant-expression* `:` *statement*
`case_ (` *constant-expression* `)` *declaration-or-statement*$_{opt}$

The next two forms are additionally permitted by C23, but forbidden by older revisions of the C standard.

`case` *constant-expression* `:` *declaration*
`CASE` *constant-expression* `:` *declaration*

If the *selection* expression of `switch_` is equal to the *constant-expression* of a case, then control jumps directly to that case statement, skipping any preceding code within the `switch_` block; thereafter subsequent statements are executed in their lexical order, until a branch or jump statement is encountered. If none of the case *constant-expression* matches the *selection* expression, then it acts as an ordinary block and executes from its beginning.

The basic `case` is a plain labeled statement, whereas the variants `CASE` and `case_` additionally have an implicit unconditional jump immediately before them, which prevents a fallthrough from the previous statement (if any). If the control reaches just before a `CASE` or `case_`, then it jumps directly to the end of that `switch_` block.

NOTE `switch_` is somewhat different from `switch`. An explicit `default` case is not mentioned in the semantics, as it is disallowed. `continue` statements are permitted, but as `switch_` is not an iteration block, `continue` acts like `break`. Also, `switch_` works like an ordinary block if none of the cases match; however, both `CASE` and `case_` have an implicit "jump to the end" immediately before them, so if a `switch_` block starts with either of them, then the non-matching behavior is identical to that of a conventional `switch` statement: the entire block is skipped. Conversely, any code that occurs lexically before the first case gets executed when a matching case is not found.

EXAMPLE The following program selects a color based on a random number (modulo 4); if the random selection does not match any of the given colors, it prints "Mixed". Bitwise AND with 3 gives an integer in the range [0, 3].

```
#include <c._>
#include <stdlib._>

Int_ main()
begin
    typedef_(Color, enum { RED, GREEN, BLUE, })
    srand(time(NULL));
    switch_((Color)(rand() & 3))
        print_("Mixed");
        case_(RED)   print_("Red");
        case_(GREEN) print_("Green");
        case_(BLUE)  print_("Blue");
    end
end
```

NOTE The selection expression is cast to type `Color` because some compilers can warn if a case is missing for an enumeration constant of that type; with `gcc` and `clang`, the option `-Wswitch-enum` enables this diagnostic.

2.3.6.1 Fallthrough

`case` does not alter the sequential flow of control, so in the absence of any branch or jump statements, when the selecting expression is equal to a `case` expression, subsequent cases that lexically follow the selected case are also executed in order; proceeding to the next `case` statement after executing the selected case is termed as fallthrough.

Fallthrough may or may not be desirable, and often the requirement is to execute only the matching case. In our previous example, the use of `case_` ensures mutual exclusion: both `CASE` and `case_` behave as if they are preceded by an implicit jump statement to the end, so only one color is printed. One important point to note is that the implicit jump applies to the end of the nearest enclosing block that supports `break` statement; this small subtlety may be insignificant for most purposes, but for the sake of completeness, we exemplify this with a concrete program.

EXAMPLE The following code demonstrates fallthrough, where both print statements will get executed.

```
#include <c._>

Int_ main()
begin
    switch_(0)
        case_(0) print_("zero");
        begin case_(1) end
        print_("one");
    end
end
```

2.3.6.2 Declaration after label

The formal language grammar specified by older revisions of the C standard did not allow a declaration to follow a label, which also applied to `case`: as per the earlier syntactic rule, the colon after a label could only be followed by a statement. C23 updates the grammar to allow both declarations and statements; however, C_ does not require complete C23 support, so for backward compatibility with the older rule, the use of `case_` may be preferred.

Both `CASE` and `case_` have an implicit jump preceding them, but the difference is that `case_` can be followed by a declaration or statement even by the older C rule, but `CASE` expression should be followed by a colon and a statement; the latter can have a leading declaration if the compiler follows the updated C23 grammar for labels.

2.3.7 `break_`

Syntax

`break_ (` *condition* `)`

Constraints

condition shall be a scalar expression. `break_` shall be used only within an iteration block, a switch block, or a C_ block that can be lexically closed with `end` or `refed`.

Semantics

`break_` executes the `break` statement only if *condition* does not evaluate to zero.

2.3.8 `continue_`

Syntax

`continue_ (` *condition* `)`

Constraints

condition shall be a scalar expression. `continue_` shall be used only within an iteration block, or a C_ block that can be lexically closed with `end` or `refed`.

Semantics

`continue_` executes the `continue` statement only if *scalar-expression* does not evaluate to zero. If the nearest enclosing block that supports `continue_` is not an iteration block, then its behavior is same as that of `break_`.

2.4 Iteration

Iteration blocks contain code that needs to be repeatedly executed as long as or until some condition is satisfied. The looping or stopping condition can be checked each time before or after executing the iteration block.

2.4.1 Entry controlled

This section describes looping blocks whose looping or stopping condition is checked at the start of each iteration.

2.4.1.1 `for_`

Syntax

`for_ (` $\textit{declaration-or-expression}_{opt}$ `;` $\textit{loop-condition}_{opt}$ `;` $\textit{expression}_{opt}$ `)`
    $\textit{declarations-and-statements}_{opt}$
`end`

Constraints

loop-condition shall be a scalar expression.

Semantics

declaration-or-expression is evaluated only once before the first iteration; declarations are visible till the end of that `for_` block. *loop-condition* is evaluated at the start of each iteration: if it is non-zero or it has been omitted, then the block is executed; otherwise iteration is stopped, and control jumps immediately after the end of that `for_` block. After each iteration, the *expression* that follows *loop-expression* gets executed if an iteration ran its natural course and reached the end of that `for_` block, or due to the execution of a `continue` statement.

NOTE For notational convenience, we can assign descriptive names to the parts of a `for_` loop, as done below.

`for_ (` *init* `;` *pre* `;` *post* `)`
    body
`end`
outside

With this notation, control flow for a conventional cycle of iterations (without jump statements) proceeds as:

$$init \rightarrow [\ pre_{true} \rightarrow body \rightarrow post\ ]_{opt} \rightarrow \cdots \rightarrow [\ pre_{true} \rightarrow body \rightarrow post\ ]_{opt} \rightarrow pre_{false} \rightarrow outside$$

2.4.1.2 `while_`

Syntax

`while_ (` *loop-condition* `)`
    $\textit{declarations-and-statements}_{opt}$
`end`

Constraints

loop-condition shall be a scalar expression.

Semantics

loop-condition is evaluated at the start of each iteration: if it is non-zero, then the loop body is executed, and the expression is re-evaluated after reaching the end of that `while_` block, or after executing a `continue` statement; looping is stopped if the *loop-condition* is found to be zero.

2.4.1.3 `until_`

Syntax

`until_ (` *stop-condition* `)`
    declarations-and-statements$_{opt}$
`end`

Constraints

stop-condition shall be a scalar expression.

Semantics

If *stop-condition* is zero, then the loop body is executed before re-evaluating it; otherwise looping is stopped.

2.4.1.4 `until`

Syntax

`until (` *stop-condition* `)` *statement*

Constraints

stop-condition shall be a scalar expression.

Semantics

`until` is equivalent to `while` with *stop-condition* being logically negated.

2.4.2 Exit controlled

We also have looping blocks whose condition is checked after each iteration; such blocks are executed at least once.

Syntax

`begin`
    declarations-and-statements$_{opt}$
`again_ (` *loop-condition* `)`
	`begin`
	    declarations-and-statements$_{opt}$
	`end_ (` *stop-condition* `)`

Constraints

Both *loop-condition* and *stop-condition* shall be scalar expressions.

Semantics

loop-condition and *stop-condition* are evaluated on reaching end of that block, or due to a `continue` statement. Execution of the block is repeated as long as *loop-condition* is non-zero, or *stop-condition* is zero, after each iteration.

NOTE `end` is equivalent to `end_(1)`. `end_` is equivalent to `again_` with the condition being logically negated.

2.4.3 Infinite loop

Syntax

`begin`
    declarations-and-statements$_{opt}$
`again`

Semantics

`again` is equivalent to `again_(1)`, where the looping condition is always true. Jump statements work as usual.

2.4.4 Integer loops

C_ provides an additional kind of iteration block to simplify looping over an arithmetic progression of integers. The conventional approach of using relational operators may not work as expected in rare cases, such as signed integer overflow on updating a control variable after the final iteration, which causes implementation-defined behavior.

EXAMPLE The following code is meant to print only three values, but it ends up producing a lot more output than one might expect: it is actually an infinite loop, assuming overflow causes the signed equivalent of wraparound.

```
#include <c._>

Int_ main()
begin
    let Int max = max_(Int);
    for_(Var_ i_ = -max; i_ <= max; i_ += max)
        print_(i_);
    end
end
```

Integer loops take care of such fine-grained subtleties, thereby easing the programmer to focus on the loop body.

Syntax

`loop_` (*range*)
`loop_` (*start* , *stop* [, *step*])
`loop_1_` (*range*)
`loop_2_` (*start* , *stop*)
`loop_3_` (*start* , *stop* , *step*)

Constraints

range shall be an expression of some `Range` type; *start*, *stop*, and *step* shall be expressions of some integer type.

NOTE *range* is pointer to a triplet that encodes an integer sequence; `Range` types are presented in chapter 5.

Semantics

`loop_` invokes `loop_`n`_` if the expanded argument sequence contains n arguments. These are used to iterate over the following arithmetic sequence, where k is a non-negative integer determined from the given relational inequality.

RELATION $|start + k * step| \leq |stop| \leq |start + (k+1) * step|$
SEQUENCE $start, start + step, start + 2 * step, \cdots, start + k * step$

`loop_1_` obtains *start*, *stop*, and *step* from *range*. For `loop_2_` and `loop_3_`, all arguments are converted to a common type, which is the type of the expression $start - stop$. For `loop_2_`, *step* is taken as -1 if *start* is greater than *stop*, 1 if *start* is smaller than *stop*, and zero otherwise. For `loop_3_`, if $start - stop$ has an unsigned type and *start* is greater than *step*, then *step* is also converted to the same unsigned type, and *start* is moved backwards by the converted step value; in other words, it considers the mathematical negation of the converted step value, even though an unsigned type cannot represent negative values. If *step* is specified as zero, two things can happen: if *start* is equal to *stop*, then the sequence is a singleton, and only one iteration is performed; otherwise, the sequence is a pair and there are two iterations: first for *start*, then for *stop*. If the result of adding *step* to *start* diverges away from *stop*, then the sequence is considered to be empty, and no iterations are performed; more precisely, $start + step$ moves farther away from *stop* if $start - stop$ has a signed type and one of the following conditions is met:

- *start* is smaller than *stop* and *step* is negative.
- *start* is greater than *stop* and *step* is positive.

Within the loop block, the modifiable variable `_i_` stores the current sequence value that can be updated, and the non-modifiable variables `_omega` and `_delta` respectively store final sequence value and converted step value.

NOTE *step* size zero is given a special meaning to specify singleton sequences, and because sometimes it may not be possible to represent "one giant leap" from *start* to *stop*; for example, `loop_(min_(Int), max_(Int), 0)` jumps directly from `min_(Int)` to `max_(Int)`, but the difference between them cannot be represented in type `Int`.

EXAMPLE The buggy code in our earlier example can be fixed with `loop_`, which correctly prints three values.

```
#include <c._>

Int_ main()
begin
    let Int max = max_(Int);
    loop_(-max, max, max)
        print_(_i_);
    end
end
```

2.5 Defer*

The defer family allows statements to be postponed, which are executed after reaching the end of the innermost block that is lexically closed with `refed`. This feature is useful for registering cleanup or release of resources.

EXAMPLE Consider a function that needs n resources for some task; if a resource is not available, then the acquired resources must be released. With three resources, we can express this requirement with the following code.

```
#include <c._>

Bool_ work()
begin
    Void_ *acquire(), release(Void_ *),
    utilize(Void_ *, Void_ *, Void_ *);
{   Var  r1 = acquire();
    guard_(r1, 0)
{   Var  r2 = acquire();
    if_(!r2)
        release(r2);
        return 0;
    end
{   Var  r3 = acquire();
    if_(!r3)
        release(r2);
        release(r1);
        return 0;
    end
    utilize(r1, r2, r3);
    release(r3);
}   release(r2);
}   release(r1);
}   return 1;
end
```

The crucial observation is that resource i has to be released if another resource is available. A call to `release` lexically occurs i times for the i^{th} resource, so for n resources there will be $n * (n+1)/2$ instances of `release` calls, even though at most n such calls will actually be made (one for each acquired resource). This bloats the executable as code size grows at a quadratic rate in terms of the number of resources. `defer_` offers a more elegant approach by postponing the release of acquired resources; deferred statements are executed in reverse order of registration.

NOTE Before presenting a cleaner alternative with `defer_`, we shall take another look at the earlier code, and observe the use of brace-enclosed nested blocks: the intent is to make a released resource unavailable for any subsequent use, which is done by making the pointer variable go out of scope just after calling `release`. Some programmers advocate setting it to `NULL`, so that defensive null pointer checks can catch an invalid use; however, that means the variable has to be modifiable, which opens up the possibility of unintended mutations due to bugs.

We prefer to follow the previous chapter's advice on enforcing immutability as much as possible, so we rely on scoping rules instead: every resource is associated with its own lexical scope that starts with the resource is acquired, and ends when the resource is released; this ensures that each pointer variable is no longer accessible once its use is over. Plain nested scopes can prove beneficial in the long run for functions that manage several resources. We consider this practice as a part of structured programming that has also been followed in later chapters.

2.5.1 `defer_*` and `refed*`

Without any further ado, we shall now re-implement our earlier example with the help of `defer_` and friends.

```
#include <c._>

Bool_ work()
deferrable
    Void_ *acquire(), release(Void_ *),
    utilize(Void_ *, Void_ *, Void_ *);
    Var  r1 = acquire();
    if (!r1) return_(0)
    defer_(release(r1))
    Var  r2 = acquire();
    if (!r2) return_(0)
    defer_(release(r2))
    Var  r3 = acquire();
    if (!r3) return_(0)
    defer_(release(r3))
    utilize(r1, r2, r3);
    return_(1)
refed
```

`defer_` registers expressions and statements to be postponed at the end of the nearest enclosing block that ends with `refed`. The name `refed` alludes to the fact that `defer_` statements get executed in reverse order of reaching them; however, multiple statements within a single `defer_` are executed in their lexical order. For example, if one writes `defer_(`*expr1*`;` *expr2*`)` followed by `defer_(`*expr3*`;` *expr4*`)`, they get executed in due course in the order: *expr3*`;` *expr4*`;` *expr1*`;` *expr2*`;` . Reaching `refed` via `guard_1_` or `stop_1_` executes `defer_` statements registered within that block; however, `guard_2_` and `stop_2_` directly return from the function, ignoring deferred statements.

NOTE Due to a minor flaw in the reference implementation, `refed` can generate false warnings when it ends a function (other than `main`) whose return type is not `Void_`; for `gcc` and `clang`, one can disable such warnings with the option `-Wno-return-type`. It was also observed during testing that `gcc` generates false warnings for the given example that is implemented using `defer_`; these latter ones can be suppressed with `-Wno-maybe-uninitialized`.

2.5.2 deferrable* and start*

A function that supports the use of `defer_` is started with `deferrable` instead of `begin` or a plain opening brace. If statements are to be postponed to the end of an inner block, then such a block is started with `start`; reaching its corresponding `refed` executes the statements deferred within that block in reverse order of registering them.

EXAMPLE The following code demonstrates execution sequence of deferred statements within nested blocks.

```
#include <c._>

Int_ main()
deferrable
    defer_(puts("All's well that ends well"))
    puts("Before block");
    start
        defer_(puts("leaving block"))
        defer_(printf("\tPrint before "))
        puts("\tInside block");
    refed
    defer_(puts("Yet another defer"))
    puts("After block");
refed
```

NOTE Asterisk means that the reference implementation relies on non-standard extensions for providing these features; they are reported by `-Wpedantic` (enabled in `cc_`), but we can suppress such warnings with `-Wno-pedantic`.

```
$ cc_ -Wno-pedantic defer.c_ && ./a.out
Before block
    Inside block
    Print before leaving block
After block
Yet another defer
All's well that ends well
```

2.5.3 return_* and yield*

A conventional `return` is oblivious of `defer_`, so the pending statements do not get executed. `return_` takes an expression that is compatible with the function return type, that is to say, it should be possible to return that value without an explicit type cast; the expression is first evaluated, and then deferred statements are executed starting from the innermost block, moving to outer blocks in the order of their ending. When deferred statements of the function block started with `deferrable` has been executed, the expression that was evaluated earlier is returned.

`return_` cannot be used if function return type is `Void_`; as `return` keyword is unaware of pending statements, an early return should be written as `yield;` which returns from the function after executing the deferred statements.

2.5.4 DEFER_MAX

An implementation can impose a upper bound on the number of `defer_` statements that can be registered by a single function; this limit is available as the macro `DEFER_MAX`, which expands to a non-negative integer constant. The behavior is implementation-defined if the number of `defer_` statements registered at runtime exceeds this limit; an implementation that does not have this limit shall still define the macro `DEFER_MAX`, which shall expand to 0.

NOTE The reference implementation defines `DEFER_MAX` as 128, which can be changed in the header `<defer._>`.

Chapter 3

Expressions

This chapter describes several features of C_ that are syntactically classified as expressions: each expression has a well-defined type, which can be a complete type, a function type, or an incomplete type; if the type is not `Void_` or an incomplete aggregate type, an expression also has a value. Unlike statements discussed in the previous chapter, these features can be used as sub-expressions within a larger expression; an expression becomes a statement when it is ended by a semicolon. Before proceeding to the main content, it should be mentioned that this chapter does not contain a complete collection of all expression-like features in C_; the rest have been documented in later chapters.

NOTE ISO C grammar does not allow statements within expressions; however, the GNU C dialect permits this as a non-standard language extension that is supported by many compilers (including `gcc` and `clang`). C_ features marked with an asterisk (*) are provided by the reference implementation using this extension; depending on how a C_ feature is implemented, this is required to avoid multiple evaluation of arguments with variably modified types. Use of such features can generate compiler warnings that can be disabled with `-Wno-pedantic` for `gcc` and `clang`.

3.1 Constant expressions

The features listed in this section are compile time constants; they can also be used in `#if` preprocessing directive.

NOTE The reference implementation provides all three features described below as object-like macros.

3.1.1 `FALSE` and `TRUE`

`FALSE` and `TRUE` are differently typed expressions whose values are of boolean nature. `FALSE` is an integer constant expression with type `False_` and value zero. `TRUE` is an integer constant expression with type `True_` and value one.

NOTE The reference implementation provides `FALSE` and `TRUE` as object-like macros. `False_` is a synonym for an `enum` having one member: `FALSE` with value zero; `True_` is a synonym for another `enum` having one member: `TRUE` with value one. Both enumeration constants are masked by macro definitions that can be used in `#if` directives, and an expression whose outcome is `FALSE` or `TRUE` can be further used for `_Generic` selections during translation.

3.1.2 `NULLPTR`

`NULLPTR` is the null pointer constant of type `Ptr_(Void_)`.

NOTE The C standard does not specify a particular type for the macro `NULL`, which can be of integer type, pointer type, or the special type `nullptr_t` (since C23); the only benefit of `NULLPTR` is that its type is well-defined.

3.2 Compound operators

C_ has two unusual operators whose semantics can be achieved by the combined use of unary, equality, and logical operators; for this reason, we refer to them as compound operators. Both are useful for expressing logical assertions; they are frequently used for writing pre-conditions and post-conditions within protocols (introduced in chapter 6).

Due to the way they are provided by the reference implementation, both operators require parentheses as part of their syntax. If the parenthesized expression is part of a larger expression, then it should be doubly parenthesized, as precedence and associativity are implementation defined when there are other operators outside the parentheses.

3.2.1 `iff`

Syntax
`(` *expression* `iff` *expression* `)`
Constraints
Both operands shall be scalar expressions.
Semantics
`iff` checks logical equivalence of the two scalar expressions: the result has value one if both operands compare equal to zero, or both operands compare unequal to zero; otherwise the result is zero. The result is of type `Int_`.

NOTE The reference implementation provides `iff` as an object-like macro that expands to the text `)==0 == !(`

3.2.2 `implies`

Syntax
`(` *implicant* `implies` *implicand* `)`
Constraints
Both *implicant* and *implicand* shall be scalar expressions.
Semantics
`implies` checks if a logical implication exists between the two operands: the result has value one if *implicant* is zero or *implicand* is non-zero; otherwise the result is zero. The result is of type `Int_`, and the operation follows short-circuit evaluation: *implicant* is evaluated first, and if it is non-zero, only then *implicand* is evaluated.

NOTE The reference implementation provides `implies` as an object-like macro that expands to `)==0 || (`

3.3 Scalar to text

The features described here give a text corresponding to the outcome of comparing a scalar expression with zero. For all these features, the outcome is a pointer to a non-modifiable object, which is an array of plain/wide characters.

3.3.1 Scalar to string

Syntax
`text_BOOL_ (` *expression* `)`
`text_Bool_ (` *expression* `)`
`text_bool_ (` *expression* `)`
Constraints
expression shall have scalar type.

Semantics

`text_BOOL_` gives the string "`FALSE`" if *expression* compares equal to zero, and "`TRUE`" otherwise (sans quotes).
`text_Bool_` gives the string "`False`" if *expression* compares equal to zero, and "`True`" otherwise (sans quotes).
`text_bool_` gives the string "`false`" if *expression* compares equal to zero, and "`true`" otherwise (sans quotes).
The result is of type `Ptr_(Char)` but not a string literal, so it cannot be used for translation time concatenation.

3.3.2 Scalar to wide string

Syntax

`wtext_BOOL_ (` *expression* `)`
`wtext_Bool_ (` *expression* `)`
`wtext_bool_ (` *expression* `)`

Constraints

expression shall have scalar type.

Semantics

`wtext_BOOL_`, `wtext_Bool_`, and `wtext_bool_` are respectively the wide string equivalents of `text_BOOL_`, `text_Bool_`, and `text_bool_`. Rest of the semantics are identical, and the outcome is of type `Ptr_(WChar)`.

NOTE `WChar_` is a synonym for `wchar_t`, and `WChar` is its non-modifiable twin; both are provided by `<stddef._>`.

3.4 Bit shifting

Certain semantics of shift operators are implementation-defined or undefined, particularly with negative integers. C_ fills these gaps with bit shifting features that have well-defined behavior for all integers when compiled in production mode; however, operands that cause undefined behavior for the built-in shift operators are caught at runtime when compiled in debugging mode, as if with assertions. C_ broadens the set of operands for which shifting by s bits is same as multiplication or division by 2^s, even with one's complement and sign-magnitude representations.

The following subsections describe features that have a more uniform semantics for bitwise shift operations on negative integers; these features are representation agnostic and their behavior is defined in terms of value of the outcome for both unsigned and signed integers. For the different representations of signed integers, there can be two distinct notions of uniformity in outcome when a bitwise operation is performed on a negative value:

- Outcomes are specified in terms of bit patterns, but their corresponding values depend on the representation.
- Outcomes are specified in terms of values, but their corresponding bit patterns depend on the representation.

C_ prioritizes the latter kind of uniformity for signed types, as bit pattern manipulations are mostly done with unsigned types. For each of these features, the left expression is converted to the widest integer type that is not a bit-precise type: `UIntmax` is used for `ulsh_` and `ursh_`; `Intmax` is used for `lsh_` and `rsh_`. The converted left expression is shifted by the number of bits given by the right expression; the latter is converted to an unsigned type and it should be less than width of the left expression's type. The outcome has the same type as the left expression.

The bit shifting features are grouped into the headers `<lshift._>` and `<rshift._>`; their behavior can be configured during compilation with the `DEBUG` macro. These two headers are aggregated by `<shift._>`; the purpose of including `<shift._>` in a source file is to ensure a common configuration for both `<lshift._>` and `<rshift._>`.

NOTE Both `UIntmax` and `Intmax` are synonyms defined in `<stdint._>`; they can be extended integer types. With the ubiquity of two's complement representation that is also mandated by C23, the bit shifting features described here are of limited use, and most programmers are likely to prefer the basic shift operators for simple bit pattern manipulations. Chapter 5 presents dynamically resizable bit arrays that serve a wider variety of purposes.

3.4.1 Left shift

`ulsh_` and `lsh_` are configured by `LSHIFT__`, which is an object-like macro that expands to the integer constant `1` if the macro `DEBUG` remains defined when the header `<lshift._>` is included; otherwise `LSHIFT__` expands to `0`.

When `LSHIFT__` expands to `1`, `ulsh_` and `lsh_` are configured in debugging mode, in which the shift value given by the unsigned right expression is asserted to be less than width of the left expression's type both before and after conversion; otherwise `ulsh_` and `lsh_` are configured in production mode, in which the result is zero if shift value is greater than or equal to width of the left expression's original type or `width_(UIntmax)` (whichever is smaller).

3.4.1.1 Unsigned

Syntax

`# include <lshift._>`
`ulsh_ (` *left-expression* `,` *right-expression* `)`

Constraints

Both expressions shall have integer types.

Semantics

left-expression is converted to the type `UIntmax`; we shall denote this converted value as u. *right-expression* is converted to an unsigned type; we shall denote this converted value as s. Let w denote minimum width of the type of *left-expression* before and after conversion. If shift s is less than width w, then u is left shifted by s bits, as if with `u << s`; more precisely, if the unsigned value u is expressed as $\sum_{i=0}^{w-1} b_i.2^i$ with the bit pattern $b_{w-1} \cdots b_0$ (here b_0 is the least significant bit), then the result of left shifting it by s bits has the value $\sum_{i=s}^{w-1} b_{i-s}.2^i$ with the bit pattern $b_{w-1-s} \cdots b_0 0_1 \cdots 0_s$. This unsigned result is converted back to the original type of *left-expression*.

3.4.1.2 Signed

Syntax

`# include <lshift._>`
`lsh_ (` *left-expression* `,` *right-expression* `)`

Constraints

Both expressions shall have integer types.

Semantics

left-expression is converted to the type `Intmax`; we shall denote this converted value as n and its absolute value $|n|$ as u. *right-expression* is converted to an unsigned type; we shall denote this converted value as s. Let w denote minimum width of the type of *left-expression* before and after conversion. If shift s is less than width w, then u is left shifted by s bits, as if with `ulsh_(u, s)`; only the least significant $w - 1$ bits of the shifted value are retained, and sign negation is done if n is negative. This becomes the outcome after conversion to the type of *left-expression*.

3.4.2 Right shift

The object-like macro `RSHIFT__` records the `defined` state of `DEBUG` macro when the header `<rshift._>` is included: it expands to the integer constant `1` if `DEBUG` was defined, and `0` otherwise; `ursh_` and `rsh_` are configured accordingly.

When `RSHIFT__` expands to `1`, `ursh_` and `rsh_` are configured in debugging mode, in which the shift value given by the unsigned right expression is asserted to be less than width of the left expression's type both before and after conversion; otherwise `ursh_` and `rsh_` are configured in production mode, in which the result is zero if shift value is greater than or equal to width of the left expression's original type or `width_(UIntmax)` (whichever is smaller).

3.4.2.1 Unsigned

Syntax

`# include <rshift._>`
`ursh_ (` *left-expression* `,` *right-expression* `)`

Constraints

Both expressions shall have integer types.

Semantics

left-expression is converted to the type `UIntmax`; we shall denote this converted value as u. *right-expression* is converted to an unsigned type; we shall denote this converted value as s. Let w denote minimum width of the type of *left-expression* before and after conversion. If shift s is less than width w, then u is right shifted by s bits, as if with `u >> s`; more precisely, if the unsigned value u is expressed as $\sum_{i=0}^{w-1} b_i.2^i$ with the bit pattern $b_{w-1}\cdots b_0$ (here b_0 is the least significant bit), then the result of right shifting it by s bits has the value $\sum_{i=s}^{w-1} b_s.2^{i-s}$ with the bit pattern $0_1\cdots 0_s b_{w-1}\cdots b_s$. This unsigned result is converted back to the original type of *left-expression*.

3.4.2.2 Signed

Syntax

`# include <rshift._>`
`rsh_ (` *left-expression* `,` *right-expression* `)`

Constraints

Both expressions shall have integer types.

Semantics

left-expression is converted to the type `Intmax`; we shall denote this converted value as n and its absolute value $|n|$ as u. *right-expression* is converted to an unsigned type; we shall denote this converted value as s. Let w denote minimum width of the type of *left-expression* before and after conversion. If shift s is less than width w, then the absolute value u is right shifted by s bits, as if with `ursh_(u, s)`; if n is negative, then sign negation is performed on the shifted value. This becomes the outcome after conversion to the original type of *left-expression*.

3.5 Evaluation

3.5.1 `value_`

Syntax

`value_ (` *expression* `)`

Semantics

Outcome is the value after lvalue conversion, array to pointer decay, and function to function pointer conversion.

3.5.2 `faux_`

Syntax

`faux_ (` *type-name* `,` *expression* `)`

Constraints

type-name shall not be `Void_` or an incomplete aggregate type. *expression* shall not be an unparenthesized comma expression; it shall be a valid function call argument for a parameter whose type is specified by *type-name*.

Semantics

expression is not evaluated, and the outcome is a `Void_` expression which does not affect translation or execution.

3.5.3 `eval_`

Syntax

`eval_ (` *type-name* `,` *expression* `)`

Constraints

`eval_` shall have precisely the same constraints as those applicable for `faux_`.

Semantics

`eval_` has the same semantics as `value_`.

NOTE `eval_` combines the type checking constraints of `faux_` with the evaluation semantics of `value_`.

3.5.4 `lvalue__`

Syntax

`lvalue__ (` *expression* `)`

Semantics

The result is an unqualified lvalue initialized to the value of *expression*, with the same type as `value_(`*expression*`)`. In other words, the same type conversions are performed as done by `value_`: if *expression* is an array, then the lvalue is a pointer to base element; if *expression* is a function, then the lvalue is a corresponding function pointer. The lvalue has automatic storage duration, and its lifetime is limited to the innermost block where it is created.

expression can be evaluated more than once only if it has a variably modified type.

NOTE As the outcome is an lvalue, the address-of operator `&` can be applied to it; however, the resulting pointer should not be used outside the current block, as the object lifetime expires and the pointer becomes dangling.

3.6 Conditionals

Syntax

`test_ (` *condition* `,` *yes-expression* [`,` *no-expression*] `)`
`test_2_ (` *condition* `,` *yes-expression* `)`
`test_3_ (` *condition* `,` *yes-expression* `,` *no-expression* `)`

Constraints

condition shall be a scalar expression.

Semantics

`test_` invokes `test_`n`_` if the expanded argument sequence contains n arguments. *condition* is evaluated first: if it is non-zero, then *yes-expression* is evaluated, and `test_3_` does not evaluate *no-expression*; otherwise *condition* is zero, *yes-expression* is not evaluated, and `test_3_` evaluates *no-expression*. The outcome is a `Void_` expression.

NOTE The `test_` family works like the conditional operator `? :`, except that there is no constraint between the types of second and third expressions: exactly one of them is evaluated, and the result is discarded. The primary use of `test_` family is to write branching expressions with mutually incompatible types within a larger expression.

3.7 Generic selections

Generic selections use the converted type of an expression to select another expression; the former expression is not evaluated. They are analogous to `switch`, except that `switch` selections are done at runtime using value of the controlling expression, whereas generic selections are performed during translation, based on the type of the controlling expression. Generic selections were standardized in C11, and they have the following syntax:

`_Generic (` *controlling-expression* [`,` *type-name* `:` *expression*] ⋯ [`, default :` *expression*] `)`

A generic selection consists of a controlling expression followed by a comma-separated list of generic associations:
type-name `:` *expression*
`default` `:` *expression*

A `_Generic` selection requires at least one generic association having one of the above forms; at most one `default` association is permitted. As comma is used to separate different parts of a generic selection, none of the expressions can be an unparenthesized comma expression. Type for the value of *controlling-expression* is determined as if with `typeof_(`*controlling-expression*`)`, which is itself equivalent to `typeof (value_(`*controlling-expression*`))`.

If the resulting type is compatible with *type-name* of a generic association, then outcome of the generic selection is the *expression* for that matching association. If the optional `default` association is present, then its *expression* is selected iff no other generic association matches the resulting type of *controlling-expression*; it need not be the last in sequence. Exactly one generic association must match, and type of a `_Generic` expression is same as type of its selected expression, as if the entire generic selection is replaced by an implicitly parenthesized form of that expression. None of the other expressions are evaluated; however, they must all be semantically valid C expressions.

NOTE Recall that `typeof_` and `value_` perform lvalue conversion, array to pointer decay, and function to function pointer conversion; if *type-name* of an association is qualified, then its *expression* will never be selected.

EXAMPLE `_Generic (*"", Char : "error")` causes a compilation error as the controlling expression `*""` undergoes lvalue conversion, and its resulting type `Char_` does not match the *type-name* of any generic association.

3.7.1 Generalized generic

The basic `_Generic` selection only allows a single controlling expression; C_ offers an extension that generalizes it to a tuple of controlling expressions, and correspondingly, each generic association also permits a tuple of types.

Syntax

`generic_ ( (` *expression-list* `)` [`, (` *type-list* `,` *expression* `)`] $\cdots$ [`, (` *default-expression* `)`] `)`

Constraints

`generic_` accepts a sequence of parenthesized lists, where the first list is a tuple of controlling expressions, and each subsequent list is an association. *expression-list* shall be a comma-separated list of expressions, such as $expr_1$ `,` $\cdots$ `,` $expr_n$. Each *type-list* shall be a comma-separated list of types, such as $type\text{-}name_1$ `,` $\cdots$ `,` $type\text{-}name_k$, and a *type-name* shall not specify an incomplete aggregate type; a *type-name* can be `Void_` only if there are no other types in an association list. Number of types in a *type-list* need not match the count of expressions in *expression-list*.

An optional association without *type-list* is a default association, which shall consist of a single *default-expression*; at most one default association is permitted, and it need not be the last in sequence. All associations shall be distinct, such that any two non-default association lists differ in at least one *type-name*, after considering type adjustments described by the semantics. There shall be exactly one association list that can be selected based on semantic rules.

Semantics

A type sequence is constructed from the controlling *expression-list*, as if with `typeof_`; more precisely, each expression in the first list is subjected to lvalue conversion, array to pointer decay, and function to function pointer conversion; then its resulting type is determined. For each *type-name* in a non-default association list, outermost type qualifiers are discarded, array types are adjusted to pointer types, and function types are adjusted to function pointer types; the latter two adjustments are identical to those which are performed for function parameters.

After performing these conversions, if the type list constructed from the controlling *expression-list* is compatible element-wise with the type list of an association, then that association is selected, and its expression becomes the outcome of the `generic_` expression; otherwise a default association list is present, having a single *default-expression* that is selected as the outcome. Type of the `generic_` expression is same as type of the selected expression, as if the entire `generic_` expression is replaced by an implicitly parenthesized form of the selected expression.

Other than the expression from the selected association list, none of the other expressions are evaluated.

EXAMPLE `generic_` can be used to emulate "function overloading", where a unified function call interface is designed for a group of possibly related functions. It is a form of static polymorphism because binding a call to a specific function is decided during translation itself, depending on the type of arguments given by that invocation.

```
#include <c._>

UInt_ add2(let UInt l, let UInt r)
begin
    print_("Adding two integers", l, "and", r);
    return l + r;
end

Char_ *append(let Ptr(Char_) l, let Ptr(Char) r)
begin
    output__(stdout, "", "Joining two strings \"", l, "\" and \"", r, "\"\n");
    Char_ *strcat(String_, String);
    return strcat(l, r);
end

UInt_ add3(let UInt l, let UInt m, let UInt r)
begin
    print_("Adding three integers", l, ",", m, ",", r);
    return l + m  +  r;
end

Void_ def(...)
begin
    print_("Useless default function hides error");
end

#define adder_(...) generic_((__VA_ARGS__), (def),\
 (String_, String , append), (UInt, UInt, add2),\
 (String_, String_, append), (UInt, UInt, UInt, add3))\
(__VA_ARGS__)

Int_ main()
begin
{   Var sum = adder_(1U, 2U);
    print_("Sum is", sum);
}{  Var cat = adder_((Char_ [20]){"static "}, "polymorphism");
    print_("Function overloading is a form of", cat);
}{  Var sum = adder_(4U, 8U, 16U);
    print_("Sum is", sum);
}   adder_("Bad call");
end
```

NOTE C23 allows variable argument functions to be declared without a named parameter before ellipsis (...).

3.7.2 Qualifier sensitivity

Both `_Generic` and its extension `generic_` ignore type qualifiers when checking type compatibility of a controlling expression (list) with the type (list) of an association. C_ has another kind of generalization called `generiq_`, which is a qualifier preserving variation of `generic_`: both have the same syntax, but different constraints and semantics.

Syntax

`generiq_` `(` `(` *expression-list* `)` [`,` `(` *type-list* `,` *expression* `)`] $\cdots$ [`,` `(` *default-expression* `)`] `)`

Constraints

`generiq_` accepts a sequence of parenthesized lists, where the first list is a tuple of controlling expressions, and each subsequent list is an association. *expression-list* shall be a comma-separated list of expressions, such as $expr_1$ `,` $\cdots$ `,` $expr_n$. Each *type-list* shall be a comma-separated list of types, such as $type\text{-}name_1$ `,` $\cdots$ `,` $type\text{-}name_k$. Number of types in a *type-list* need not match the count of expressions in *expression-list*.

An optional association without *type-list* is a default association, which shall consist of a single *default-expression*; at most one default association is permitted, and it need not be the last in sequence. All associations shall be distinct, such that any two non-default association lists differ in at least one *type-name*, after considering type qualifiers. There shall be exactly one association list that can be selected based on semantic rules.

Semantics

A type sequence is constructed from the controlling *expression-list* retaining type qualifiers, and if it is compatible element-wise with the type list of an association, then that association is selected, and its expression becomes the outcome of the `generiq_` expression; otherwise a default association list is present, having a single *default-expression* that is selected as the outcome. Type of the `generiq_` expression is same as type of the selected expression, as if the entire `generiq_` expression is replaced by an implicitly parenthesized form of the selected expression.

Other than the expression from the selected association list, none of the other expressions are evaluated.

EXAMPLE Our `cc_` alias for `gcc` and `clang` includes the option `-Wwrite-strings` which enforces the type of string literals to be “array of `Char`”, consistent with their non-modifiable nature; we can use `generiq_` to test this.

```
generiq_((*""), (Char, FALSE), (Char_, TRUE))
```

The outcome would be `FALSE` when compiled with `cc_`, and `TRUE` when compiled with `cc_ -Wno-write-strings`.

NOTE Recall that `FALSE` has type `False_` and `TRUE` has type `True_`, so type of the result indicates its value.

3.8 Detecting qualifiers

Syntax

`has_qualifier_` `(` [*qualifier* `,`] *expression* `)`
`has_qualifier_` `(` [*qualifier* `,`] *type-name* `)`
`has_qualifier_1_` `(` *expression* `)`
`has_qualifier_1_` `(` *type-name* `)`
`has_qualifier_2_` `(` *qualifier* `,` *expression* `)`
`has_qualifier_2_` `(` *qualifier* `,` *type-name* `)`

Constraints

expression shall have object type and shall not designate a bit-field. *type-name* shall not specify a function type.

Semantics

`has_qualifier_` invokes `has_qualifier_`n`_` if the expanded argument sequence contains n arguments. The outcome of `has_qualifier_1_` is `FALSE` if *expression* or *type-name* is unqualified; otherwise the outcome is `TRUE`.

`has_qualifier_2_` checks if *expression* or *type-name* is qualified with *qualifier*, which can specify multiple qualifiers separated by whitespace: the outcome is `TRUE` if *qualifier* is detected in the type, and `FALSE` otherwise.

EXAMPLE `has_qualifier_(Char_)` is `FALSE`. `has_qualifier_(const volatile, volatile Char)` is `TRUE`.

3.9 Call stack growth

Two features can be used to inspect the direction of growth for the function call stack in the execution environment. If the call stack grows towards lower addresses when a function call pushes a new activation record, then `DNSTACK` equals one and `UPSTACK` equals zero; otherwise the call stack grows towards higher addresses and the values are opposite: `UPSTACK` equals one and `DNSTACK` equals zero. Both `DNSTACK` and `UPSTACK` are expressions of type `Bool_`; they are not required to be constant expressions, and need not be suitable for use with `#if` preprocessing directive.

NOTE The reference implementation provides both `UPSTACK` and `DNSTACK` as object-like macros.

3.10 Allocation

The features listed in the following subsections are used for dynamic memory allocation: on success, they return a typed pointer to an object whose lifetime is not limited by the lexical scope, and extends throughout the process until that pointer is passed to the library function `free`. If the required allocation cannot be obtained in one contiguous memory block, then the outcome is a null pointer. The reference implementation provides these features as function-like macros in the header <allocation._>, which also declares the `free` function as: `Void_ free(Void_ *);`

NOTE It is the responsibility of a programmer to release the acquired memory when it is no longer required, as there is no automatic "garbage collection" for unreachable objects. On most hosted environments, dynamically allocated memory is released by the operating system when the process terminates, so calling `free` just before exiting the process is often a redundant operation that causes a marginal increase in code size and execution time.

3.10.1 New allocation

The `new__` and `new_` families are used for dynamically allocating a new modifiable object. The outcome is a suitably typed pointer to the object, and on success, size of the allocation is same as size of the dereferenced pointer.

3.10.1.1 `new__`

Syntax

`new__` (*expression* [, *array-length* [, *initializer*]])
`new__` (*type-name* [, *array-length* [, *initializer*]])
`new__1__` (*expression*)
`new__1__` (*type-name*)
`new__2__` (*expression* , *array-length*)
`new__2__` (*type-name* , *array-length*)
`new__3__` (*expression* , *array-length* , *initializer*)
`new__3__` (*type-name* , *array-length* , *initializer*)

Constraints

expression shall have a complete object type and shall not designate a structure bit-field member. *type-name* shall specify a complete object type. *array-length* shall have an integer type. It shall be possible to use the value of *initializer* to initialize a variable declared with the same type as *expression*, or the type specified by *type-name*.

Semantics

`new__` invokes `new__`n`__` if the expanded argument sequence contains n arguments. In all cases, *expression* or *type-name* may not be evaluated at all if they do not specify a variably modified type, or they can be evaluated more than once only if their type is variably modified. *array-length* can be evaluated multiple times only if it is not an integer constant expression. *initializer* is always evaluated only once. The order of evaluation is unspecified.

A non-null outcome of `new__1__` is pointer to an object that has the unqualified type of *expression* or *type-name*.

`new__2__` converts *array-length* to type `Size` and a non-null outcome is pointer to an array whose elements have the unqualified type of *expression* or *type-name*; length of the array is equal to the converted value of *array-length*.

`new__3__` allocates an array in the same way as `new__2__`, and on success, `new__3__` also initializes each element of the array with the resulting value of *initializer* after it is converted to the type of *expression* or *type-name*.

NOTE `Size_` is a synonym for `size_t`, which is defined as the type of a `sizeof` expression. For `new__2__` and `new__3__`, the outcome is of type "pointer to array" which encodes length information; a similarly typed outcome is also given by `new__1__` if its *expression* or *type-name* specifies an array type, such as `new__(Char_ [BUFSIZ])`.

EXAMPLE `new__(Char_ *, 10, 0)` is a portable approach to populate a dynamically allocated array of ten elements with null pointers. Note that this may not be equivalent to simply zeroing out the memory with `calloc` or `memset`, as there are execution environments (mostly archaic) where null pointer representation has non-zero bits.

3.10.1.2 `new_*`

Syntax

`new_` (*expression* [, *array-length* [, *initializer*]])
`new_` (*type-name* [, *array-length* [, *initializer*]])
`new_1_` (*expression*)
`new_1_` (*type-name*)
`new_2_` (*expression* , *array-length*)
`new_2_` (*type-name* , *array-length*)
`new_3_` (*expression* , *array-length* , *initializer*)
`new_3_` (*type-name* , *array-length* , *initializer*)

Constraints

The `new_` family shall have precisely the same constraints as those applicable for the `new__` family.

Semantics

`new_` family evaluates each expression and *type-name* only once; rest of the semantics are same as `new__` family.

3.10.2 Resizing arrays

The `renew__` and `renew_` families are used for resizing dynamically allocated arrays. An existing array can be relocated if it cannot be resized in-place, so the resulting pointer may not compare equal to the original pointer.

3.10.2.1 `renew__`

Syntax

`renew__` (*array-pointer* , *array-length* [, *initializer*])
`renew__2__` (*array-pointer* , *array-length*)
`renew__3__` (*array-pointer* , *array-length* , *initializer*)

Constraints

array-pointer shall be pointer to unqualified complete array type. *array-length* shall have integer type. It shall be possible to assign *initializer* to element of the array obtained on dereferencing *array-pointer*, without type cast.

Semantics

`renew__` invokes `renew__`n`__` if the expanded argument sequence contains n arguments. In all cases, *array-pointer* shall be suitable for passing to `free`; otherwise the behavior is undefined. *array-pointer* can be evaluated more than once only if element type of the array is variably modified. *array-length* is converted to type `Size`, and it can be evaluated multiple times only if it is not an integer constant expression. *initializer* is evaluated only once.

A non-null outcome of `renew__2__` is pointer to an array having `(Size)(`*array-length*`)` elements, with the same element type as that of *array-pointer*. If *array-pointer* is null, then the behavior is identical to `new__2__`, which allocates a new array. If the outcome is null, then resizing could not be done, and the original array is preserved.

`renew__3__` resizes an array in the same way as `renew__2__`, and if the array is expanded, then `renew__3__` also initializes each additional element of the resized array with the resulting value of *initializer* after it is converted to the array element type. For the purpose of initialization, the current array length is determined solely from the type of *array-pointer*, which may or may not reflect true length of the array object. For instance, `renew__3__` also permits *array-pointer* to be null, but as per the constraints, its type shall be pointer to an unqualified complete array type: if the length encoded by that type is smaller than the converted value of *array-length*, then initialization starts from the index equal to the inferred old length, and array elements prior to that index remain uninitialized.

NOTE If the element type is not variably modified, then *array-pointer* is evaluated only once, even if the array type itself is variably modified. Practically speaking, shrinking an array should always be possible in-place, but in theory, any resizing operation can relocate the array and produce a new pointer, leaving the old pointer dangling.

3.10.2.2 `renew_*`

Syntax

`renew_` `(` *array-pointer* `,` *array-length* [`,` *initializer*] `)`
`renew_2_` `(` *array-pointer* `,` *array-length* `)`
`renew_3_` `(` *array-pointer* `,` *array-length* `,` *initializer* `)`

Constraints

The `renew_` family shall have precisely the same constraints as those applicable for the `renew__` family.

Semantics

The `renew_` family evaluates each expression exactly once; rest of the semantics are identical to `renew__` family.

3.10.3 Conditional allocation

3.10.3.1 `need__`

Syntax

`need__` `(` *expression* `)`

Constraints

expression shall be a pointer to a complete object type.

Semantics

If *expression* is not null, then that is the outcome; otherwise a non-null outcome is pointer to a dynamically allocated object, whose size is same as that of the dereferenced *expression*. The outcome is of the same type as *expression*, which can be null only if *expression* itself is null and the required size could not be allocated.

expression can be evaluated more than once only if it is pointer to a variably modified type.

3.10.3.2 `need_*`

Syntax

`need_` `(` *expression* `)`

Constraints

`need_` shall have precisely the same constraints as those applicable for `need__`.

Semantics

`need_` evaluates the pointer *expression* exactly once; rest of the semantics are identical to `need__`.

3.11 Input

C_ offers debuggable features that simplify reading formatted input for most common purposes. A wide variety of data types are supported, and programmers do not need to remember format specifiers for each type; however, the semantics of processing the text or wide text input have been described in terms of format specifiers: this is because the reference implementation provides these features using the `scanf` family of standard library functions. As always, other implementations of C_ need not rely on `scanf` family, but the semantics should still be preserved, and input shall be consumed and interpreted in precisely the same way as prescribed by the `scanf` format specifiers.

The reference implementation provides these features in the header `<input._>`. An object-like macro `INPUT__` determines the debugging configuration of all features listed in the following subsections. If the macro `DEBUG` remains defined when `<input._>` is included, then `INPUT__` expands to `1`, and the features provided by `<input._>` are collectively configured in debugging mode, in which all pointer arguments are asserted to be not null.

If `DEBUG` is not found to be defined when `<input._>` is included, then `INPUT__` expands to `0`, which configures the features under consideration in production mode, whose behavior with null pointers is described in the semantics.

3.11.1 Default source

`scan__` and `scan_` read from the standard input stream `stdin`, and interpret the data as per the type of lvalue where the processed data value needs to be stored; the first invalid character that causes a failure is left unconsumed.

3.11.1.1 `scan__`

Syntax

```
# include <input._>
scan__ ( expression-list )
```

Constraints

expression-list shall be a comma-separated list of expressions, and each expression shall be an unqualified lvalue for which a pointer can be obtained with address-of operator `&`. Each lvalue shall have one of the following types:

`UByte_`	`UShort_`	`UInt_`	`ULLong_`	`UInt_least8_`	`UInt_least16_`	`UInt_least32_`	`UInt_least64_`
`Byte_`	`Short_`	`Int_`	`LLong_`	`Int_least8_`	`Int_least16_`	`Int_least32_`	`Int_least64_`
`UInt8_`	`UInt16_`	`UInt32_`	`UInt64_`	`UInt_fast8_`	`UInt_fast16_`	`UInt_fast32_`	`UInt_fast64_`
`Int8_`	`Int16_`	`Int32_`	`Int64_`	`Int_fast8_`	`Int_fast16_`	`Int_fast32_`	`Int_fast64_`
	`Dec128_`	`Dec32_`	`Dec64_`	`Size_`	`UIntptr_`	`ULong_`	`UIntmax_`
`Char_`	`Float_`	`Double_`	`LDouble_`	`Ptrdiff_`	`Intptr_`	`Long_`	`Intmax_`
		`Void *`	`Void_ *`	`Char_ [`*n*`]`	`WChar_ [`*n*`]`		

NOTE `Char_ [`*n*`]` and `WChar_ [`*n*`]` denote complete array types having *n* elements. Plain pointers to `Char_` or `WChar_` are not supported for storing text as string or wide string because length cannot be inferred from the type, and a sufficiently long input text can get written past the buffer end, thereby corrupting adjacent memory.

Types named as `UInt`*n*`_` and `Int`*n*`_` are synonyms for exact-width types; types named as `UInt_least`*n*`_` and `Int_least`*n*`_` are synonyms for minimum-width types; types named as `UInt_fast`*n*`_` and `Int_fast`*n*`_` are synonyms for fastest minimum-width types. These synonyms are provided by the header `<stdint._>`, discussed in chapter 9.

Most implementations of C define these types as synonyms for the basic types; however, the standard allows them to be extended integer types, which can be separately provided by compilers in addition to and incompatible with the basic integer types. Moreover, the exact-width types along with `UIntptr_` and `Intptr_` are optional; these and the conditionally supported decimal floating-point types can be used with `scan__` subject to their availability.

Variables defined with `register`, `let`, `Var`, or `Var_` are unsuitable for `scan__`, as their address cannot be taken.

Semantics

scan__ reads from stdin and interprets the text as a value in the range of an lvalue in *expression-list*. Lvalues are assigned from left to right, and all expressions are evaluated in an unspecified order before reading starts. Input is consumed as long as the unconverted text read so far can be interpreted as a valid value as per the specifiers listed below. The first non-matching character acts as a delimiter and the unconverted input read till then is represented as a value that is stored in the first lvalue. If there are subsequent lvalues in *expression-list*, then input text is interpreted and represented as per their types. For each lvalue, any whitespace character (as identified by isspace function) found before the first matching character is consumed and discarded. An assignment is unsuccessful for an lvalue if a valid value cannot be constructed from the matching characters found (if any): this can happen if a delimiter is found too early, or if the input ends due to end of file (EOF), or due to an error or interruption. When an lvalue cannot be assigned, reading stops there and subsequent lvalues (if any) are also left unassigned; if the interpretation fails due to the early occurrence of a non-matching character, then that character is not consumed.

For each lvalue in *expression-list*, input text is interpreted as if scanf is called with a pointer to that lvalue as the second argument, the first argument being a format string with a percent symbol "%" followed by one of the following format specifiers as per the non-array lvalue type; for the array types, % occurs after an opening backtick (`).

UByte_	"hhu"	Byte_	"hhi"	Char_	" c"
UShort_	"hu"	Short_	"hi"	Float_	"g"
UInt_	"u"	Int_	"i"	Double_	"lg"
ULong_	"lu"	Long_	"li"	LDouble_	"Lg"
ULLong_	"llu"	LLong_	"lli"	Dec32_	"Hg"
Size_	"zu"	Ptrdiff_	"ti"	Dec64_	"Dg"
UIntmax_	"ju"	Intmax_	"ji"	Dec128_	"DDg"
UInt8_	SCNu8	Int8_	SCNi8	Void *	"p"
UInt16_	SCNu16	Int16_	SCNi16	Void_ *	"p"
UInt32_	SCNu32	Int32_	SCNi32	Char_ [$n+1$]	"`%$n$[^`]`"
UInt64_	SCNu64	Int64_	SCNi64	WChar_ [$n+1$]	"`%$n$l[^`]`"
UInt8_least_	SCNuLEAST8	Int8_least_	SCNiLEAST8		
UInt16_least_	SCNuLEAST16	Int16_least_	SCNiLEAST16		
UInt32_least_	SCNuLEAST32	Int32_least_	SCNiLEAST32		
UInt64_least_	SCNuLEAST64	Int64_least_	SCNiLEAST64		
UInt8_fast_	SCNuFAST8	Int8_fast_	SCNiFAST8		
UInt16_fast_	SCNuFAST16	Int16_fast_	SCNiFAST16		
UInt32_fast_	SCNuFAST32	Int32_fast_	SCNiFAST32		
UInt64_fast_	SCNuFAST64	Int64_fast_	SCNiFAST64		
UIntptr_	SCNuPTR	Intptr_	SCNiPTR		

If an lvalue is of type Char_ [n] or WChar_ [n], its matching input text is enclosed within backticks, optionally preceded by whitespace. For that lvalue, at most $n-1$ non-backtick characters after the opening backtick are stored in the array, and a null character is appended (wide null character is used for WChar_ arrays). If a closing backtick is found among the first n characters after the opening backtick, it is consumed but not stored; otherwise the n^{th} character (if any) is not consumed, and matching failure occurs for any subsequent lvalue in the same *expression-list*.

An lvalue can be evaluated more than once only if it is a variable length array, namely Char_ [n] or WChar_ [n] with n not being an integer constant expression. Outcome of scan__ is of type Int_: a positive outcome indicates number of lvalues assigned in sequence; otherwise none were assigned, and a negative outcome indicates read error.

If INPUT__ is 1, pointers to lvalues are asserted to be not null; otherwise INPUT__ shall be 0, and if pointer to a non-array lvalue is null, scan__ proceeds as if it was not null: matching text is consumed and counted in outcome.

3.11.1.2 `scan_*`

Syntax

`# include <input._>`
`scan_` (*expression-list*)

Constraints

`scan_` shall have precisely the same constraints as those applicable for `scan__`.

Semantics

`scan_` evaluates each lvalue in *expression-list* exactly once; rest of the semantics are identical to `scan__`.

NOTE `scan__` and `scan_` cannot store whitespace in a `Char_` lvalue. For signed types, if the input (optionally preceded by sign) starts with `0`, it is interpreted as octal; if it starts with `0X` or `0x`, it is interpreted as hexadecimal.

3.11.2 Custom source

`input__` and `input_` offer a unified mechanism for reading from both input streams as well as strings or wide strings, with an optional separator that specifies a text pattern to be matched between two consecutive inputs.

3.11.2.1 `input__`

Syntax

`# include <input._>`
`input__` ([*source*=`stdin` , [*separator*=`""` ,]] *expression-list*)
`input__0__` (*source* , *separator* , *expression-list*)
`input__1__` (*expression*)
`input__2__` (*source* , *expression*)
`input__3__` (*source* , *separator* , *expression*)

Constraints

source shall be a stream (`File_ *`), a string, or a wide string. *separator* shall be a string or a wide string: if *separator* is a string, then *source* shall not be a wide string; otherwise *separator* is a wide string, and *source* shall not be a string. *expression* shall be a single lvalue expression and *expression-list* shall be a comma-separated list of lvalue expressions, each of which shall have precisely the same constraints as those applicable for `scan__`.

Semantics

`input__` invokes `input__0__` if the expanded argument sequence contains more than three arguments; otherwise it invokes `input__`*n*`__` if the expanded argument sequence contains n arguments, with n not exceeding three. `input__1__` is equivalent to `scan__` with a single expression. `input__2__` reads from *source*, terminated by a null byte or wide null character for string and wide string respectively; rest of the semantics are similar to `input__1__`, and if *source* is an input stream, it is assumed to be byte-oriented. `input__3__` is similar to `input__2__` with some constraints between *source* and *separator*: if these are not violated and *source* is an input stream, then *separator* decides how the text is processed: if *separator* is a string, then *source* is assumed to be byte-oriented; otherwise *separator* is a wide string and *source* is assumed to be wide-oriented. `input__0__` is a customizable variation of `scan__`: the first argument specifies a *source*, and the second argument specifies a pattern to be matched between every two consecutive inputs. *separator* can be any pattern that is supported by `scanf`, but it should not contain any format specifier: more precisely, any percent symbol occurring in *separator* should be immediately followed by another percent symbol, so that each pairing of `%%` matches a literal percent symbol in the input. If *separator* contains a percent symbol not paired with another percent symbol just after that, then *separator* is dereferenced but not utilized, and an empty string `""` or wide string `L""` is used as the separator instead. The behavior is undefined if *source* is a stream that is not an input stream, or its orientation is different from what is indicated by *separator*.

source and *separator* are evaluated exactly once, and an lvalue in *expression* or *expression-list* can be evaluated more than once only if it is a variable length array, namely Char_ [*n*] or WChar_ [*n*] with *n* not being an integer constant expression. Outcome of the input__ family is of type Int_: a positive outcome indicates the number of lvalues assigned in sequence; otherwise none of them were assigned, and a negative outcome indicates read error.

If INPUT__ expands to 1, then *source*, *separator* for input__0__, and each pointer to an lvalue in *expression* or *expression-list* is asserted to be not null; otherwise INPUT__ shall be 0, and null pointers are handled as follows:

- If *source* is null, then the outcome is equal to EOF, and none of the lvalues are assigned.
- If *separator* for input__0__ is null, it is evaluated and replaced by a pattern with one space (" " or L" ").
- If pointer to a non-array lvalue in *expression* or *expression-list* is null, the input__ family proceeds as if that pointer was not null: a matching text is consumed and counted in outcome, but the interpreted value is lost.
- If pointer to an array lvalue is null, then attempting to store a matching text causes undefined behavior.

NOTE For input__3__, only the data type of *separator* is used to indicate the input stream orientation, and the value of *separator* is not required at all; however, it is still evaluated and the result is discarded without dereferencing, so *separator* can also be a null pointer for input__3__ (the type is relevant, the value is not).

3.11.2.2 input_*

Syntax

include <input._>
input_ ([*source*=stdin , [*separator*="" ,]] *expression-list*)
input_0_ (*source* , *separator* , *expression-list*)
input_1_ (*expression*)
input_2_ (*source* , *expression*)
input_3_ (*source* , *separator* , *expression*)

Constraints

The input_ family shall have precisely the same constraints as those applicable for the input__ family.

Semantics

input_ family evaluates each expression exactly once; rest of the semantics are identical to input__ family.

3.12 Output

C_ offers debuggable features that simplify writing formatted input; these are complementary to the input features we saw earlier. A wide variety of data types are supported, and as with the input counterparts, programmers do not need to remember format specifiers for each type; however, the semantics of processing semantics for different data types have been described in terms of format specifiers: this is because the reference implementation provides these features using the printf family of standard library functions. Other implementations of C_ can provide them as built-in features having the same semantics as described by this document, and the textual representation for the value of a permitted *expressions* should be identical to the one prescribed by a printf format specifier.

The reference implementation provides these features in the header <output._>. An object-like macro OUTPUT__ determines the debugging configuration of all features listed in the following subsections. If the macro DEBUG remains defined when <output._> is included, then OUTPUT__ expands to 1, and the features provided by <output._> are collectively configured in debugging mode, in which the *sink* and *separator* pointers are asserted to be not null.

If DEBUG is not found to be defined when <output._> is included, then OUTPUT__ expands to 0, which configures the features under consideration in production mode, whose behavior with null pointers is described in the semantics.

The header <io._> aggregates both <input._> and <output._>; the purpose of including <io._> in a source file is to provide a common debugging configuration for all input and output facilities based on the DEBUG macro.

3.12.1 Default sink

`print_` writes to the standard output stream `stdout`; space acts as a separator, and a newline is printed at the end.

3.12.1.1 `print_`

Syntax

```
# include <output._>
print_ ( expression-list )
```

Constraints

Each expressions shall have one of the following types after doing lvalue conversion and array to pointer decay.

UByte_	UShort_	UInt_	ULLong_	UInt_least8_	UInt_least16_	UInt_least32_	UInt_least64_
Byte_	Short_	Int_	LLong_	Int_least8_	Int_least16_	Int_least32_	Int_least64_
UInt8_	UInt16_	UInt32_	UInt64_	UInt_fast8_	UInt_fast16_	UInt_fast32_	UInt_fast64_
Int8_	Int16_	Int32_	Int64_	Int_fast8_	Int_fast16_	Int_fast32_	Int_fast64_
Bool_	Dec128_	Dec32_	Dec64_	Size_	UIntptr_	ULong_	UIntmax_
Char_	Float_	Double_	LDouble_	Ptrdiff_	Intptr_	Long_	Intmax_
	Void *	Void_ *	Char *	Char_ *	WChar *	WChar_ *	

NOTE Unlike the lvalues required by input facilities, the expressions given to `print_` and other output facilities are not constrained to be lvalues. The permitted type list of `print_` is a superset of the types supported by `scan__`.

Semantics

`print_` writes the textual representation of each expression to `stdout`. Expressions are printed left to right as per their sequence in *expression-list*, and they are evaluated in an unspecified order before printing the first expression; consecutive outputs are separated by a single space character, and a newline is printed at the end.

Text printed for each expression is formatted as per one of the following `printf` specifiers (preceded by `"%"`), depending on the resulting type of each expression after it undergoes lvalue conversion and array to pointer decay.

UByte_	"hhu"	Byte_	"hhi"	Bool_	"d"
UShort_	"hu"	Short_	"hi"	Char_	"c"
UInt_	"u"	Int_	"i"	Float_	"g"
ULong_	"lu"	Long_	"li"	Double_	"lg"
ULLong_	"llu"	LLong_	"lli"	LDouble_	"Lg"
Size_	"zu"	Ptrdiff_	"ti"	Dec32_	"Hg"
UIntmax_	"ju"	Intmax_	"ji"	Dec64_	"Dg"
UInt8_	PRIu8	Int8_	PRIi8	Dec128_	"DDg"
UInt16_	PRIu16	Int16_	PRIi16	Void *	"p"
UInt32_	PRIu32	Int32_	PRIi32	Void_ *	"p"
UInt64_	PRIu64	Int64_	PRIi64	Char *	"s"
UInt8_least_	PRIuLEAST8	Int8_least_	PRIiLEAST8	Char_ *	"s"
UInt16_least_	PRIuLEAST16	Int16_least_	PRIiLEAST16	WChar *	"ls"
UInt32_least_	PRIuLEAST32	Int32_least_	PRIiLEAST32	WChar_ *	"ls"
UInt64_least_	PRIuLEAST64	Int64_least_	PRIiLEAST64		
UInt8_fast_	PRIuFAST8	Int8_fast_	PRIiFAST8		
UInt16_fast_	PRIuFAST16	Int16_fast_	PRIiFAST16		
UInt32_fast_	PRIuFAST32	Int32_fast_	PRIiFAST32		
UInt64_fast_	PRIuFAST64	Int64_fast_	PRIiFAST64		
UIntptr_	PRIuPTR	Intptr_	PRIiPTR		

`print_` evaluates each expression exactly once and its outcome is of type `Int_`: a non-negative outcome indicates the number of characters printed by that invocation (counting each space separator and the terminating newline); a negative outcome means nothing was printed due to write error. If an expression results in a pointer to `Char`/`Char_` or `WChar`/`WChar_`, the behavior is undefined if it does not refer to a valid null terminated string or wide string.

NOTE The format specifies starting with `PRI`, along with those starting with `SCN` (mentioned in the semantics of `scan__`), are collectively defined as object-like macros in the C standard header `<inttypes.h>` (extended by `<inttypes._>`). These macros expand to implementation-defined format specifiers for the type synonyms provided by another standard header `<stdint.h>` (extended by `<stdint._>`). Recall that these synonyms can be mapped to extended integer types, each of which can have different format specifiers for `printf` and `scanf` families; in general, a macro starting with `PRI` need not expand to the same format specifier as its counterpart macro named with `SCN`.

3.12.2 Custom sink

`output__` and `output_` provide common facilities for writing to output streams as well as character or wide character buffers; an optional separator specifies text to be placed between two consecutive expressions, gluing them together.

3.12.2.1 `output__`

Syntax

```
# include <output._>
output__    ( [sink=stdout , [separator=" " ,]] expression-list )
output__0__ ( sink , separator , expression-list )
output__1__ ( expression )
output__2__ ( sink , expression )
output__3__ ( sink , separator , expression )
```

Constraints

sink shall be a stream (`File_ *`), or an unqualified and complete array of ordinary characters (`Char_ [`*n*`]`) or wide characters (`WChar_ [`*n*`]`). *separator* shall be a string or a wide string: if *separator* is a string, then *sink* shall not be a `WChar_` array; otherwise *separator* is a wide string, and *source* shall not be a `Char_` array. *expression* and each argument in *expression-list* shall have precisely the same constraints as those applicable for `print_`.

Semantics

`output__` invokes `output__0__` if the expanded argument sequence has more than three arguments; otherwise it invokes `output__`*n*`__` if the expanded argument sequence contains n arguments, with n not exceeding three. `output__1__` is similar to `print_` with a single expression, except that an extra newline is not printed at the end. `output__2__` is an extension of `output__1__` that writes to *sink*: if sink is a (wide-)character array of length n, then at most $n-1$ (wide-)characters are written, and a null (wide-)character is appended; otherwise *sink* is assumed to be a byte-oriented output stream. `output__3__` is similar to `output__2__` with some constraints between *sink* and *separator*: if these are not violated and *sink* is an output stream, then *separator* decides how the text is printed: if *separator* is a string, then *sink* is assumed to be byte-oriented; otherwise *separator* is a wide string and *sink* is assumed to be wide-oriented. `output__0__` is a customizable variation of `print_` without an extra newline: the first argument specifies a *sink*, and the second argument specifies a text to be printed between every two consecutive expressions. *separator* should not contain any format specifier: more precisely, any percent symbol in *separator* should be immediately followed by another percent symbol, so that each pairing of `%%` prints a literal percent symbol. If *separator* contains a percent symbol not paired with another percent symbol just after that, then it is dereferenced but not utilized, and an empty string `""` or wide string `L""` is used as the separator. The behavior is undefined if *sink* is a stream that is not an output stream, or its orientation is different from what is indicated by *separator*.

sink can be evaluated more than once only if it is a variable length array `Char_ [`*n*`]` or `WChar_ [`*n*`]`, with n not being an integer constant expression; rest of the arguments are evaluated once. Outcome of the `output__` family is of type `Int_`: a non-negative outcome indicates the number of characters printed by that invocation; a negative outcome means nothing was printed due to write error. If a printable expression results in a pointer to `Char`/`Char_` or `WChar`/`WChar_`, the behavior is undefined if it does not refer to a null terminated string or wide string.

If `OUTPUT__` expands to `1`, then *sink* and `output__0__` *separator* are asserted to be not null; otherwise `OUTPUT__` shall expand to `0`, and null pointers are handled as follows:

- If *sink* is null, then the outcome is equal to `EOF`, and none of the expressions are printed (but all are evaluated).
- If *separator* for `output__0__` is null, it is evaluated and replaced by a text with only one space (`" "` or `L" "`).

3.12.2.2 `output_*`

Syntax

`# include <output._>`
`output_ (` [*sink*=`stdout ,` [*separator*=`" " ,`]] *expression-list* `)`
`output_0_ (` *sink* `,` *separator* `,` *expression-list* `)`
`output_1_ (` *expression* `)`
`output_2_ (` *sink* `,` *expression* `)`
`output_3_ (` *sink* `,` *separator* `,` *expression* `)`

Constraints

The `output_` family shall have precisely the same constraints as those applicable for the `output__` family.

Semantics

`output_` family evaluates each expression exactly once; rest of the semantics are identical to `output__` family.

NOTE Unlike `print_`, the `output__` and `output_` families do not write an extra newline at the end.

3.13 Logging

C_ provides basic logging facilities that can be configured with two object-like macros: `LOGGER` and `LOGGER__`. Logging is enabled only if `LOGGER` expands to `1`; otherwise `LOGGER` is not defined or it shall expand to `0`, which disables logging in a translation unit until `LOGGER` is defined as `1`. `LOGGER__` records whether the macro `DEBUG` is defined each time the header `<logger._>` is included, which decides the debugging configuration of logging facilities.

NOTE Changing the definition of `LOGGER` does not require a re-inclusion of `<logger._>` to come into effect.

Syntax

`# include <logger._>`
`logger_ (` [*sink*=`stderr ,` [*separator*=`" " ,`]] *expression-list* `)`
`logger_0_ (` *sink* `,` *separator* `,` *expression-list* `)`
`logger_1_ (` *expression* `)`
`logger_2_ (` *sink* `,` *expression* `)`
`logger_3_ (` *sink* `,` *separator* `,` *expression* `)`

Constraints

sink shall be a stream (`File_ *`). *separator* shall be a string or a wide string. *expression* and each argument in *expression-list* shall have precisely the same constraints as those applicable for `print_` and the `output__` family.

Semantics

`logger_` invokes `logger_0_` if the expanded argument sequence contains more than three arguments; otherwise it invokes `logger_`*n*`_` if the expanded argument sequence contains n arguments, with n not exceeding three.

If `LOGGER` is defined as an object-like macro that expands to `1`, then logging is enabled, and the `logger_` family constructs a log message that starts with a blank line, followed by a heading text having the form given below.

```
Mon Dec 25 12:34:56 2023
function func, file file.c_, line 25
```

Timestamp is determined from the return value of `time` function just before logging; function identifier, file name, and line number are respectively obtained from `__func__`, `__FILE__`, and `__LINE__`. Heading is followed by the textual representation of expressions as they would be printed by the `output__` family. `logger_1_` writes the output to `stderr`; rest of the semantics are same as those of the `output__` family, except that *sink* can only be a stream, which should be a byte-oriented output stream for `logger_2_`, and for `logger_0_` or `logger_3_`, *sink* should be an output stream without orientation or the same orientation as inferred from the type of *separator*.

If `LOGGER` is not defined or it expands to `0`, then all arguments are evaluated once, and the results are discarded. The macro `LOGGER__` determines the debugging configuration of logging facilities in the same way `OUTPUT__` configures the `output__` family, both in debugging mode (`LOGGER__` as `1`) and in production mode (`LOGGER__` as `0`). In debugging mode, both *sink* and `logger_0_` *separator* are asserted to be not null even if logging is disabled; in other words, a definition of `LOGGER` (or its absence) does not affect the debugging configuration of `logger_` family.

Chapter 4

Ellipsis

The ellipsis framework is a collection of metaprogramming facilities that work entirely using the C preprocessor; it serves as a foundation for the reference implementation of C_, as the rest of its features are made possible with the help of preprocessing-based transpilation performed using macros discussed throughout this chapter.

Technically, the ellipsis framework is nothing but an assortment of non-trivial macros, organized into a hierarchy of several header files, culminating in the header <ellipsis._>. All implementations of C_ are required to provide these headers, and unlike the rest of this dialect, all features of the ellipsis framework are required to be available as macros, compatible with the preprocessing rules of C99 which standardized function-like macros accepting a variable number of arguments, defined with a parameter list that ends with three dots `...` (known as "ellipsis"). For C99 compatibility, at least one argument must be provided for the ellipsis; a macro argument can be blank, and the number of arguments is determined from the count of unparenthesized commas that act as argument separator.

This framework essentially provides a mini language within C itself, capable of performing any kind of logical and mathematical computation using only the preprocessor. To give an analogy, the basic use of macros for trivial text substitutions is merely the tip of the iceberg: computational capabilities of C99 preprocessors have remained largely undiscovered for decades. Ellipsis framework is a constructive proof showing that the preprocessor can be used as a general computation model using iterated composition, which can emulate branching, iteration, and recursion. Macros for fundamental operations are modeled on common arithmetic and logical instructions of microprocessors.

Basic support for computations is over a limited domain of non-negative decimal integer constants, which can be extended to larger integers or even fractional numbers if these are represented as tuples; for example, a floating-point number can be represented as a pair, with the parts before and after the decimal point expressed as integers. Larger integers can be broken down into a comma-separated list of smaller integers, each of which shall be within the supported domain; for example, 1234 can be expressed as the tuple `(12, 34)`. Signed integers can also be expressed as a tuple whose first element indicates the sign: zero means non-negative and non-zero means negative.

This entire chapter is based on preprocessing, so we do not need all options included in the `cc_` alias, and warnings about unused options can be safely ignored. To see the preprocessed code without actually compiling it, we can use the `-E` option with `gcc` and `clang`, so we shall test the examples in this chapter with `cc_ -E`; going one step further, we can also ensure C99 compatibility with the option `-std=c99`. In fact, we can invoke the preprocessor directly with the command `cpp` or `clang-cpp`; as noted in the introductory chapter, the preprocessor should be able to locate the required headers, whose paths can be specified in the same way as shown earlier for `cc_` alias.

NOTE The reference implementation does not rely on compiler-specific extensions for providing the ellipsis framework, as is meant to be fully portable with compilers that support C99 (or later). Headers of this framework are placed in the `ellipsis/` subdirectory within the `.include/` directory that houses the reference implementation.

4.1 Constants

Almost all features of ellipsis framework are constrained by preprocessing limits: these decimal integer constants are available as object-like macros named entirely in uppercase and prefixed with `PP_`. They also limit the domain of non-negative integers over which arithmetic, logical, and relational operations can be performed using the preprocessor.

4.1.1 `PP_MAX`

The macro `PP_MAX` expands to the maximum decimal integer constant supported by ellipsis framework; the reference implementation defines `PP_MAX` as 127, which is the smallest upper bound on the number of macro arguments. The behavior is undefined if more than `PP_MAX` arguments are passed to a feature implemented as a function-like macro.

NOTE The reference implementation can be easily modified to support values above 127, up to the maximum number of macro arguments supported by a given C preprocessor; the necessary steps are described in appendix C.

4.1.2 `PP_MAW`

`PP_MAW` expands to the decimal integer constant one less than `PP_MAX`; it is 126 for the reference implementation.

NOTE Macros that require argument counting and invoke `COUNT_` can support at most `PP_MAW` arguments.

4.1.3 `PP_LOG2`

`PP_LOG2` expands to the decimal integer constant equal to the truncated binary logarithm of `PP_MAX`; equivalently, it is one less than the number of significant bits necessary to represent the value of `PP_MAX` in standard binary form.

NOTE The reference implementation defines `PP_LOG2` as 6 ($\lfloor \log_2 127 \rfloor$), which is used by the `LOG_` macro.

4.1.4 `PP_SQRT`

The macro `PP_LOG2` expands to the decimal integer constant equal to the truncated square root of `PP_MAX`.

NOTE The reference implementation defines `PP_LOG2` as 11 ($\lfloor \sqrt{127} \rfloor$), which is used by the `ROOT_` macro.

4.1.5 `PP_INT`

The macro `PP_INT` expands to a list of all positive decimal integer constants less than `PP_MAW`, in decreasing order.

4.1.6 `PP_RANGE`

The macro `PP_RANGE` expands to list of all positive decimal integer constants less than `PP_MAW`, in increasing order.

4.2 Primitives

The ellipsis framework forms the core of the reference implementation, and its architecture is strongly influenced by the microprogramming approach to processor design; in a sense, function-like macros for arithmetic, relational, and logical operations are analogous to micro-programs, which are executed by the preprocessor that acts as a "microcode engine". Each non-trivial computation is performed as a sequence of primitive operations that act as micro-instructions. The elementary operations on macro arguments are done with macros defined in the header `<primitives._>`, which is the topmost root ancestor of the entire header hierarchy for the reference implementation.

NOTE `cat_` and `echo_` are named after UNIX commands; `pop_` and `top_` are named after stack operations.

4.2.1 C_

Definition

```
#define C_
```

Semantics

The macro `C_` expands to empty text. Despite its simplicity, almost all non-trivial macros rely on `C_` directly or indirectly, as it plays a crucial role in deferred expansion, the fundamental mechanism used by the ellipsis framework.

4.2.2 COMMA

Definition

```
#define COMMA ,
```

Semantics

`COMMA` acts as a non-separating comma in argument lists for macro calls, merging multiple arguments into one.

4.2.3 cat_

Definition

```
#define cat_(l, r) l ## r
```

Constraints

The operation shall not generate any invalid preprocessing token.

Semantics

`cat_` pastes its unexpanded arguments verbatim, and the joined section is part of a single preprocessing token.

EXAMPLE Both `cat_(&, =)` and `cat_(0, _)` produce valid preprocessing tokens, but `cat_(&&, =)` causes a constraint violation for the preprocessor, as the hypothetical operator `&&=` is not defined by the C grammar.

4.2.4 echo_

Definition

```
#define echo_(...) __VA_ARGS__
```

Semantics

An invocation of `echo_` expands to the same text as its argument sequence after expanding all *`ACTIVE`* macros.

4.2.5 pop_

Definition

```
#define pop_(t, ...) __VA_ARGS__
```

Semantics

`pop_` removes the first argument from an unexpanded argument list, expanding *`ACTIVE`* macros for the rest.

EXAMPLE `pop_(0 COMMA 0, 5)` expands to `5`, as the unexpanded text `0 COMMA 0` forms a single argument.

4.2.6 top_

Definition

```
#define top_(t, ...) t
```

Semantics

`pop_` expands *`ACTIVE`* macros for the first argument in an unexpanded argument list, and discards the rest.

EXAMPLE `top_(0 COMMA 0, 6)` expands to `0 , 0` as the unexpanded `0 COMMA 0` forms a single argument.

4.3 Deferred expansion

Deferred expansion is main working principle behind the ellipsis framework. A function-like macro is invoked only when the macro name is followed by a left parenthesis (with optional whitespace in between); the invocation is suppressed if this condition is not satisfied, such as by parenthesizing the macro name. As opposed to not invoking a function-like macro at all, deferred expansion simply postpones the macro invocation: one possible approach is to place a transient token between the macro name and the opening parenthesis before the argument sequence.

To illustrate the idea behind deferred expansion, let us consider the `cat_` macro which pastes its two arguments without scanning them for macro expansions. `cat_(cat_(0, 0), 7)` does not work as expected because pasting the arguments verbatim produces `)7` as a bad token. We can get the desired outcome by deferring the outer call as:

```
echo_(cat_ C_(cat_(0, 0), 7))
```

The above expansion works due to a relatively obscure preprocessing rule: the substitution of a function-like macro invocation involves two phases of scanning for macro expansions in the arguments and the replacement text.

- In the first phase, each occurrence of a macro parameter is replaced by its corresponding argument, and if it is not subjected to the operators `#` and `##`, then the argument itself is scanned for macros prior to substitution.
- After substituting the arguments (expanded when applicable), `#` and `##` operations are performed (if present), and the resulting replacement text is scanned from left to right for macro expansions in the second phase.

In our example, the first phase expands the argument `cat_ C_(cat_(0, 0), 7)`: the outer invocation of `cat_` is deferred by the macro `C_`, which expands to the empty text; the second invocation of `cat_` works as usual, producing the token `00`. The expanded argument is substituted in the replacement text of `echo_`, producing the text `cat_(00, 7)`. This text is scanned for macro expansions in the second phase: as the obstruction `C_` was erased during the first phase, the outer invocation of `cat_` takes place as usual, producing the expected outcome `007`.

In summary, our intent in this example was to expand the argument of `cat_` prior to token pasting, as the expansion is suppressed by the `##` operator; to accomplish this, we simply deferred the outer invocation of `cat_` until its arguments were expanded during the first phase (argument substitution). The purpose of `echo_` here is to complete the invocation deferred by `C_`; without that we would end up with a partially expanded text `cat_(00, 7)`. `echo_` and `C_` are nothing special; in fact, we could have used any function-like macro other than `cat_` to perform the full expansion. For instance, `top_(cat_ C_(cat_(0, 0), 7),)` expands to the same text as before: `007`.

NOTE Deferred expansion came to be realized through a slightly different approach with two macros `L` and `R`.

```
#define L (
#define R )
```

With this alternative technique, the earlier example would be written as `echo_(cat_ L cat_(0, 0), 7 R)`.

4.3.1 Liveness

Between argument substitution in first phase and scanning in second phase, two important operations take place:

- For each occurrence of a macro parameter that is subjected to the preprocessing operator `#` or `##` in the replacement text, the corresponding argument is substituted verbatim without scanning for macro expansion: the `#` operator "stringifies" the original text of the argument when the macro was invoked, and the `##` operator pastes its left and right tokens as they appeared when the invocation took place (it may have been deferred).
- The other operation is more relevant to our discussion. At a conceptual level, the preprocessor maintains an internal stack of macros whose replacement text is being scanned. An object-like macro does not have the first phase of argument substitution, so it is pushed to the stack as soon as it is scanned for replacement. During the invocation of a function-like macro, it is pushed to the stack after the first phase is over, and before commencing the second phase. A macro is popped from the stack after completing its ongoing substitution.

This conceptual stack is analogous to the runtime call stack of activation records; for the preprocessor, this stack maintains the sequence of active macro invocations that are yet to be completed, and its significance is that all macros in this stack are rendered "passive". We have used the term "*`ACTIVE`* macros" while describing the primitives, and to understand the distinction between active and passive, we shall introduce the concept of "liveness" states.

A macro is said to be "live" or in *`ACTIVE`* state if it can be expanded: an object-like macro is live before and after its substitution, and a function-like macro remains live throughout the first phase of an invocation, so if the same macro is invoked in one of the arguments, that expansion works as expected (except with `#` and `##`). A macro is said to be "dormant" or in *`PASSIVE`* state if it cannot be expanded: an object-like macro remains dormant throughout its substitution, and a function-like macro is becomes dormant when its invocation enters the second phase.

The C preprocessor does not natively support recursion, so unlike a runtime call stack, the conceptual preprocessor stack always contains unique records of macros whose expansion is underway, and it can be considered as a "set" of dormant macros. At any stage of macro substitution, if a dormant macro is found while scanning, then that occurrence of the macro is marked as *`DEAD`*, meaning that even if the text is rescanned once again after the macro becomes live, an instance marked as *`DEAD`* still cannot be expanded; in other words, the text is retained verbatim.

NOTE *`DEAD`* state is applicable to occurrence of a macro if it is scanned when the macro is in *`PASSIVE`* state.

EXAMPLE The concept of liveness is important to understand iterated composition with deferred expansion, so we shall clarify it with a concrete example that covers all the state transitions. Consider the following invocation:

```
top_(echo_(cat_ C_(echo, _)(0), cat_ top_ C_(,)(echo, _)(0)),)
```

We shall analyze this elaborate macro invocation from the perspective of a C preprocessor. Assuming this is directly part of the source text and not in the replacement text of another macro, initially the stack is empty.

- When `top_` is invoked, its first argument `echo_(cat_ C_(echo, _)(0), cat_ top_ C_(,)(echo, _)(0))` is scanned for replacement (recall that `top_` discards rest of the arguments). `echo_` is then invoked, and its arguments `cat_ C_(echo, _)(0)` and `cat_ top_ C_(,)(echo, _)(0)` are expanded before substitution. Note that the stack is still empty, as both `top_` and `echo_` invocations are currently in their first phase.
- Both occurrences of `C_` are erased (replaced with empty text), and the resulting replacement text of `echo_` is `cat_ (echo, _)(0), cat_ top_ (,)(echo, _)(0)`. The first phase is complete for `echo_`, so it is pushed onto the stack and its second phase begins; `echo_` is now in *`PASSIVE`* state, while `top_` is still in *`ACTIVE`* state.
- While scanning the replacement text of `echo_`, `cat_` is invoked which produces the token `echo_` (assuming `echo` and `_` are not defined as non-trivial object-like macros lest they be replaced in the first phase, just like the `C_` macro). Now `cat_` is pushed onto the stack, so we have two macros in *`PASSIVE`* state: `echo_` and `cat_`.
- While scanning the replacement text of `cat_`, the pasted token `echo_` is found to be in *`PASSIVE`* state, so its occurrence is marked as *`DEAD`*; this completes the expansion of `cat_`, and it is popped off the stack.
- Coming back to the second phase of `echo_`, scanning the replacement text continues, and `top_` is invoked next, which simply expands to the empty text. At the end of its second phase, outcome of the `echo_` invocation is `echo_(0), cat_(echo, _)(0)`. `echo_` is popped and stack becomes empty: all macros are in *`ACTIVE`* state.
- The outcome of `echo_` is nothing but the expanded first argument of `top_`, which is then substituted in its replacement text. The first phase being over, `top_` is moved to *`PASSIVE`* state and its second phase commences.
- Even though `echo_` is currently in *`ACTIVE`* state, the occurrence of `echo_` that was marked as *`DEAD`* earlier is not expanded, and left as it is. `cat_` is invoked next which again produces the token `echo_`, but this time `echo_` is live, so this instance can be invoked later (note that `cat_` itself is pushed and then popped from the stack, with its second phase happening in between, when both `top_` and `cat_` are in *`PASSIVE`* state).
- The expansion of `cat_` is followed by an opening parenthesis, so the pasted token `echo_` gets invoked. The final outcome is the text `echo_(0), 0` (one instance of `echo_` is unexpanded as it had been marked as *`DEAD`*).

NOTE This example uses doubly deferred expansion for the invocation of `cat_` in the second argument of `echo_`. For the sake of brevity, we shall not elaborate on higher levels of deferring, as these are increasingly complex and found to be of limited use; the basic deferred expansion proves insufficient only in very rare circumstances.

4.4 Iterated composition

Iterated composition is arguably the most powerful feature of not just ellipsis framework, but the entire C_ dialect. The function-like macros `o__` and `meta__` are defined in the header `<meta._>` (short for metaprogramming).

4.4.1 `o__`

The fundamental design technique behind iteration composition of function-like macros is to separate text generation from macro invocation, which is postponed with the help of deferred expansion; an actual function can also be used.

Syntax

`o__ (` *function* `,` *exponent* `, (` *argument-list* `) )`

Constraints

exponent shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or expand to such a constant.

Semantics

The macro `o__` produces an iterated function composition text of the form $f(f(\cdots f(arguments)\cdots))$; if *function* is a function-like macro, then it is not invoked. The number of iterations for the composition is given by *exponent*: if it is zero, then *function* is ignored, and the text produced contains only *argument-list* (f^0 is identity function).

NOTE The macro `o__` is named after the mathematical symbol $\circ$ for function composition, such as $f \circ g$.

EXAMPLE The macro `turn_` performs a single left rotation on its expanded argument list, and is defined as:

```
#define turn_(...) pop_(__VA_ARGS__, top_(__VA_ARGS__,))
```

It is generalized by the macro `left_`, which rotates its expanded argument sequence by n positions on the left.

```
#define left_(n, ...) echo_(o__(turn_, n, (__VA_ARGS__)))
```

The invocation of `o__` merely generates the iterated function composition, without invoking the macro `turn_`; for instance, `o__(turn_, 3, (e, u, r, e, k, a))` expands to `turn_(turn_(turn_(e, u, r, e, k, a)))`. To complete each invocation of `turn_`, we need to pass it through another function-like macro that is not used by `turn_` itself: we have chosen `echo_` for this purpose. Recall the two phases during a function-like macro invocation:

- In the first phase, the `echo_` argument `o__(turn_, 3, (e, u, r, e, k, a))` is expanded to `turn_(turn_(turn_(e, u, r, e, k, a)))`, and this is substituted in the replacement text of `echo_`.
- After expanded argument substitution, `echo_` is moved to *`PASSIVE`* state, and the replacement text is scanned left to right, during which `turn_` is invoked: each invocation works as expected because it is performed during the argument substitution phase of `turn_`, when it remains in *`ACTIVE`* state.

Due to the above steps, an invocation `left_(3, e, u, r, e, k, a)` would produce the text `e, k, a, e, u, r`. What makes iterated composition the crown jewel of this dialect is that the function being composed can itself be a function-like macro that uses iterated composition: for example, a function-like macro `f` can be the iterated composition of another function-like macro `g`, whose definition itself can involve iterated composition of another function-like macro `h`, and so on; the nesting of iterated composition can continue up to any depth, as long as the entire expansion does not exhaust resources in the translation environment (memory footprint is a major concern).

Nesting of iterated composition within the function-like macro being iterated works due to two factors: two-phase scanning and separation of text generation from macro invocation. For the given example, text generation occurs in the argument substitution phase of `echo_`, during which the `o__` macro remains in *`PASSIVE`* state; once that is done, `o__` returns to *`ACTIVE`* state, so when the replacement text of `echo_` is scanned in the second phase, `o__` is already live and gets expanded if it is used within the function-like macro whose iterated composition is currently being invoked. The same mechanism works well for further nesting of the depth of `o__` in the full expansion tree of the outermost macro invocation, as opposed to the number of iterations at any particular depth.

A crucial observation is that we need a macro to complete the invocations which were deferred during text generation, and this expander macro itself cannot be used either directly or indirectly by the function being composed. This is because the function-like macro supplied to `o__` as the first argument is invoked in the second phase of the expander macro, during which the latter remains in *`PASSIVE`* state, so if it is invoked by the iterated macro, the expansion fails and each occurrence of the expander macro is retained verbatim; furthermore, all such instances are marked as *`DEAD`*, so they cannot be invoked even if rescanned after the expander macro returns to *`ACTIVE`* state.

Recommended practice

Our example used `echo_` as the expander macro, as it is not used by the iterated macro `turn_`, which only relies on the primitives `pop_` and `top_`. In general, it may be considered a good practice to define a new macro created only to expand the text generated by an invocation of `o__`; such an expander macro can have only a single parameter that occurs as the sole entity in the replacement text, since an invocation of `o__` itself acts as one single argument. This avoids the burden of having to know how the iterated macro works; more precisely, using an existing macro to complete invocation of the iterated macro requires knowledge of the complete expansion tree for the latter, all the way down to the primitives, as one must ensure that the expander macro is not used either directly or indirectly.

Another disadvantage of using an existing macro is tight coupling, as the expander macro can no longer be used by the iterated macro or any of its dependencies in a future refactoring: the text for iterated composition would still be generated correctly, but its expansion would not fully work in the second phase. A workaround is that if an existing macro needs to be introduced while refactoring another macro, it cannot be used directly; instead, its functionality can be achieved by using a newly created macro with the same definition as the existing macro.

NOTE One of the design goals for the reference implementation is economy of names: iterated composition is used to generate preprocessing "microcode" for almost every non-trivial feature of the C_ dialect, and creating a new macro to expand each iterated composition would quickly cause an unnecessary proliferation of macros, almost doubling in number (one extra macro for each use of `o__`). As the definitions of macros are already known, their iterated composition can be invoked with a function-like macro that is not used in the resulting expansion tree.

Knowledge about implementation details can be tempting for minimalists, but many developers may find it beneficial in the long run to prefer our earlier suggestion of creating a separate expander macro for each new feature that is to be implemented using iterated composition; an extra macro can save a lot of hassle in the long game of code maintenance. Before moving on to the next section, we shall complete this discussion by revisiting our previous example: instead of `echo_`, we could have used a separate expander macro `left_c_` as shown below.

```
#define left_(n, ...) left_c_(o__(turn_, n, (__VA_ARGS__)))
#define left_c_(o) o
```

Now we need not worry about the macros used by `turn_`, which can safely use `echo_` without breaking `left_`.

4.4.2 `meta__`

`meta__` is another macro for iterated composition, and if the iterated function happens to be a function-like macro, then the invocations are expanded as well; ironically, this extra convenience makes `meta__` less useful than `o__`.

Syntax

`meta__ (` *function* `,` *exponent* `, (` *argument-list* `) )`

Constraints

exponent shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or expand to such a constant.

Semantics

The macro `meta__` produces an iterated function composition text as generated by `o__`; if *function* is a function-like macro, then it is invoked when `meta__` is in *`PASSIVE`* state. The number of iterations for the composition is given by *exponent*: if it is zero, then *function* is ignored, and the text produced contains only *argument-list*.

NOTE `meta__` has its own expander macro, and if *function* uses `meta__`, those instances are marked as *`DEAD`*.

4.4.3 `o`n`__`

The `o`n`__` family of function-like macros can be used to design `o__`, as is done by the reference implementation.

Syntax

`o`n`__ (` *function* `,` *argument* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`.

Semantics

`o`n`__(f, arg)` repeats the text `f(arg)` n consecutive times, as `f(arg)f(arg)` $\cdots$ `f(arg)` (n times); an invocation of `o0__` expands to blank text. If *function* is a function-like macro, then each invocation is also expanded as well.

NOTE Other than the null macro `o0__`, each `o`n`__(f, a)` is defined as `f(a)o`m`__(f, a)`, where $m = n - 1$.

4.5 Counting arguments

`COUNT_` gives the number of arguments in its expanded argument sequence; it is defined in the header `<count._>`.

Syntax

`COUNT_ (` *argument-list* `)`

Constraints

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

The outcome of an invocation of `COUNT_` is the positive integer constant that is one more than the number of unparenthesized commas in the expanded *argument-list*, indicating the number of arguments after macro expansions.

NOTE Even a blank text is counted as an argument.

4.6 Utilities

The header `<utilities._>` provides a miscellaneous collection of helper macros, listed in the following subsections.

4.6.1 `peel_`

Syntax

`peel_ (` *text* `)`

Semantics

If *text* has a parenthesized prefix, then `peel_` removes its outermost parentheses; otherwise it is not changed.

NOTE `peel_` should be used with caution, as it can cause undesirable changes if *text* is not fully parenthesized, but contains a parenthesized proper prefix: for example, `peel_((1 + 2) * 3)` alters the expression to `1 + 2 * 3`.

4.6.2 `reverse_`

Syntax

`reverse_ (` *argument-list* `)`

Constraints

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

argument-list is first expanded, and the resulting sequence is reversed.

EXAMPLE `PP_RANGE` can be defined as `reverse_(PP_INT)` as the argument `PP_INT` is expanded before reversal.

4.6.3 DEC_

Syntax

`DEC_ (` *n* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

`DEC_(0)` gives `PP_MAX`; if *n* is not zero, then `DEC_(`*n*`)` expands to the decimal integer constant one less than *n*.

4.6.4 INC_

Syntax

`INC_ (` *n* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

`INC_(PP_MAX)` is `0`; if *n* is not `PP_MAX`, then `INC_(`*n*`)` expands to the decimal integer constant one more than *n*.

NOTE `DEC_` and `INC_` are inverses of each other, so both `DEC_(INC_(`*n*`))` and `INC_(DEC_(`*n*`))` expand to *n*.

4.6.5 get_

Syntax

`get_ (` *index* `,` *default-argument* `,` *argument-list* `)`

Constraints

index shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or expand to such a constant.

Semantics

default-argument and a comma are prepended before *argument-list*, and the resulting sequence is expanded; the first argument of the augmented list is removed after expansion, which acts as the default text. After truncating the augmented list, its elements are indexed from 0 left to right, and if an element exists at the given *index*, then that is the outcome; otherwise the default text is produced (first element of the augmented argument list after expansion).

4.6.6 mux_

Syntax

`mux_ (` *selection* `) (` *yes-text* `,` *no-text* `)`

Constraints

selection shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or expand to such a constant.

Semantics

If *selection* is non-zero, then *yes-text* is expanded; otherwise *selection* is zero and *no-text* is expanded.

NOTE `mux_` is named after multiplexer, and it works like a branching instruction in preprocessing code.

4.6.7 repeat__

Syntax

`repeat__ (` *frequency* `,` *text* `)`

Constraints

frequency shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or expand to such a constant.

Semantics

repeat__ creates consecutive replicas of *text* for the number of times specified by *frequency*: an invocation of the form repeat__(n, text) expands to text text $\cdots$ text (n times), which is then scanned for macro expansions.

NOTE If *frequency* is zero, then the outcome is a blank text.

4.6.8 unary_

Syntax

unary_ (n)

Constraints

n shall be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant.

Semantics

unary_(n) is equivalent to the invocation repeat__(n,) which generates a sequence of n consecutive commas.

4.7 Logical operations

The header <logic._> provides function-like macros for fundamental logical operations, whose syntax and semantics are tabulated below; the latter is described in terms of C operators. All these macro invocations expand to 0 or 1.

Constraints

Except for FALSE_ and TRUE_, n shall be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant. If the result can be determined from n, then there is no constraint on m; otherwise m shall also be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant.

Syntax	Semantics	Syntax	Semantics
FALSE_ (n)	0	TRUE_ (n)	1
BOOL_ (n)	!!n	NOT_ (n)	!n
AND_ (n) (m)	n && m	NAND_ (n) (m)	!(n && m)
OR_ (n) (m)	n || m	NOR_ (n) (m)	!(n || m)
XOR_ (n) (m)	!n != !m	XNOR_ (n) (m)	!n == !m
IMPLY_ (n) (m)	!n || m	NIMPLY_ (n) (m)	!(!n || m)
CIMPLY_ (n) (m)	n || !m	CNIMPLY_ (n) (m)	!(n || !m)

Both FALSE_ and TRUE_ act as constant functions that ignore the argument n. Binary operations other than exclusive-OR and exclusive-NOR are described in terms of logical operators in C, whose short-circuit nature is also part of the semantics of these macros: m is processed only if outcome cannot be decided on the basis on n alone.

NOTE An unused argument is entirely discarded, so constraints do not apply and it is not checked for violations. For example, the invocation AND_(NOT_(10), cat_()) expands to 0, even though cat_() would otherwise cause a constraint violation; it goes undetected because the outcome becomes known from NOT_(10), so cat_() is ignored.

4.8 Relational operations

The header <relation._> provides function-like macros for relational operations and macros for finding maximum and minimum. The syntax and semantics are tabulated below; the latter is described in terms of C operators. MAX_ and MIN_ expand to one of the two arguments; rest of the macros for relational operations expand to either 0 or 1.

Constraints

m and n shall be non-negative decimal integer constants not exceeding PP_MAX, or expand to such a constant.

Syntax	Semantics	Syntax	Semantics
GT_ (m , n)	m > n	LE_ (m , n)	m <= n
LT_ (m , n)	m < n	GE_ (m , n)	m >= n
EQ_ (m , n)	m == n	NE_ (m , n)	m != n
MAX_ (m , n)	m >= n ? m : n	MIN_ (m , n)	m <= n ? m : n

4.9 Tools

The header <tools._> for working with argument lists includes two other headers: <knife._> defines macros for slicing through lists, and <wheel._> provides macros for performing both left and right circular rotations on lists.

4.9.1 Slicing

<knife._> provides function-like macros for slicing through lists, facilitating argument isolation and grafting. Macros named with a leading 'x' are complementary to their counterparts without the 'x', which means "excluding".

NOTE head_ and tail_ are named after UNIX commands; they generalize the C_ primitives top_ and pop_.

4.9.1.1 head_

Syntax

head_ (n , *argument-list*)

Constraints

n shall be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant. The number of elements in the expanded *argument-list* shall be less than PP_MAX.

Semantics

Macros in *argument-list* are expanded. If the resulting number of arguments is greater than n, then all arguments appear in the outcome; otherwise only the first n arguments from the expanded list are present in the outcome.

4.9.1.2 xhead_

Syntax

xhead_ (n , *argument-list*)

Constraints

n shall be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant. The number of elements in the expanded *argument-list* shall be less than PP_MAX.

Semantics

Macros in *argument-list* are expanded. If the resulting number of arguments is greater than n, then the outcome is blank; otherwise first n arguments from the expanded list are excluded, and rest of them appear in the outcome.

NOTE xhead_ is complementary to head_, as it includes precisely those elements that are excluded by head_.

4.9.1.3 tail_

Syntax

tail_ (n , *argument-list*)

Constraints

n shall be a non-negative decimal integer constant not exceeding PP_MAX, or it shall expand to such a constant. The number of elements in the expanded *argument-list* shall be less than PP_MAX.

Semantics

Macros in *argument-list* are expanded. If the resulting number of arguments is greater than n, then all arguments appear in the outcome; otherwise only the last n arguments from the expanded list are present in the outcome.

NOTE `tail_(n, ...)` can be implemented via reversal as `reverse_(head_(n, reverse_(__VA_ARGS__)))`.

4.9.1.4 `xtail_`

Syntax

`xtail_ (` *n* `,` *argument-list* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Macros in *argument-list* are expanded. If the resulting number of arguments is greater than n, then the outcome is blank; otherwise last n arguments from the expanded list are excluded, and rest of them appear in the outcome.

NOTE `xtail_(n, ...)` can be implemented via reversal as `reverse_(xhead_(n, reverse_(__VA_ARGS__)))`.

4.9.1.5 `slice_`

Syntax

`slice_ (` *alpha* `,` *omega* `,` *argument-list* `)`

Constraints

Both *alpha* and *omega* shall be non-negative decimal integer constants not exceeding `PP_MAX`, or each of them shall expand to such a constant. The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

argument-list is expanded first, and elements in the resulting list are indexed left to right starting at index zero. Let there be n elements in the expanded list, so that its last element has index $n-1$. If *omega* is not less than n, it is adjusted to $n-1$. If *alpha* exceeds the adjusted value of *omega*, then the outcome is a blank text; otherwise the outcome contains all elements of the expanded list in the index range from *alpha* through *omega* (both inclusive).

NOTE `slice_(alpha, omega, ...)` is equivalent to `xhead_(alpha, head_(INC_(omega), __VA_ARGS__))`.

4.9.1.6 `xslice_`

Syntax

`xslice_ (` *alpha* `,` *omega* `,` *argument-list* `)`

Constraints

Both *alpha* and *omega* shall be non-negative decimal integer constants not exceeding `PP_MAX`, or each of them shall expand to such a constant. The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

argument-list is expanded first, and elements in the resulting list are indexed left to right starting at index zero. Let there be n elements in the expanded list, so that its last element has index $n-1$. If *omega* is not less than n, it is adjusted to $n-1$. If *alpha* exceeds the adjusted value of *omega*, then outcome is the entire expanded list; otherwise elements in the index range from *alpha* through *omega* are excluded, and rest of them appear in the outcome.

NOTE `xslice_` is complementary to `slice_`, as it includes precisely the elements that are excluded by `slice_`.

4.9.1.7 `push_`

Syntax
`push_` (*graft* , *index* , *argument-list*)
Constraints
index shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or shall expand to such a constant.
The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.
Semantics
argument-list is expanded first, and elements in the resulting list are indexed left to right starting at index zero. Let there be n elements in the expanded list, so that its last element has index $n-1$. If *index* exceeds n, then it is adjusted to n. *graft* is injected at the adjusted index position, and if that position is not n (end of the list), then subsequent elements from that index onward (before grafting) are pushed to the right, keeping their existing order.

NOTE Multiple elements can be grafted by wrapping with a macro such as `echo_`, making it a single argument.

EXAMPLE `push_(echo_(u, s), 1, p, h)` expands to the text `p, u, s, h`.

4.9.1.8 `put_`

Syntax
`put_` (*clobber* , *index* , *argument-list*)
Constraints
If *clobber* expands to an unparenthesized list, then the number of elements in that list shall be less than `PP_MAX`.
index shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or shall expand to such a constant.
The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.
Semantics
argument-list is expanded first, and elements in the resulting list are indexed left to right starting at index zero. Let there be n elements in the expanded list, so that its last element has index $n-1$. If *index* exceeds n, then it is adjusted to n. If *clobber* expands to a list of length m, then m elements starting from the adjusted index are replaced by *clobber*; if there are fewer than m elements from the adjusted index, the list is extended on right side.

EXAMPLE `put_(echo_(a, p), 0, m, a, p, l, e)` expands to the text `a, p, p, l, e`.

4.9.2 Rotation

The header `<wheel._>` defines macros for bidirectional rotation of lists, along with a cyclic variation of `get_`.

4.9.2.1 `turn_`

Syntax
`turn_` (*argument-list*)
Semantics
argument-list is expanded then rotated by one element on the left, by moving the first element to the end.

NOTE `turn_(...)` can be implemented using primitive macros as `pop_(__VA_ARGS__, top_(__VA_ARGS__,))`.

4.9.2.2 `left_`

Syntax
`left_` (n , *argument-list*)
Constraints
n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

`left_` is a generalization of `turn_`: *argument-list* is first expanded then it is rotated by n elements on the left.

NOTE `left_(n, ...)` is provided using iterated composition as `echo_(o__(turn_, n, (__VA_ARGS__)))`.

4.9.2.3 `cycle_`

Syntax

`cycle_ (` i `,` *argument-list* `)`

Constraints

i shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

`cycle_` gets the first element after left-rotating expanded *argument-list* by i elements; it treats the list as cyclic.

NOTE `cycle_(i, ...)` is provided using deferred expansion as `echo_(top_ C_(left_(i, __VA_ARGS__),))`.

4.9.2.4 `right_`

Syntax

`right_ (` n `,` *argument-list* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

`right_` is the inverse of `left_`: *argument-list* is first expanded then it is rotated by n elements on the right.

NOTE `right_(n, ...)` can be implemented via reversal as `reverse_(left_(n, reverse_(__VA_ARGS__)))`.

4.10 Selection

`<selection.h>` defines macros for extracting a sub-sequence from a list of arguments, and also for name mangling.

4.10.1 Compression

Syntax

`compress_ ( (` *argument-list* `) ,` *binary-list* `)`

Constraints

Number of elements in the expanded *argument-list* shall be less than `PP_MAX`. Elements in *binary-list* up to the length of expanded *argument-list* are significant: each such element shall expand to the decimal integer constant `0` or `1`; at least one significant element shall be `1`, and excess elements shall not violate preprocessing constraints.

Semantics

An element in the expanded *argument-list* is included in the outcome if the index-wise corresponding element in *binary-list* is `1`; otherwise the associated element in *binary-list* is `0` and the element in *argument-list* is not selected.

binary-list is considered to be circular, so if it is exhausted before *argument-list*, it is read again from the start; excess elements in *binary-list* beyond the length of *argument-list* are unused, but they are still scanned as usual.

NOTE Due to the constraint that *binary-list* shall not be all zeros, at least one element will always be selected.

EXAMPLE `compress_((a, e, i, o, u), 0, 1)` expands to `e, o` as the pattern repeats like `0, 1, 0, 1, ...`

4.10.2 Periodicity

The macros `select_` and `except_` support skipping through elements at regular intervals in an expanded list. They are complementary to each other, as each macro includes precisely those elements that are excluded by the other.

4.10.2.1 Inclusion

Syntax

`select_` (*period* , *argument-list*)

Constraints

period shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or shall expand to such a constant.

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

argument-list is expanded and if *period* is not zero, then the first element in the resulting list is included in the outcome; thereafter every element at an interval of *period* is included (from left to right), and the rest are skipped.

Equivalently, *argument-list* is expanded and then its elements are indexed from the left starting at index 0. The outcome contains precisely those elements whose index is an integer multiple of a non-zero *period*. Index 0 is a multiple of any *period*; however, if *period* is zero, then all elements are skipped and the outcome is a blank text.

EXAMPLE `select_(2, a, e, i, o, u)` expands to `a, i, u`; it is complementary to the outcome of `except_`.

4.10.2.2 Exclusion

Syntax

`except_` (*period* , *argument-list*)

Constraints

period shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or shall expand to such a constant.

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

argument-list is expanded and if *period* is not zero, then the first element in the resulting list is not included in outcome; thereafter every element at an interval of *period* is skipped (from left to right), and the rest are included.

Equivalently, *argument-list* is expanded and then its elements are indexed from the left starting at index 0. The outcome contains precisely those elements whose index is not an integer multiple of a non-zero *period*. Index 0 is a multiple of any *period*; however, if *period* is zero, all elements are included (for *period* 1, all elements are skipped).

EXAMPLE `except_(2, a, e, i, o, u)` expands to `e, o` which is complementary to the outcome of `select_`.

4.10.3 Polymorphism

The `poly_` family of macros can be used to emulate static polymorphism via name mangling on the basis of argument count in an invocation, which can be determined with the `COUNT_` macro (arguments are expanded before counting).

NOTE `poly_` is itself a polymorphic macro based on the same working principle: name mangling.

Syntax

`poly_` (*function-identifier* , *argument-count* [, *suffix* [, *upper-bound*]])
`poly_2_` (*function-identifier* , *argument-count*)
`poly_3_` (*function-identifier* , *argument-count* , *suffix*)
`poly_4_` (*function-identifier* , *argument-count* , *suffix* , *upper-bound*)

Constraints

Pasting the expanded forms of *function-identifier*, *argument-count*, and *suffix* (in that order) shall not produce an invalid token. *upper-bound* shall expand to a decimal integer constant more than 1 and not exceeding `PP_MAX`; if *upper-bound* is given, *argument-count* shall expand to a non-negative decimal integer constant not exceeding `PP_MAX`.

Semantics

`poly_` invokes `poly_`*n*`_` if the expanded argument sequence contains n arguments. `poly_2_` and `poly_3_` paste their expanded arguments in the given order. For `poly_4_`, if *argument-count* is less than *upper-bound*, the outcome is same as `poly_3_`; otherwise *argument-count* is replaced with `0`, and pasting is done the same way as `poly_3_`.

NOTE The `poly_` family merely generates a mangled name that has to be invoked separately. *argument-count* is typically supplied via `COUNT_`, whose outcome is always non-zero. For `poly_4_`, the special argument count of zero is used to designate a default function/macro meant to be invoked when argument count exceeds a certain value; in other words, a function/macro should be available for each supported argument count below *upper-bound*.

EXAMPLE `RANGE_(...)` can be implemented as `poly_(RANGE_, COUNT_(__VA_ARGS__), _)(__VA_ARGS__)`.

4.10.3.1 Disadvantages

Generalization achieved with the use of polymorphic macros has certain fundamental limitations in the design itself.

- The preprocessing "microcode" of `COUNT_` uses an incrementing strategy to count the arguments one by one, which incurs a small transpilation overhead; in fact, any other implementation of static polymorphism based on argument counting will have some name resolution overhead, even though it may be considered insignificant.
- If *function-identifier* is defined as a non-trivial object-like macro, then pasting the expanded arguments would not produce the required name. For example, if a polymorphic macro `fn_(...)` is meant to invoke `fn2` or `fn3`, the definition `poly_(fn, COUNT_(__VA_ARGS__))(__VA_ARGS__)` has a subtle issue: if `fn` is defined as an object-like macro that expands to `func`, then `fn_(0, 1)` would expand to `func2(0, 1)` instead of `fn2(0, 1)`.
- The same issue also affects *suffix*: for example, the reference implementation does not use `__` directly as *suffix* with `poly_3_`, as `__` is a reserved identifier in C, and compilers can define it as a non-trivial object-like macro, which would result in an incorrect name mangling when `__` expands to something else before token pasting.
- When a polymorphic macro invokes another macro obtained with name mangling, the latter receives expanded arguments that may look unrecognizable when stringified. Many debuggable features of C_ are polymorphic in nature (such as `assert_`), and it is desirable that the original text is printed in a diagnostic message. The reference implementation takes extra precautions to retain the original argument text in a diagnostic message.

Due to these subtleties, the reference implementation only provides, but rarely makes use of polymorphic macros: avoiding argument counting overhead in the source code leads to a noticeable improvement in preprocessing time.

4.10.4 Ranges

The `RANGE_` family generates an arithmetic sequence of non-negative decimal integer constants not exceeding `PP_MAX`.

Syntax

`RANGE_` (*stop*)
`RANGE_` (*alpha* , *omega* [, *delta*=1])
`RANGE_1_` (*stop*)
`RANGE_2_` (*alpha* , *omega*)
`RANGE_3_` (*alpha* , *omega* , *delta*)

Constraints

Each argument shall be a non-negative decimal integer constant not above `PP_MAX`, or expand to such a constant.

Semantics

`RANGE_` invokes `RANGE_`*n*`_` if the expanded argument sequence contains n arguments.

RANGE_1_ generates the sequence of non-negative integers less than *stop*, in increasing order (starting from zero).

RANGE_2_ generates the sequence of integers from *alpha* through *omega* (both inclusive): if *alpha* is less than *omega*, then the sequence is in increasing order (in steps of 1), and decreasing order otherwise (with step value -1).

RANGE_3_ generates an arithmetic sequence starting from *alpha*, with *delta* as the step value: if *alpha* is greater than *omega*, then the step value is subtracted each time. Last element of the sequence does not go beyond *omega*: if *omega* can be obtained by repeatedly adding or subtracting *delta* from *alpha*, then *omega* is the last element.

NOTE The sequence is empty only if *stop* is zero; if *alpha* is equal to *omega*, then the sequence is a singleton.

4.11 Arithmetic operations

The header <abacus._> for arithmetic operations includes two other headers: <additive._> provides macros for addition and subtraction; <multiplicative._> defines macros for multiplicative, exponential, and shift operations. Their syntax and semantics are grouped and tabulated below; the semantics are summarized under "Op" columns.

Constraints

For each macro, both arguments shall expand to non-negative decimal integer constants not exceeding PP_MAX.

Syntax and semantics

Except for MINUS_, the outcome is a non-negative decimal integer constant not exceeding PP_MAX, or blank text.

Additive		Multiplicative		Exponential		Shift	
Syntax	**Op**	**Syntax**	**Op**	**Syntax**	**Op**	**Syntax**	**Op**
ADD_ (m , n)	$m + n$	MUL_ (n , d)	$n * d$	POW_ (n , p)	n^p	LSH_ (n , s)	$n \ll s$
SUB_ (m , n)	$m - n$	DIV_ (n , d)	n / d	LOG_ (n , b)	$\lfloor \log_b n \rfloor$	RSH_ (n , s)	$n \gg s$
DIF_ (m , n)	$\lvert m - n \rvert$	MOD_ (n , d)	$n \% d$	ROOT_ (n , p)	$\lfloor \sqrt[p]{n} \rfloor$		
MINUS_ (m , n)	$[-]\lvert m - n \rvert$						

Both ADD_ and SUB_ perform modular wraparound similar to unsigned arithmetic, so that their final outcome never exceeds PP_MAX: if the result of addition is greater than PP_MAX, then PP_MAX + 1 is subtracted; if the result of subtraction is negative, then PP_MAX + 1 is added. After these adjustments, the outcome is in the domain of preprocessing integers, making it suitable for further computations with arithmetic, logical, or relational macros.

DIF_ computes the absolute difference between its arguments; MINUS_ puts a minus sign before the outcome of DIF_ iff its first argument is greater than the second: in that case, the outcome is not a decimal integer constant.

DIV_ and MOD_ respectively compute the quotient and remainder; the outcome is blank iff the divisor d is zero.

For MUL_, POW_, and LSH_, the outcome is a blank text iff the true mathematical result is greater than PP_MAX.

LOG_ and ROOT_ apply floor function on the true mathematical result, so any fractional part gets discarded.

4.12 Templates

The header <templates._> defines macros for higher order functions that iterate over lists and perform an operation on each element; the operation can be a function call, a function-like macro invocation, or even ordinary C operators.

4.12.1 omni__

omni__ is perhaps the most versatile higher order function in C_, as the other iterator macros can be implemented with it; however, it is not as powerful as iterated composition, since a function-like macro given to omni__ cannot itself invoke omni__ or any other macro implemented with it. The superiority of o__ is due to its support of nested composition, where a function-like macro being iterated can itself iterate over another function-like macro using o__, and this pattern can continue up to any depth that does not exhaust resources of the translation environment.

Syntax

`omni__ (` *left-arg* `,` *f* `,` *right-list* `)`

Constraints

Number of elements in the expanded *right-list* shall be less than `PP_MAX`, after removal of optional parentheses.

If *left-arg* expands to a list with multiple elements, the same constraint shall also apply to its element count.

Semantics

The `peel_` macro is applied on *left-arg* and *right-list*, and their results are expanded. Both *left-arg* and f are single arguments that can expand to a list with multiple elements; the third argument onward are part of *right-list*. *left-arg* and *right-list* can be optionally parenthesized: each parenthesized list acts as a single argument, and the outermost parentheses are removed by `peel_`. Length of the longer expanded list determines number of iterations.

The lists are scanned left to right starting from their first elements. In each iteration, f is invoked with two arguments: current element of *left-arg* is the first argument, and current element of *right-arg* is the second argument. Current element of *left-arg* is replaced with the outcome of current invocation, and the index moves one step forward in each list for the next iteration. Whenever the shorter list ends, its index is moved back to the beginning; in other words, the shorter list is logically circular. If f expands to a list of functions, then that list is also considered circular, and each function is invoked in a round robin sequence starting from the left; if the number of functions is more than the number of iterations, then the extra functions remain unused, but all of them are scanned as usual.

Formally, let the elements of *left-arg* be labeled as l_0 , $\cdots$, l_{L-1}; similarly, the elements of *right-list* are labeled as r_0 , $\cdots$, r_{R-1}. L and R denote the number of elements, and let N denote maximum of the two. Considering f to be a list, let its elements be labeled in the same way as f_0 , $\cdots$, f_{F-1}, where F denotes the number of functions. With these notations, we can denote an invocation of `omni__` with the following form (`echo_` is only an example):

`omni__ ( (` l_0 `,` $\cdots$ `,` l_{L-1} `) , echo_ (` f_0 `,` $\cdots$ `,` f_{F-1} `) ,` r_0 `,` $\cdots$ `,` r_{R-1} `)`

An invocation of the above form performs N iterations, generating a comma-separated list of the following form:

f_0 `(` l_0 `,` r_0 `) ,` $f_{1\%F}$ `(` $l'_{1\%L}$ `,` $r_{1\%R}$ `) ,` $\cdots$ `,` $f_{(N-1)\%F}$ `(` $l'_{(N-1)\%L}$ `,` $r_{(N-1)\%R}$ `)`

Only the rightmost L elements of the above list are retained in the final output list: each element is of the form $f_{i\%F}$ `(` $l'_{i\%L}$ `,` $r_{i\%R}$ `)`, where i ranges from $N-L$ through $N-1$, and % gives the remainder after division. l' denotes the current element in *left-arg*, which is updated in each iteration: initially, each l' is same as the original element in *left-arg*, and whenever it is used in an iteration, it is replaced with the element generated in that iteration. In each case, if $f_{i\%F}$ is a function-like macro other than `omni__`, it is invoked while `omni__` remains in ***`PASSIVE`*** state.

NOTE If *left-arg* contains fewer elements than *right-list*, intermediate results get used in subsequent iterations. If a function-like macro in f produces the name `omni__`, that instance is marked as ***`DEAD`*** and cannot be invoked.

EXAMPLE I When both sides have equal number of elements, `omni__` pairs each element of *left-arg* with the index-wise corresponding element in *right-list*, and invokes (an element of) f with that pair of arguments. For instance, `omni_((di, sa), cat_, (na, ur))` expands to `cat_(di, na), cat_(sa, ur)`, and then `dina, saur`. Deferred expansion works as usual, so `echo_(cat_ C_(omni__((di, sa), cat_, (na, ur))))` produces `dinasaur`.

EXAMPLE II All elements of *right-list* can be aggregated with a single element of *left-arg*. For instance, `omni__(0, ADD_, RANGE_(11, 20, 2))` computes the sum of all odd numbers from 11 to 20, which is 75.

EXAMPLE III The element in a singleton *right-list* can be applied to each element of *left-arg*. For instance, `omni__(RANGE_(10), echo_(ADD_, MUL_), 2)` iterates over integers from 0 through 9, adding 2 to each even number and multiplying each odd number by 2: the resulting text is `2, 2, 4, 6, 6, 10, 8, 14, 10, 18`.

Recommended practice

We shall reiterate our earlier cautionary note on the use of `peel_` macro: if the argument to `peel_` is not fully parenthesized but contains a parenthesized proper prefix, then `peel_` can silently change the meaning of a code. For example, the text `(a + b) * c` should never be used as *left-arg* or *right-list*, as `peel_` will alter it to `a + b * c`.

A good practice is to fully enclose the actual list within a single pair of parentheses, which is removed by `omni__` using `peel_`; often it will be redundant, but this habit can help to avert potentially disastrous changes by `peel_`.

4.12.2 `gate__`

Due to the unusual syntax of function-like macros for logical operations, they cannot be directly used with `omni__`. Those macros are modeled after logic gates, and the `gate__` iterator can be used to apply them over two lists.

Syntax

`gate__` `(` *left-arg* `,` *f* `,` *right-list* `)`

Constraints

`gate__` shall have precisely the same constraints as those applicable for `omni__`.

Semantics

`gate__` generates a list similar to `omni__`, except that element at index i is of the form $f_{i\%F}$ `(` $l'_{i\%L}$ `)` `(` $r_{i\%R}$ `)`; each symbol has the same meaning as described for `omni__`. Output contains last L elements of the following list:

$$f_0 \texttt{ (} l_0 \texttt{) (} r_0 \texttt{) , } f_{1\%F} \texttt{ (} l'_{1\%L} \texttt{) (} r_{1\%R} \texttt{) , } \cdots \texttt{ , } f_{(N-1)\%F} \texttt{ (} l'_{(N-1)\%L} \texttt{) (} r_{(N-1)\%R} \texttt{)}$$

EXAMPLE `gate__((0, 1), XNOR_, (2, 3))` generates `XNOR_(0)(2), XNOR_(1)(3)` that expands to `0, 1`.

4.12.3 `FILTER_`

Syntax

`FILTER_` `(` *predicate* `,` *arg2* `,` *argument-list* `)`

Constraints

Each element in *predicate* shall be a function-like macro that can be invoked with two arguments, and its outcome shall be `0` or `1` when the first argument is from *argument-list* and the second from *arg2*, as described in semantics.

If *arg2* is a list with multiple elements after expansion, the resulting element count shall be less than `PP_MAX`.

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

For each element in the expanded *argument-list*, a function-like macro from *predicate* is invoked with that element as the first argument, and the second argument is an element from *arg2* after peeling and expansion of *arg2*: if outcome is `1`, then the element from *argument-list* is included; otherwise outcome is `0` and the element is excluded.

If *predicate* or *arg2* expands to a list with multiple elements, the elements are used in a round robin sequence, moving back to the first element each time the list ends; extra elements beyond the number of iterations are unused. *arg2* is also subjected to `peel_`, which removes the outermost parentheses from an optionally parenthesized prefix.

EXAMPLE The invocation `FILTER_(echo_(MOD_, LT_), (2, 10), 2, 3, 5, 7, 11, 13)` extracts all odd integers at even indices, and single digits at odd indices; the resulting expansion generates the text `3, 5, 7, 11`.

NOTE As per our naming conventions, a function-like macro named in uppercase suggests that its outcome is a list of constants. The naming of `FILTER_` is an exception, since its outcome need not be a list of constants. The name `filter_` is already used for a similar iterator that works with runtime arrays, so the metaprogramming variant was named as `FILTER_`; moreover, `FILTER_` is closely associated with `SEARCH_`, so the naming is justified.

4.12.4 `SEARCH_`

Syntax

`SEARCH_` `(` *predicate* `,` *arg2* `,` *argument-list* `)`

Constraints

`SEARCH_` shall have precisely the same constraints as those applicable for `FILTER_`.

Semantics

`SEARCH_` generates an index list for elements in the expanded *argument-list* that would be selected by `FILTER_`.

NOTE As always, the expanded *argument-list* is indexed from left to right starting at index zero.

EXAMPLE `SEARCH_(EQ_, 10, 0, 10, 1, 10)` generates `1, 3` which is the list of indices at which ten is found.

4.12.5 `REL_`

Primary use of the `REL_` family is element-wise comparison of two lists, and to check for ordering in a single list; in other words, `REL_`, `REL_0_`, and `REL_2_` extend the application of relational operations over lists of constants.

Syntax

`REL_` (*left-arg* , *f* , *right-list*)
`REL_` (*f* , *list-arg*)
`REL_0_` (*left-arg* , *f* , *right-list*)
`REL_2_` (*f* , *list-arg*)

Constraints

Number of elements in *list-arg* or *right-list* shall be less than `PP_MAX`, after removal of optional parentheses.

If *left-arg* expands to a list with multiple elements, the same constraint shall also apply to its element count.

f shall be a single argument that expands to an optionally parenthesized list with less than `PP_MAX` elements; *f* needs to be parenthesized for `REL_` if the unparenthesized form of *f* expands to more than one element. If an element of *f* is used in output, then it shall be a function-like macro that can be invoked as *f* (*l* , *r*), where *l* and *r* are respectively elements from *left-arg* and *right-list* for `REL_0_`; for `REL_2_`, *l* and *r* can be any pair of consecutive elements from *list-arg*. Each invocation shall produce a non-negative decimal integer constant not exceeding `PP_MAX`.

NOTE *list-arg* means a single argument that expands to a list after removing optional parentheses with `peel_`.

Semantics

Arguments to `REL_` are counted after excluding the first argument in its unexpanded form, and rest of the arguments are expanded: if the count is 1, then `REL_2_` is invoked; otherwise the count is more than one, and `REL_0_` is invoked. `REL_` considers *f* to be a singleton list: it should not expand to more than one element, as rest of them are considered to be the initial elements of *right-list* (*f* should be parenthesized if it has multiple elements).

left-arg, *right-list*, and *list-arg* are peeled and expanded in the way described in the semantics of `omni__`.

f is also subjected to the `peel_` macro, and let F be the number of elements after peeling and expansion.

- `REL_0_` (*left-arg* , *f* , *right-list*)

Let L and R be the number of elements in *left-arg* and *right-list* respectively; let N be their maximum value.

Outcome of `REL_0_` is a logical conjunction that is equivalent to the expansion of a text with the following form:

`gate__ ( 1 , AND_ ,` f_0 (l_0 , r_0) , $f_{1\%F}$ ($l_{1\%L}$, $r_{1\%R}$) , $\cdots$, $f_{(N-1)\%F}$ ($l_{(N-1)\%L}$, $l_{(N-1)\%R}$))

In each invocation of the form $f_{i\%F}$ ($l_{i\%L}$, $r_{i\%R}$), $l_{i\%L}$ and $r_{i\%R}$ denote elements from *left-arg* and *right-list*.

- `REL_2_` (*f* , *list-arg*)

If *list-arg* expands to a singleton list, then the outcome of `REL_2_` is `1`; otherwise, let N be the number of elements in *list-arg* after peeling and expansion. When N is above 1, `REL_2_` performs the following computation:

`gate__ ( 1 , AND_ ,` f_0 (l_0 , l_1) , $f_{1\%F}$ (l_1 , l_2) , $\cdots$, $f_{(N-2)\%F}$ (l_{N-2} , l_{N-1}))

In other words, `REL_2_` computes the logical conjunction of a series of macro invocations, each one having the form $f_{i\%F}$ (l_i , l_{i+1}); here l_i and l_{i+1} are a pair of consecutive elements from *list-arg* after peeling and expansion.

NOTE If a list contains fewer elements than the number of iterations, its elements are reused in a round robin sequence. Despite the "short-circuit" semantics of `AND_`, all the invocations are performed in intermediate steps.

EXAMPLE I `REL_(RANGE_(0, 7, 2), LT_, RANGE_(1, 7, 2))` checks if each element of left list is less than the corresponding element in right list; outcome is `1` for the given lists expanding to `0, 2, 4, 6` and `1, 3, 5, 7`.

EXAMPLE II Both `REL_((1, 2, 3, 4, 5), LT_, 10)` and `REL_(10, GT_, 6, 7, 8, 9, 10)` check if each element of the given list is less than ten; the outcome is `1` for the first list and `0` for the second (`GT_(10, 10)` is `0`).

EXAMPLE III `REL_(LT_, (0, 1, 1))` tests if the list is monotonic increasing, whereas `REL_(LE_, (0, 1, 1))` relaxes the ordering to non-decreasing; for the given list `(0, 1, 1)`, the outcome is `0` with `LT_`, and `1` with `LE_`.

4.12.6 `glue__`

Syntax

`glue__` (*separator* , *argument-list*)

Constraints

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Each argument separating comma in the expanded *argument-list* is replaced with *separator*, forming one element.

4.12.7 `map__`

`map__` implements the higher order function *map* for element-wise transformation of a list.

Syntax

`map__` (f , *argument-list*)

Constraints

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Let the expanded form of f be labeled as f_0 , $\cdots$, f_{F-1}; similarly, the expanded form of *argument-list* is denoted as a_0 , $\cdots$, a_{N-1}. With this notation, the output of `map__` is represented as a list of the following form:

$$f_0 \texttt{ (} a_0 \texttt{) } , f_{1\%F} \texttt{ (} a_1 \texttt{) } , \cdots , f_{(N-1)\%F} \texttt{ (} a_{N-1} \texttt{)}$$

Each element of the output list has the form $f_{i\%F}$ (a_i); if $f_{i\%F}$ is a function-like macro, then it is expanded.

NOTE `map__` can be used to parenthesize elements of a list simply by leaving f as blank (comma is required).

EXAMPLE `map__(echo_(INC_, DEC_), RANGE_(10))` increments each integer at even indices, and decrements those at odd indices; for the given list `RANGE_(10)`, the generated text is `1, 0, 3, 2, 5, 4, 7, 6, 9, 8`.

4.12.8 `wrap__`

`wrap__` is a variant of `map__` that can be used when each element of the argument list is already parenthesized; furthermore, each output element is also fully parenthesized, so the resulting text is a list of parenthesized elements.

Syntax

`wrap__` (f , *argument-list*)

Constraints

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Let the expanded form of f be labeled as f_0 , $\cdots$, f_{F-1}; similarly, the expanded form of *argument-list* is denoted as a_0 , $\cdots$, a_{N-1}. With this notation, the output of `wrap__` is represented as a list of the following form:

$$\texttt{(} f_0\ a_0 \texttt{) } , \texttt{ (} f_{1\%F}\ a_1 \texttt{) } , \cdots , \texttt{ (} f_{(N-1)\%F}\ a_{N-1} \texttt{)}$$

Each element of the output list has the form ($f_{i\%F}$ a_i); if $f_{i\%F}$ is a function-like macro and a_i contains a parenthesized prefix, then the macro is invoked (assuming it is in *`ACTIVE`* state).

EXAMPLE `wrap__(top_, (0, 1), (2, 3), (4, 5))` extracts the first element from each pair, producing the output text `(0), (2), (4)`; the input pairs can be generated with `omni__(RANGE_(0, 5, 2),, RANGE_(1, 5, 2))`.

4.12.9 `fold__`

`fold__` implements the higher order function *fold* for combining multiple elements of a list into a single result using an aggregation function that can accept two arguments: if the list contains more elements, the function is called again with the result of its previous invocation as one argument, the other argument being the next input element.

Syntax

`fold__` (f, *argument-list*)

Constraints

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Let the expanded form of f be labeled as f_0 , $\cdots$, f_{F-1}; similarly, the expanded form of *argument-list* is denoted as a_0 , $\cdots$, a_{N-1}. With this notation, the output of `fold__` can be represented with the following form:

$$f_{(N-2)\%F} \texttt{ (} f_{(N-3)\%F} \texttt{ (} \cdots f_{2\%F} \texttt{ (} f_{1\%F} \texttt{ (} f_0 \texttt{ (} a_0 \texttt{ , } a_1 \texttt{) } \texttt{ , } a_2 \texttt{) } \texttt{ , } a_3 \texttt{) } \cdots \texttt{ , } a_{N-2} \texttt{) } \texttt{ , } a_{N-1} \texttt{) }$$

In other words, if *argument-list* expands to a list of N elements ($N > 1$), then there are $N - 1$ iterations: intermediate results are used in subsequent stages until all elements are combined into a single result. The first invocation is of the form f_0 (a_0 , a_1); if the list has more than two elements, then the result of each invocation is used as the first argument in the next invocation, and the next unprocessed element is used as the second argument.

NOTE `fold__` aggregates the expanded *argument-list* by processing its elements from left to right.

EXAMPLE `fold__(echo_(ADD_, SUB_), RANGE_(10))` alternately adds and subtracts non-negative integers below 10; in effect, it subtracts the sum of odd integers from the sum of even integers in given range, resulting in `5`.

4.12.10 `reduce__`

`reduce__` is also an aggregating macro similar to `fold__`, except that the elements are combined right to left.

Syntax

`reduce__` (f, *argument-list*)

Constraints

The number of arguments in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Let the expanded form of f be labeled as f_0 , $\cdots$, f_{F-1}; similarly, the expanded form of *argument-list* is denoted as a_0 , $\cdots$, a_{N-1}. With this notation, the output of `fold__` can be represented with the following form:

$$f_{(N-2)\%F} \texttt{ (} a_0 \texttt{ , } f_{(N-3)\%F} \texttt{ (} a_1 \texttt{ , } \cdots f_{2\%F} \texttt{ (} a_{N-4} \texttt{ , } f_{1\%F} \texttt{ (} a_{N-3} \texttt{ , } f_0 \texttt{ (} a_{N-2} \texttt{ , } a_{N-1} \texttt{))) } \cdots \texttt{)) }$$

The first invocation is of the form f_0 (a_{N-2} , a_{N-1}); thereafter the result of each invocation is used as the second argument in the next invocation, and the previous unprocessed element is used as the first argument.

NOTE Even though `reduce__` processes its *argument-list* right to left, the function list f is used left to right.

EXAMPLE `reduce__(SUB_, 3, 2, 1)` performs `SUB_(3, SUB_(2, 1))`, whose outcome is `2`. On the contrary, `fold__(SUB_, 3, 2, 1)` works left to right as `SUB_(SUB_(3, 2), 1)`, and the result is `0`.

4.12.11 `op__`

The `op__` family is modeled after the semantics of `omni__`, except that `op__` is intended to be used with operators. Each element in the operand list(s) is parenthesized, and each element in the output list is also fully parenthesized.

Syntax

`op__` (*left-arg* , *op* , *right-list*)
`op__` (*op* , *list-arg*)
`op__0__` (*left-arg* , *op* , *right-list*)
`op__2__` (*op* , *list-arg*)

Constraints

Number of elements in *list-arg* or *right-list* shall be less than `PP_MAX`, after removal of optional parentheses.

If *left-arg* expands to a list with multiple elements, the same constraint shall also apply to its element count.

op shall be a single argument that expands to an optionally parenthesized list with less than `PP_MAX` elements; *op* needs to be parenthesized for `op__` if the unparenthesized form of *op* expands to more than one element.

NOTE *list-arg* means a single argument that expands to a list after removing optional parentheses with `peel_`.

Semantics

Arguments to `op__` are counted after excluding the first argument in its unexpanded form, and rest of the arguments are expanded: if the count is 1, then `op__2__` is invoked; otherwise the count is more than one, and `op__0__` is invoked. `op__` considers *op* to be a singleton list: it should not expand to more than one element, as rest of them are considered to be initial elements of *right-list* (*op* should be parenthesized if it has multiple elements).

left-arg, *right-list*, and *list-arg* are peeled and expanded in the way described in the semantics of `omni__`.

op is also subjected to the `peel_` macro, and let P be the number of elements after peeling and expansion.

- `op__0__ (` *left-arg* `,` *op* `,` *right-list* `)`

After peeling *left-arg*, each of its elements is parenthesized, and let the resulting list be labeled as l_0 , $\cdots$, l_{L-1}; similar steps are also performed for *right-list*, whose parenthesized elements can be labeled as r_0 , $\cdots$, r_{L-1}. Note that l_i and r_i represent elements after they are parenthesized by `op__0__`. Let N be the maximum of L and R. `op__0__` generates a list of the following form: its rightmost L elements are parenthesized and retained in output.

$$l_0 \; op_0 \; r_0 \;,\; l'_{1\%L} \; op_{1\%P} \; r_{1\%R} \;,\; \cdots \;,\; l'_{(N-1)\%L} \; op_{(N-1)\%P} \; r_{(N-1)\%R}$$

l' denotes the current element in *left-arg*, which is updated in each iteration: initially, each l' is same as the parenthesized element from *left-arg*, and whenever it is used in an iteration, it is replaced with the new element.

NOTE A newly generated element is not immediately parenthesized to avoid a proliferation of excessive parentheses in intermediate steps, whenever *left-arg* is shorter than *right-list* and generated elements are used in subsequent iterations. Only after the entire output list has been generated, its last L elements are parenthesized.

- `op__2__ (` *op* `,` *list-arg* `)`

After peeling *list-arg*, let its expanded elements be labeled as l_0 , $\cdots$, l_{N-1}; output of `op__2__` has the form:

$$(\; op_0 \;(\; l_0 \;)\;) \;,\; (\; op_{1\%P} \;(\; l_1 \;)\;) \;,\; \cdots \;,\; (\; op_{(N-1)\%P} \;(\; l_{N-1} \;)\;)$$

Equivalently, the output is a list of N parenthesized elements, each having the form $(\; op_{i\%P} \;(\; l_i \;)\;)$ $(0 \le i < N)$.

EXAMPLE I When both sides have equal number of elements, `op__2__` pairs each element of *left-arg* with the index-wise corresponding element in *right-list*, and places (an element of) *op* in between them. For instance, `op__((0 + 1, 2 + 3), (*, /), (4 + 5, 6 + 7))` expands to `((0 + 1) * (4 + 5)), ((2 + 3) / (6 + 7))`.

EXAMPLE II All elements of *right-list* can be aggregated with a single element of *left-arg*. For instance, `op__(0, +, a, b, c)` generates the parenthesized expression `((0) + (a) + (b) + (c))`.

EXAMPLE III The element in a singleton *right-list* can be applied to each element of *left-arg*. For instance, `op__((i, j, k), *, p + q)` generates the list `((i) * (p + q)), ((j) * (p + q)), ((k) * (p + q))`.

EXAMPLE IV `op__2__` can be used to apply a unary operator to each element of a list. For instance, `op__(-, (x + 1, y + 2, z + 3))` gives the negation of each element as `(-(x + 1)), (-(y + 2)), (-(z + 3))`.

4.12.12 `rel__`

The `rel__` family is modeled after the semantics of `REL_`, except that `rel__` is intended to be used with C relational operators, and in most cases, the outcome is not a decimal integer constant (depending on the operands, it can be an integer constant expression). Each element in the operand list(s) is parenthesized (as done by `op__`), and the outcome is a parenthesized logical conjunction of expressions: it is of type `Int_` and evaluates to either zero or one.

Syntax

rel__ (*left-arg* , *op* , *right-list*)
rel__ (*op* , *list-arg*)
rel__0__ (*left-arg* , *op* , *right-list*)
rel__2__ (*op* , *list-arg*)

Constraints

The rel__ family shall have precisely the same constraints as those applicable for the op_ family.

Semantics

Arguments to rel__ are counted after excluding the first argument in its unexpanded form, and rest of the arguments are expanded: if the count is 1, then rel__2__ is invoked; otherwise the count is more than one, and rel__0__ is invoked. rel__ considers *op* to be a singleton list: it should not expand to more than one element, as rest of them are taken to be initial elements of *right-list* (*op* should be parenthesized if it has multiple elements).

left-arg, *right-list*, and *list-arg* are peeled and expanded in the way described in the semantics of omni__.

op is also subjected to the peel_ macro, and let P be the number of elements after peeling and expansion.

- rel__0__ (*left-arg* , *op* , *right-list*)

Let L and R be the number of elements in *left-arg* and *right-list* respectively; let N be their maximum value.

Outcome of rel__0__ is a parenthesized expression that computes a logical conjunction of the following form:

$$\texttt{(}\ \texttt{(}\ l_0\ \texttt{)}\ op_0\ \texttt{(}\ r_0\ \texttt{)}\ \texttt{\&\&}\ \texttt{(}\ l_{1\%L}\ \texttt{)}\ op_{1\%P}\ \texttt{(}\ r_{1\%R}\ \texttt{)}\ \texttt{\&\&}\ \cdots\ \texttt{\&\&}\ \texttt{(}\ l_{(N-1)\%L}\ \texttt{)}\ op_{(N-1)\%P}\ \texttt{(}\ r_{(N-1)\%R}\ \texttt{)}\ \texttt{)}$$

In each element of the form ($l_{i\%L}$) $op_{i\%P}$ ($r_{i\%R}$), $l_{i\%L}$ and $r_{i\%R}$ denote elements from *left-arg* and *right-list*.

- rel__2__ (*op* , *list-arg*)

If *list-arg* expands to a singleton list, then the outcome of rel__2__ is 1; otherwise, let N be the number of elements in *list-arg* after peeling and expansion. When N is above 1, rel__2__ generates the following expression:

$$\texttt{(}\ \texttt{(}\ l_0\ \texttt{)}\ op_0\ \texttt{(}\ l_1\ \texttt{)}\ \texttt{\&\&}\ \texttt{(}\ l_1\ \texttt{)}\ op_{1\%P}\ \texttt{(}\ l_2\ \texttt{)}\ \texttt{\&\&}\ \cdots\ \texttt{\&\&}\ \texttt{(}\ l_{N-2}\ \texttt{)}\ op_{(N-2)\%P}\ \texttt{(}\ l_{N-1}\ \texttt{)}\ \texttt{)}$$

In other words, rel__2__ evaluates the logical conjunction of a series of expressions, each one having the form (l_i) $op_{1\%P}$ (l_{i+1}); here l_i and l_{i+1} are a pair of consecutive elements from *list-arg* after peeling and expansion.

NOTE If a list contains fewer elements than the number of iterations, its elements are reused in a round robin sequence. Due to the "short-circuit" semantics of &&, the left to right evaluation stops at the first zero expression.

EXAMPLE I rel__((a, b, c), <=, (x, y, z)) evaluates to true iff each element of left list is less than the corresponding element in right list; the generated expression is ((a) <= (x) && (b) <= (y) && (c) <= (z)).

EXAMPLE II Both rel__((a, b, c), <, 10) and rel__(10, >, x, y, z) evaluate to true iff each element of the given list is less than ten; the generated expressions are ((a) < (10) && (b) < (10) && (c) < (10)) and ((10) > (x) && (10) > (y) && (10) > (z)) respectively.

EXAMPLE III rel__(<=, (a, b, c, d)) evaluates to true iff the list is non-decreasing; for the given list, the generated expression is ((a) <= (b) && (b) <= (c) && (c) <= (d)).

4.13 Additional utilities

<ellipsis._> includes all other headers of the ellipsis framework, and it also provides some additional utility macros for metaprogramming: these are function-like macros for non-trivial computations and list transformations.

4.13.1 GCD

Syntax

`GCD_ (` *m* `,` *n* `)`

Constraints

m and *n* shall be non-negative decimal integer constants not exceeding `PP_MAX`, or expand to such a constant.

Semantics

Outcome is the decimal integer constant that is the greatest common divisor of *m* and *n*.

NOTE `GCD_(0, 0)` is `0`.

4.13.2 Primality

Syntax

`IS_PRIME_ (` *n* `)`

Constraints

n shall be a non-negative decimal integer constant not exceeding `PP_MAX`, or it shall expand to such a constant.

Semantics

The outcome is `1` if *n* has at most two divisors; otherwise the outcome is `0`.

NOTE By this definition, 1 is considered as prime, so `IS_PRIME_(1)` is `1`.

4.13.2.1 Coprimality

Syntax

`ARE_COPRIME_ (` *m* `,` *n* `)`

Constraints

m and *n* shall be non-negative decimal integer constants not exceeding `PP_MAX`, or expand to such a constant.

Semantics

Outcome is `1` if the greatest common divisor of *m* and *n* is 1; otherwise the outcome is `0`.

NOTE `ARE_COPRIME_(`*m*`,` *n*`)` is equivalent to `EQ_(GCD_(`*m*`,` *n*`), 1)`.

4.13.3 Sorting

Syntax

`sort_ (` *key* `,` *argument-list* `)`

Constraints

key shall be a function-like macro that can accept a single argument; whenever *key* is invoked with an element of *argument-list*, its outcome shall be a non-negative decimal integer constant not exceeding `PP_MAX`.

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

Semantics

Each element of the expanded *argument-list* is mapped to the integer generated on invoking *key* with that element, and the expanded list is sorted as per non-decreasing order of the mapping of its elements.

EXAMPLE I A plain list of integers can be sorted using `echo_` as *key*; for instance,
`sort_(echo_, RANGE_(0, 9, 2), RANGE_(9, 0, 2))` expands to `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`.

EXAMPLE II A list of triplets can be sorted on the basis of middle elements using the following macro as *key*:

```
#define mid_(_args_) echo_(top_ C_(pop_ _args_))
```

For instance, `sort_(mid_, (C, 3, c), (A, 1, a), (B, 2, b))` gives `(A, 1, a), (B, 2, b), (C, 3, c)`.

EXAMPLE III `sort_(INC_, 125, 126, 127)` expands to `127, 125, 126`, as its mapping list is `0, 126, 127`.

4.13.4 Permutation

Syntax

`permute__ ( (` *argument-list* `) ,` *index-list* `)`

Constraints

The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.

If *argument-list* expands to N elements and *index-list* expands to I elements, then for each non-negative integer i less than N, the element at index $i\%I$ in *index-list* shall be a non-negative decimal integer not exceeding `PP_MAX`.

Semantics

Let the expanded *argument-list* be labeled as l_0 , $\cdots$, l_{N-1}; similarly the expanded *index-list* can be labeled as i_0 , $\cdots$, i_{I-1}. If elements in the *index-list* at indices less than N have values smaller than N, then the output is:

$$l_{i_0} \ , \ l_{i_{1\%I}} \ , \ \cdots \ , \ l_{i_{(N-1)\%I}}$$

Equivalently, the output is a list of N elements, where the element at index j is of the form $l_{i_{j\%I}}$ $(0 \le j < N)$; in other words, the output element at index j is an input element from the expanded *argument-list* at index $i_{j\%I}$.

If $i_{j\%I}$ is not less than N, it indicates an invalid index, and the existing element l_j is retained at index j.

NOTE The name `permute__` is a misnomer, as some elements can be repeated, and some may not occur at all; a true permutation is obtained if *index-list* is a shuffling of `RANGE_(`N`)`, where every index occurs exactly once.

4.13.5 Transposition

Syntax

`transpose__ (` *parenthesized-lists* `)`

Constraints

Each argument shall be a parenthesized list, and the number of such lists shall be less than `PP_MAX`.

The number of elements in each parenthesized list shall be less than `PP_MAX` after macro expansions.

Semantics

Each parenthesized list is considered to be a row of a matrix, and the outcome is the transpose of such a matrix. If any list contains fewer elements, then its elements are reused from the beginning for generating the transpose.

NOTE Transpose of a transpose restores the original lists only if all of them have same number of elements.

EXAMPLE `transpose__((0, a), (1, b), (2, c))` expands to `(0, 1, 2), (a, b, c)`.

4.13.6 Projection

Syntax

`project_ (` *alpha* `,` *omega* `,` *parenthesized-lists* `)`

Constraints

Both *alpha* and *omega* shall be non-negative decimal integer constants not exceeding `PP_MAX`, or each of them shall expand to such a constant. Each argument shall be a parenthesized list, and the number of such lists shall be less than `PP_MAX`. Number of elements in each parenthesized list shall be less than `PP_MAX` after macro expansions.

Semantics

For each parenthesized list, the elements from index *alpha* through *omega* are selected, as if by applying `slice_` on each expanded list after removing the outermost parentheses. The outcome for each list is fully parenthesized.

NOTE *alpha* is compared with the adjusted value of *omega* in the same way as done by `slice_`.

EXAMPLE `project_(1, 2, (A, B, C, D), (a, b, c, d))` expands to `(B, C), (b, c)`.

4.13.6.1 Inversion

Syntax
`eject_ (` *alpha* `,` *omega* `,` *parenthesized-lists* `)`
Constraints
`eject_` shall have precisely the same constraints as those applicable for `project_`.
Semantics
`eject_` is complementary to `project_` and performs inverse projection on a sequence of parenthesized lists: for each list, `eject_` drops only those elements that are selected by `project_`, and retains rest of the elements.

NOTE The behavior of `eject_` is equivalent to applying `xslice_` on each list after removing the parentheses.

EXAMPLE `eject_(1, 2, (A, B, C, D), (a, b, c, d))` expands to `(A, D), (a, d)`.

4.13.7 Stringizing

Syntax
`quote_ (` *argument-list* `)`
Semantics
Outcome is a string literal for the unexpanded text of *argument-list*, as obtained with preprocessing operator `#`.

EXAMPLE `quote_(top_(ping,), pop_(, corn))` expands to `"top_(ping,), pop_(, corn)"`.

4.13.7.1 Generalization

Syntax
`stringize_ (` *argument-list* `)`
Constraints
The number of elements in the expanded *argument-list* shall be less than `PP_MAX`.
Semantics
Outcome is a list of string literals where each element of the expanded *argument-list* is individually stringized.

NOTE `stringize_(...)` is equivalent to `map__(quote_, __VA_ARGS__)`.

EXAMPLE `stringize_(top_(thundering,), pop_(, typhoons))` expands to `"thundering", "typhoons"`.

Chapter 5

Arrays

An array is a contiguous sequence of objects having the same type. An array having one element is allocated in the same way as a non-array lvalue of the same type; if there are more elements, they are allocated successively after the first element without any padding between two elements: thus the byte offset between two elements is equal to size of the element type. It is permissible to obtain a pointer to one past the last element: it points to the address that is one byte after the end of the last element, which is outside the array and should not be accessed; however, the pointer can be used for comparison with another pointer to an element of the same array.

Elements of an array are indexed from zero (base element). An element can be accessed by specifying its index with the subscript operator: both *array* `[` *index* `]` and *index* `[` *array* `]` are equivalent to `* (` *array* `+` *index* `)`, all of which access the element at the given *index*; the outcome is an lvalue, so the address-of operator can be applied to it (assignment operators can be used only if the type is modifiable). Note that when address arithmetic is performed directly on an array name, it is implicitly "decayed" to a pointer for the base element.

An array definition must specify a complete type, as the length is required to determine allocation; however, an array declaration can omit the length, making it an incomplete array that cannot be used as the operand of `sizeof`. Recall that in C, function parameters of array types are adjusted as pointer to the corresponding element type, so applying `sizeof` to a parameter declared with array type is permitted, though it gives only the size of the adjusted pointer type (which may not be the intent of the programmer). Also, declarations of external arrays can be incomplete. However, the element type of an array cannot be incomplete, so if the element type is also an array type (often called multi-dimensional arrays), then it must have a complete type that contains the length information. In other words, only the "outermost" length of a multi-dimensional array can be omitted in declarations, but the inner dimension(s) must be present, in order to determine the byte offset between two consecutive elements.

NOTE Zero length arrays are not allowed by the ISO C standard, and they can cause undefined behavior.

5.1 Variably modified types

Historically, C required array lengths to be compile-time constants. C99 standardized the practice of variable length arrays (VLA) that allow array definitions to specify a length that can be known at runtime; as a consequence, applying the `sizeof` operator on a VLA works during process execution, and the operand is evaluated as well (side effects can be of concern). VLAs with automatic storage duration are typically allocated on the stack, which can quickly exhaust the stack space that is often very limited on most execution environments; the chances are higher if a function that creates VLAs is called recursively in contexts where tail call optimization is not possible.

C11 revised VLAs to be an optionally supported feature; however, C23 mandates support for variably modified types. The most common example of variably modified types is pointer to VLA, which is heavily used in C_.

NOTE Automatic creation of VLA continues to remain an optionally supported feature, and for portability concerns, they are not used by the reference implementation or the examples in this documentation.

5.2 Pointer to an array

Pointer to an array refers to the same address as pointer to its base element; however, pointer to array has a fundamentally different type from pointer to base element, and this distinction is important for address arithmetic with pointers: adding one with “pointer to an array” makes it point to the next “array”, not to the next element. One advantage of using pointer to array type is that it not implicitly “decay” to a base element pointer.

Pointer to an array is also a natural extension of pointer to other object types, and using pointer to array as a function parameter clearly conveys the intent that some function expects (pointer to) an array, not pointer to a single element. For instance, the standard library function `fputs` can be declared as `int fputs(Char *, File_ *)` where the first parameter expects an array of characters, and the second parameter expects pointer to a single `File_` object. The distinction of array and pointer is not evident from the declaration itself, which can be achieved with pointer to arrays (doing so with existing functions would change their type).

As an example, `strcpyn(Size length, Ptr (Char_ [length]) target, Ptr (Char_ []) source)` declares both `target` (destination) and `source` as pointers to arrays: `target` expects pointer to a complete array having a capacity of `length` elements, whereas source is a pointer to an incomplete array whose length is not specified in the type (it is determined by looking for the first null byte). Most features of C_ that work with arrays expect a pointer to a complete array, so that length information can be extracted from the pointer itself.

5.3 Length

The number of elements in an array is termed as its length. C_ offers the facility to determine array length using `length__` and `length_`*: both expect pointer to a complete array, and length is inferred from the pointer type.

NOTE An array type is not directly allowed in a cast, but pointer to an array can be used instead; this can be used to provide incorrect length information that is different from the actual number of elements in an array. C does not offer any mechanism to invalidate such constructs, and the cast itself is well-defined; trouble brews only when executing some code that attempts to access the array outside its actual bounds. Casting as (pointer to) an array of smaller size is always safe, since an array of n elements ($n > 0$) can act as an array of $n-1$ elements as well. For example, pointer to a variable declared as `Int num;` can be cast as pointer to an array: `(Ptr (Int [1]))&num`; this works because a single `Int` can be interpreted as an array of `Int` having a single element.

5.3.1 `length__`

Syntax

`length__` (*pointer-to-array*)

Constraints

pointer-to-array shall be pointer to a complete array.

Semantics

`length__` infers array length from the type of *pointer-to-array*; the outcome is an expression of type `Size_`. The argument need not be evaluated at all if it does not have a variably modified type; it may be evaluated more than once only if element type of the array is itself variably modified. The behavior is undefined if the pointer is null.

5.3.2 length_*

Syntax
`length_` (*pointer-to-array*)
Constraints
`length_` shall have precisely the same constraints as those applicable for `length__`.
Semantics
`length_` evaluates *pointer-to-array* exactly once; rest of the semantics are identical to `length__`.

5.4 is_array_

Syntax
`is_array_` (*expression-or-type*)
Semantics
If *expression-or-type* does not have a variably modified type, then the outcome is `TRUE` if the argument is an array, and `FALSE` if it is not an array; otherwise the behavior is implementation-defined. The outcome is a constant expression whose type depends on the argument, so it can be used as the controlling expression of a generic selection.

NOTE The reference implementation does not support variably modified types for *expression-or-type*.

EXAMPLE `is_array_("string literal")` is `TRUE`, whereas `is_array_(&"pointer to array")` is `FALSE`.

5.5 Indexing

The `at__` and `at_`* families provide a bounds-checking mechanism for array indexing. These features can be configured with the `DEBUG` macro before including the header `<array._>`: the `defined` state of `DEBUG` is recorded by the macro `ARRAY__` every time `<array._>` is included. If `ARRAY__` expands to `1`, then `DEBUG` was defined, so `at__` and `at_`* families are configured in debugging mode, where they assert that the index is not out of bounds; otherwise `ARRAY__` shall expand to `0`, and if the index is out of bounds, then a default value can be used (if provided).

Additionally, these features also support negative indexing from the end (influenced by the Python language). Negative indexing is done right to left, with the last element being indexed as `-1`, and the base element having an index equal to negative of the array length. Indices always increase when moving from base element to the end.

5.5.1 at__

Syntax
`# include <array._>`
`at__` (*pointer-to-array* , *index* [, *default-value*])
`at__2__` (*pointer-to-array* , *index*)
`at__3__` (*pointer-to-array* , *index* , *default-value*)
Constraints
pointer-to-array shall be pointer to a complete array. *index* shall have an integer type. It shall be possible to use *default-value* to initialize a variable declared with element type of the array on dereferencing *pointer-to-array*.
Semantics
`at__` invokes `at__`*n*`__` if the expanded argument sequence contains n arguments. Both `at__2__` and `at__3__` infer the array bound as if by using `length__`. *index* is converted to the type `Ptrdiff`: if *index* is negative, then `at__3__` adjusts it by adding the (inferred) length; no such adjustment is done by `at__2__`. *default-value* is converted to the type of `**(`*pointer-to-array*`)`, as if by initializing a variable declared with the element type.

The (adjusted) index is compared with the length: if it is non-negative and less than the length, then the array is accessed at that index, and the corresponding element is the outcome; otherwise the index is out of bounds.

When compiled with `ARRAY__` expanding to `1`, one of the following cases can happen for index out of bounds:

- If *index* is negative after conversion to `Ptrdiff`, `at__2__` prints a diagnostic message of the following form:

  `Assertion failed: (`*index*`) >= 0, function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

- If *index* is negative after conversion to `Ptrdiff`, and it remains negative after adding length of the array, `at__3__` prints a diagnostic message of the following form:

  `Assertion failed: (`*index*`) >= -length_(`*pointer-to-array*`),`
  `function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

- If *index* is greater than array length, `at__2__` and `at__3__` print a diagnostic message of the following form:

  `Assertion failed: (`*index*`) < length_(`*pointer-to-array*`),`
  `function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

After writing the diagnostic message to the standard error stream `stderr`, the process terminates as if by calling `exit(1)`. *function-identifier*, *file-name*, and *line-number* are obtained from `__func__`, `__FILE__`, and `__LINE__`.

When compiled with `ARRAY__` expanding to `0`, an invocation of `at__2__` is same as `(*(`*pointer-to-array*`))[`*index*`]`, and the behavior is undefined if *index* is out of bounds. If the adjusted index is out of bounds, then *default-value* is used as the outcome after conversion to element type of the array.

The outcome is an lvalue having the same type as array element. Dereference can be suppressed with address-of operator `&` on the outcome; however, this does not disable bounds-checking. If *default-value* is used as the outcome of `at__3__`, then the lvalue has automatic storage duration, and its lifetime is limited to the nearest enclosing block.

pointer-to-array can be evaluated more than once only if it has a variably modified type; both *index* and *default-value* are evaluated exactly once, even if the latter value is unused. The behavior is implementation-defined if *index* causes a signed overflow during conversion to the type `Ptrdiff`. The behavior is undefined if the inferred length is greater than the actual number of elements in the array, and the (adjusted) index is outside the true array bounds.

5.5.2 `at_*`

Syntax

`# include <array._>`
`at_ (` *pointer-to-array* `,` *index* [`,` *default-value*] `)`
`at_2_ (` *pointer-to-array* `,` *index* `)`
`at_3_ (` *pointer-to-array* `,` *index* `,` *default-value* `)`

Constraints

The `at_` family shall have precisely the same constraints as those applicable for the `at__` family.

Semantics

`at_` family evaluates each expression exactly once; rest of the semantics are identical to `at__` family.

5.6 Synonyms

C_ defines synonyms for some array types that are used frequently: `Encoding` is a synonym for `UByte []`, `String` is a synonym for `Char []`, and `Tape` is a synonym for `Ptr (Void) []`; the modifiable twins end with an underscore.

5.7 Range

A `Range` type specifies a pointer type to a non-modifiable integer array of three elements: *alpha*, *omega*, and *delta*. This triplet encodes an arithmetic sequence that begins at *alpha* and does not go beyond *omega*, with *delta* being the difference between two consecutive members. The last member of the series is of the form $alpha + k * delta$, where k is the maximum non-negative integer for which the last member does not go beyond *omega*.

NOTE If $omega - alpha$ is a multiple of k, then the sequence ends with *omega* as its last member.

Syntax

`Range` (*type*)
`Range_` (*type*)

Constraints

type shall specify an integer type.

Semantics

Both `Range` and `Range_` specify pointer types to a non-modifiable array of three elements. *type* is subjected to integer promotion rules, and non-modifiable form of the promoted type becomes the array element type.

NOTE `Range_` means that the pointer itself is modifable, but the array it points to remains non-modifiable.

5.7.1 range_

The `range_` family is used to instantiate a `Range`: it creates a triplet array and gives a pointer to that array.

Syntax

`range_` (*stop*)
`range_` (*alpha* , *omega* [, *delta*=`1`])
`range_1_` (*stop*)
`range_2_` (*alpha* , *omega*)
`range_3_` (*alpha* , *omega* , *delta*)

Constraints

alpha, *omega*, *delta*, and *stop* shall be expressions having integer types.

Semantics

`range_` invokes `range_`*n*`_` if the expanded argument sequence contains n arguments. `range_3_` creates a non-modifiable array of integers initialized with *alpha*, *omega*, and *delta* (in order), and gives a pointer to that array. Element type of the array is same as type of the expression `(`*alpha*`) - (`*omega*`)`. If `range_` family is used within a function, the array has automatic storage duration, and its lifetime is limited to the nearest enclosing block.

`range_1_(`*stop*`)` is equivalent to `range_2_(0, (`*stop*`) - 1)`.

`range_2_(`*alpha*`, `*omega*`)` is equivalent to `range_3_(`*alpha*`, `*omega*`, 1)`.

NOTE The value of *delta* is converted to type of the expression `(`*alpha*`) - (`*omega*`)`.

5.7.2 alpha_, omega_, delta_

`alpha_`, `omega_`, and `delta_` are configurable features that can do null pointer diagnosis when compiled in debugging mode. The object-like macro `RANGE__` records the `defined` state of `DEBUG` macro every time the header `<range._>` is included: if `DEBUG` is defined before including `<range._>`, then `RANGE__` expands to `1`, and `0` otherwise.

Syntax

```
# include <range._>
alpha_ ( range )
omega_ ( range )
delta_ ( range )
```

Constraints

range shall be pointer to a non-modifiable integer array of length three, such that the element type does not change when subjected to integer promotion rules.

Semantics

When compiled with `RANGE__` expanding to `0`, `alpha_(`*r*`)`, `omega_(`*r*`)`, and `delta_(`*r*`)` have the same values as `(*(`*r*`))[0]`, `(*(`*r*`))[1]`, and `(*(`*r*`))[2]` (respectively). Otherwise `RANGE__` shall expand to `1`, and these features additionally check if *range* is a null pointer, as if by using `notnull_`. In all cases, the outcome is not an lvalue.

5.8 Bit arrays

C_ offers several facilities for bitwise operations on resizeable arrays. Memory is typically byte-addressable, and individual bits are accessed with bitwise operators. C_ provides simple abstractions for performing bit manipulations over arrays that are interpreted as a packed sequence of bits (padding bits can be present only at the end).

The reference implementation provides these features in the header `<bits._>`, most of which can be configured.

5.8.1 Bits

`Bits` is a non-modifiable array of `UInt_fast8`; `Bits_` is the modifiable counterpart. `UInt_fast8` is an implementation-defined synonym for an unsigned integer type (possibly an extended type), provided by the header `<stdint._>`.

5.8.2 BITS_WIDTH

`BITS_WIDTH` is an enumeration constant whose value is equal to `UINT_FAST8_WIDTH` defined in `<stdint._>`.

5.8.3 Word count

`Bits` is an array of `UInt_fast8`, and each array element is called a “word”.

`wordcount_` calculates the minimum number of words required to store a given number of bits.

Syntax

`wordcount_ (` *bitcount* `)`

Constraints

bitcount shall be an expression having integer type.

Semantics

bitcount is first converted to type `Size`, and `wordcount_` gives the number of words necessary to store the number of bits given by the converted value of *bitcount*. The outcome is of type `Size_`.

5.8.4 Bit count

Syntax

`bitcount_ (` *pointer-to-array* `)`

Constraints

pointer-to-array shall be pointer to a complete array.

Semantics

`bitcount_` multiplies `CHAR_BIT` with size of the array inferred from its type; the outcome is of type `Size_`.

pointer-to-array need not be evaluated if it does not have a variably modified type.

5.8.5 Creating a bit array

The `bits_` family can be used to create a dynamically resizeable bit array.

Syntax

`bits_` (*bitcount* [, *initializer*])
`bits_1_` (*bitcount*)
`bits_2_` (*bitcount* , *initializer*)

Constraints

bitcount shall be an expression having integer type, and *initializer* shall be a scalar expression.

Semantics

`bits_` invokes `bits_`*n*`_` if the expanded argument sequence contains *n* arguments. On success, the outcome is a pointer to `Bits_`, whose length (number of words) is given by `wordcount_(`*bitcount*`)`; the outcome is a null pointer if the required allocation cannot be obtained. If *initializer* compares equal to zero, then all bits are zeroed out (reset); otherwise *initializer* is non-zero, and all bits are set to one. The array can be deallocated by passing the pointer to `free`, and it can be resized with `realloc`, or with the help of `renew__`/`renew_`* families.

NOTE If the converted value of *bitcount* is not a multiple of `BITS_WIDTH`, there will be extra bits at the end.

5.8.6 Basic operations

The basic operations on bit arrays include getting the bit value at a given bit index, clearing a bit, and setting it to `1`; these facilities are respectively provided by `bit_`, `rst_`, and `set_`, all of which can be configured with the `DEBUG` macro. The macro `BITS__` records the `defined` state of `DEBUG` when the header `<bits._>` is included: `BITS__` expands to `1` if `DEBUG` was defined, configuring the features in debugging mode; otherwise `BITS__` expands to `0`.

Syntax

`# include <bits._>`
`bit_` (*pointer-to-bits* , *bit-index*)
`rst_` (*pointer-to-bits* , *bit-index*)
`set_` (*pointer-to-bits* , *bit-index*)

Constraints

pointer-to-bits shall be pointer to a complete array whose unqualified element type is `UInt_fast8_`; for `rst_` and `set_`, the array shall be modifiable. *bit-index* shall be an expression having integer type.

Semantics

Bit count is inferred from the type of *pointer-to-bits*. *bit-index* is converted to type `LLong`: if it is negative, bit count is added to it to get the adjusted index. If the adjusted index is non-negative and less than bit count, then:

- `bit_` gives the current bit value at the adjusted *bit-index*
- `rst_` sets the bit to `0` at the adjusted *bit-index*, and gives the old bit value
- `set_` sets the bit to `1` at the adjusted *bit-index*, and gives the old bit value

In all cases, the outcome is of type `Bool_`. When compiled with `BITS__` expanding to `1`, *pointer-to-bits* is asserted to be not null as if by using `notnull_`. If *bit-index* is out of bounds, one of the following cases can happen:

- If *bit-index* remains negative after adding the inferred bit count, the following diagnostic message is printed:

  `Assertion failed: (`*bit-index*`) >= -bitcount_(`*pointer-to-bits*`),`
  `function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

- If *bit-index* is not less than the inferred bit count, the following diagnostic message is printed:

  `Assertion failed: (`*bit-index*`) < bitcount_(`*pointer-to-bits*`),`
  `function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

After writing the diagnostic message to the standard error stream `stderr`, the process terminates as if by calling `exit(1)`. *function-identifier*, *file-name*, and *line-number* are obtained from `__func__`, `__FILE__`, and `__LINE__`.

When compiled with `BITS__` expanding to `0`, the outcome of `bit_`, `rst_`, and `set_` is zero if *pointer-to-bits* is null, or the adjusted value of *bit-index* is out of bounds. The behavior is undefined if the inferred bit count is greater than the actual number of bits in the array, and the adjusted *bit-index* is outside the true array bounds.

5.8.7 Aggregate operations

C_ supports the facilities of `<stdbit.h>` for bit arrays. The standard header `<stdbit.h>` added in C23 defines utility functions and macros for aggregate operations on binary representation of unsigned integers. For each utility, C_ provides a *similar* function for bit arrays, which accepts a length (word count), and a pointer to `Bits`; the functions whose names start with `first` have slightly different behavior from their respective analogues in `<stdbit.h>`.

Each function has two wrappers whose names end with underscore: these infer word count from the type of a complete bit array, as done by `length_`; type of the outcome is same as return type of the corresponding function. Each wrapper named with two trailing underscores can evaluate its pointer argument more than once only if it has a variably modified type; its companion named with a single trailing underscore evaluates the pointer exactly once.

The following subsections describe these features, all of which can be configured with the `DEBUG` macro. If `DEBUG` remains defined before including `<bits._>`, the macro `BITS__` expands to `1`, and the utilities named with trailing underscore(s) assert that the pointer is not null, as if by using `notnull_`; otherwise `BITS__` shall expand to `0`.

The reference implementation provides inline definitions for all the functions. The following wrappers ending with a single underscore are starred, as the reference implementation provides them using non-standard extensions.

`leading_zeros_`*	`trailing_zeros_`*	`first_leading_zero_`*	`first_trailing_zero_`*	`count_zeros_`*
`leading_ones_`*	`trailing_ones_`*	`first_leading_one_`*	`first_trailing_one_`*	`count_ones_`*
		`has_single_bit_`*		

NOTE If the bit count is not a multiple of `BITS_WIDTH` when creating a bit array, then there will be some padding bits at the end; the original bit count cannot be determined from the array length (word count), so all bits are considered as value bits, and extra bits at the end can affect the outcome of some aggregation operations.

5.8.7.1 Count leading zeros

Syntax

`# include <bits._>`
`ULLong_ leading_zeros ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`leading_zeros__ (` *pointer-to-bits* `)`
`leading_zeros_ (` *pointer-to-bits* `)`

Semantics

If pointer to the bit array is not null, then the outcome is the number of consecutive bits with value zero starting from the lowest bit index. The function `leading_zeros` returns zero if the argument for *bitarray* is a null pointer.

5.8.7.2 Count leading ones

Syntax

`# include <bits._>`
`ULLong_ leading_ones ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`leading_ones__ (` *pointer-to-bits* `)`
`leading_ones_ (` *pointer-to-bits* `)`

Semantics

If pointer to the bit array is not null, then the outcome is the number of consecutive bits with value one starting from the lowest bit index. The function `leading_ones` returns zero if the argument for *bitarray* is a null pointer.

5.8.7.3 Count trailing zeros

Syntax

```
# include <bits._>
ULLong_ trailing_zeros ( Size wordcount , Ptr ( UInt_fast8 [ wordcount ] ) bitarray ) ;
trailing_zeros__ ( pointer-to-bits )
trailing_zeros_  ( pointer-to-bits )
```

Semantics

If pointer to the bit array is not null, then the outcome is the number of consecutive bits with value zero starting from the highest bit index. The function `leading_zeros` returns zero if the argument for *bitarray* is a null pointer.

5.8.7.4 Count trailing ones

Syntax

```
# include <bits._>
ULLong_ trailing_ones  ( Size wordcount , Ptr ( UInt_fast8 [ wordcount ] ) bitarray ) ;
trailing_ones__  ( pointer-to-bits )
trailing_ones_   ( pointer-to-bits )
```

Semantics

If pointer to the bit array is not null, then the outcome is the number of consecutive bits with value one starting from the highest bit index. The function `leading_ones` returns zero if the argument for *bitarray* is a null pointer.

5.8.7.5 First leading zero

Syntax

```
# include <bits._>
ULLong_ first_leading_zero ( Size wordcount , Ptr ( UInt_fast8 [ wordcount ] ) bitarray ) ;
first_leading_zero__ ( pointer-to-bits )
first_leading_zero_  ( pointer-to-bits )
```

Semantics

If pointer to the bit array is not null, then outcome is the lowest bit index set to zero; if all bits are set to one, the outcome is bit count. The function `first_leading_zero` returns the bit count if *bitarray* is a null pointer.

5.8.7.6 First leading one

Syntax

```
# include <bits._>
ULLong_ first_leading_one  ( Size wordcount , Ptr ( UInt_fast8 [ wordcount ] ) bitarray ) ;
first_leading_one__  ( pointer-to-bits )
first_leading_one_   ( pointer-to-bits )
```

Semantics

If pointer to the bit array is not null, then outcome is the lowest bit index set to one; if all bits are set to zero, the outcome is bit count. The function `first_leading_one` returns the bit count if *bitarray* is a null pointer.

5.8.7.7 First trailing zero

Syntax

`# include <bits._>`
`ULLong_ first_trailing_zero ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`first_trailing_zero__ (` *pointer-to-bits* `)`
`first_trailing_zero_ (` *pointer-to-bits* `)`

Semantics

If pointer to the bit array is not null, then outcome is the highest bit index set to zero; if all bits are set to one, the outcome is bit count. The function `first_trailing_zero` returns the bit count if *bitarray* is a null pointer.

5.8.7.8 First trailing one

Syntax

`# include <bits._>`
`ULLong_ first_trailing_one ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`first_trailing_one__ (` *pointer-to-bits* `)`
`first_trailing_one_ (` *pointer-to-bits* `)`

Semantics

If pointer to the bit array is not null, then outcome is the highest bit index set to one; if all bits are set to zero, the outcome is bit count. The function `first_trailing_one` returns the bit count if *bitarray* is a null pointer.

5.8.7.9 Count zeros

Syntax

`# include <bits._>`
`ULLong_ count_zeros ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`count_zeros__ (` *pointer-to-bits* `)`
`count_zeros_ (` *pointer-to-bits* `)`

Semantics

If pointer to bit array is not null, outcome is the number of bits set to zero; otherwise the function returns zero.

5.8.7.10 Count ones

Syntax

`# include <bits._>`
`ULLong_ count_ones ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`count_ones__ (` *pointer-to-bits* `)`
`count_ones_ (` *pointer-to-bits* `)`

Semantics

If pointer to bit array is not null, outcome is the number of bits set to one; otherwise the function returns zero.

5.8.7.11 Single-bit check

Syntax

`# include <bits._>`
`Bool_ has_single_bit ( Size` *wordcount* `, Ptr ( UInt_fast8 [` *wordcount* `] )` *bitarray* `) ;`
`has_single_bit__ (` *pointer-to-bits* `)`
`has_single_bit_ (` *pointer-to-bits* `)`

Semantics

For a valid pointer to a bit array, the outcome is one if and only if a single bit is set to one, and rest of the bits are zeroed out. The function `has_single_bit` returns zero if the argument for *bitarray* is a null pointer.

5.8.8 Rotation

Syntax

`# include <bits._>`
`Void_ rotate_bits ( Size` *wordcount* `, Ptr ( UInt_fast8_ [`*wordcount*`] )` *bitarray* `, LLong` *rotation* `) ;`
`rotate_bits__ (` *pointer-to-bits* `,` *rotation* `)`
`rotate_bits_ (` *pointer-to-bits* `,` *rotation* `)`

Constraints

pointer-to-bits shall be pointer to a complete `Bits_` array; the array shall be modifiable.

For `rotate_bits__` and `rotate_bits_`*, *rotation* shall be an expression having integer type.

Semantics

`rotate_bits__` and `rotate_bits_`* infer *wordcount* from the type of *pointer-to-bits* and invoke `rotate_to_bits`. If *bitarray* is null, the function returns immediately; otherwise the bits are index left to right starting at index zero. A positive *rotation* rotates bits to the right, i.e. towards higher index, and a negative *rotation* rotates bit to the left, i.e. towards lower index; the array is considered logically circular, so the last bit is followed by the first bit.

More precisely, if *rotation* is r and bit count is n, then for each index i, the bit is moved to index $(i + r)$ `%` n;

When compiled with `BITS__` expanding to `1`, *pointer-to-bits* is asserted to be not null, as if with `notnull_`.

5.8.9 Shifting

5.8.9.1 Left shift

Syntax

`# include <bits._>`
`Void_ left_shift (Size` *wordcount*`, Ptr (UInt_fast8_ [`*wordcount*`])` *bitarray*`, ULLong` *shift*`, Bool` *bit*`);`
`left_shift__ (` *pointer-to-bits* `,` *shift* `,` *bit* `)`
`left_shift_ (` *pointer-to-bits* `,` *shift* `,` *bit* `)`

Constraints

pointer-to-bits shall be pointer to a complete `Bits_` array; the array shall be modifiable.

For `left_shift__` and `left_shift_`*, *shift* shall be an expression having integer type.

Semantics

`left_shift__` and `left_shift_`* infer *wordcount* from the type of *pointer-to-bits* and invoke `left_shift`. If *bitarray* is null or *shift* is greater than bit count, the function returns immediately; otherwise the bits are index left to right starting at index zero, and if *shift* is s, the bits are shifted towards lower indices by s positions. The bits originally at indices 0 through $s - 1$ are lost after the left shift, and the last s vacated bits are filled with *bit*.

When compiled with `BITS__` expanding to `1`, *pointer-to-bits* is asserted to be not null, as if with `notnull_`.

5.8.9.2 Shift right

Syntax

`# include <bits._>`
`Void_ shift_right (Size` *wordcount*`, Ptr (UInt_fast8_ [`*wordcount*`])` *bitarray*`, ULLong` *shift*`, Bool` *bit*`);`
`shift_right__ (` *pointer-to-bits* `,` *shift* `,` *bit* `)`
`shift_right_ (` *pointer-to-bits* `,` *shift* `,` *bit* `)`

Constraints

pointer-to-bits shall be pointer to a complete Bits_ array; the array shall be modifiable.

For shift_right__ and shift_right_, *shift* shall be an expression having integer type.

Semantics

shift_right__ and shift_right_* infer *wordcount* from the type of *pointer-to-bits* and invoke shift_right. If *bitarray* is null or *shift* is greater than bit count, the function returns immediately; otherwise the bits are index left to right starting at index zero, and if *shift* is s, the bits are shifted towards higher indices by s positions. The bits originally at the last s indices are lost after the right shift, and the first s vacated bits are filled with *bit*.

When compiled with BITS__ expanding to 1, *pointer-to-bits* is asserted to be not null, as if with notnull_.

5.9 Iterators

The header <iterators._> provides several features for iterating over arrays, and applying a function or an operator to the elements. These iterators are influenced by higher order functions from functional programming, and their semantics are similar to the macro iterators from the ellipsis framework; the fundamental difference is that the macro iterators operate during preprocessing, but the array iterators operate during execution. With the exception of join_ family, rest of the iterators are statements, so they cannot be used within expressions under ISO C syntax.

<iterators._> only aggregates several other headers that are described in the subsections; each iterator family can be configured for debugging by defining the DEBUG macro before including the associated header. The purpose of <iterators._> is to ensure a uniform configuration for all iterators; however, an iterator family can be individually (re-)configured simply by (re-)including its associated header, possibly preceded by an active definition of DEBUG.

Each header that is included by <iterators._> defines an object-like macro for recording the defined state of DEBUG macro every time that header is included: if that macro expands to 1, then the corresponding iterator family is configured in debugging mode, and it asserts that all pointer arguments are not null, as done by notnull_. Some of the iterator families accept a *range* argument, which specifies an arithmetic progression of indices; negative indices are adjusted by adding length of the array that is inferred from the array type. When compiled in debugging mode, an iterator asserts that adjusted values of *alpha* and *omega* are within the index range, in addition to checking that the *range* pointer is not null. If a *range* assertion fails, then one of the following diagnostic messages is printed:

Assertion failed: (*range*) != NULL, function *function-identifier*, file *file-name*, line *line-number*.

Assertion failed: alpha_(*range*) >= -length_(*source*),
function *function-identifier*, file *file-name*, line *line-number*.

Assertion failed: alpha_(*range*) < length_(*source*),
function *function-identifier*, file *file-name*, line *line-number*.

Assertion failed: omega_(*range*) >= -length_(*source*),
function *function-identifier*, file *file-name*, line *line-number*.

Assertion failed: omega_(*range*) < length_(*source*),
function *function-identifier*, file *file-name*, line *line-number*.

function-identifier, *file-name*, and *line-number* are respectively obtained from __func__, __FILE__, and __LINE__. Diagnostic message is written to standard error stream stderr, and the process terminates as if by calling exit(1).

When compiled in non-debugging mode, if *alpha* or *omega* is less than negative of the array length, it is adjusted to zero; if it is not less than the array length, it is adjusted to one less than the array length. *delta* is then applied on the adjusted bounds. It is possible that the resulting index series is empty, in which case no iteration is performed.

NOTE If an iterator also accepts a *destination* array, length of the smaller array is used for *range* adjustments.

5.9.1 map_

The header <map._> defines the macro MAP__ that configures the behavior of map_ family; MAP__ records the defined state of DEBUG macro every time <map._> is included: it expands to 1 if DEBUG was defined, and 0 otherwise.

Syntax

```
# include <map._>
map_   ( [destination ,] function , source [, range] )
map_2_ ( function , source )
map_3_ ( destination , function , source )
map_4_ ( destination , function , source , range )
```

Constraints

Both *source* and *destination* shall be pointers to complete arrays.

For map_2_, *source* array shall be modifiable; for map_3_ and map_4_, *destination* array shall be modifiable.

function shall be a function type expression that can be called with an element of *source* array with requiring any type cast; return type of the function shall be suitable to store the return value in *destination* without any type casting (for map_2_, *source* is itself the destination). *range* shall be an expression having a Range type.

Semantics

map_ invokes map_n_ if the expanded argument sequence contains n arguments.

For each element in *source*, map_2_ invokes *function* with that element as the argument, and the return value of that invocation replaces the element in array; the sequence of invocation and replacement is implementation-defined.

map_3_ is similar to map_2_, except that the return values are stored in *destination*, and *source* can be non-modifiable. If *destination* has length n and n is not greater than length of *source*, only the first n elements of *source* are mapped to *destination*; otherwise *destination* is longer than *source*, and extra elements are not modified.

map_4_ invokes *function* only for the index series specified by *range*, and in the given order. For each selected element of *source*, the return value is stored at the same index in *destination*; rest of the elements are not modified.

NOTE map_2_(f, *source*) is equivalent to map_3_(*source*, f, *source*), except that *source* is evaluated once.

5.9.2 fold_

The header <fold._> defines a macro FOLD__ that configures the behavior of fold_ family; FOLD__ records the defined state of DEBUG macro whenever <fold._> is included: it expands to 1 if DEBUG was defined, and 0 otherwise.

Syntax

```
# include <fold._>
fold_   ( accumulator , function , source [, range] )
fold_3_ ( accumulator , function , source )
fold_4_ ( accumulator , function , source , range )
```

Constraints

source shall be pointer to a complete array type. *range* shall be an expression having a Range type.

function shall be a function type expression that can be called with *accumulator* as the first argument and an element of *source* as the second argument, without requiring any type cast for each argument.

accumulator shall be a modifiable lvalue that can be assigned with the return value of *function* without any type cast; additionally, it shall be possible to obtain a pointer to *lvalue* with the address-of operator &.

Semantics

fold_ invokes fold_n_ if the expanded argument sequence contains n arguments. If the length of *source* array is inferred to be n, fold_2_ invokes *function* n times, with *accumulator* as the first argument and an element of *source* as the second argument: the first invocation is done with the element at index zero in *source*; subsequent invocations use the element next to the one for the previous iteration. For each invocation, the return value of *function* is stored in *accumulator* itself, which is then used in the next iteration (if any).

fold_4_ is similar to fold_3_, except that it invokes *function* only for the index series specified by *range*.

5.9.3 `reduce_`

`<reduce._>` defines the macro `REDUCE__` that configures the behavior of `reduce_` family; `REDUCE__` records the `defined` state of `DEBUG` every time `<reduce._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

`# include <reduce._>`
`reduce_` (*accumulator* , *function* , *source* [, *range*])
`reduce_3_` (*accumulator* , *function* , *source*)
`reduce_4_` (*accumulator* , *function* , *source* , *range*)

Constraints

source shall be pointer to a complete array type. *range* shall be an expression having a `Range` type.

function shall be a function type expression that can be called with *accumulator* as the first argument and an element of *source* as the second argument, without requiring any type cast for each argument.

accumulator shall be a modifiable lvalue that can be assigned with the return value of *function* without any type cast; additionally, it shall be possible to obtain a pointer to *lvalue* with the address-of operator `&`.

Semantics

`reduce_` invokes `reduce_`n`_` if the expanded argument sequence contains n arguments. If the length of *source* array is inferred to be n, `reduce_2_` invokes *function* n times, with *accumulator* as the first argument and an element of *source* as the second argument: the first invocation is done with the element at index zero in *source*; subsequent invocations use the element next to the one for the previous iteration. For each invocation, the return value of *function* is stored in *accumulator* itself, which is then used in the next iteration (if any).

`reduce_4_` is similar to `reduce_3_`, except that it invokes *function* only for the index series specified by *range*.

5.9.4 `omni_`

The header `<omni._>` defines the macro `OMNI__` that configures the behavior of `omni_` family; `OMNI__` records the `defined` state of `DEBUG` macro whenever `<omni._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

`# include <omni._>`
`omni_` ([*destination* ,] *left-operand* , *function* , *right-operand*)
`omni_3_` (*left-operand* , *function* , *right-operand*)
`omni_4_` (*destination* , *left-operand* , *function* , *right-operand*)

Constraints

destination, *left-operand*, and *right-operand* shall have scalar types: if the type is a pointer type, then it shall be pointer to a complete array; otherwise the type shall be an arithmetic type. *left-operand* (for `omni_3_`) and *destination* (for `omni_4_`) shall be pointer to a modifiable array, or a modifiable lvalue whose address can be obtained with the `&` operator, interpreted as pointer to an array having a single element of arithmetic type.

function shall be a function type expression that can be called with an element of *left-operand* as the first argument and an element of *right-operand* as the second argument, without requiring type cast for each argument.

Semantics

`omni_` invokes `omni_`n`_` if the expanded argument sequence contains n arguments. These features are modeled after `omni__` macro from the ellipsis framework (provided by the header `<templates._>`), and their operational semantics are identical (the precise behavior is described in chapter 4). If *destination*, *left-operand*, or *right-operand* has an arithmetic type, it is interpreted as an array having a single element. `omni_3_` stores the return values in *left-operand* itself, whereas `omni_4_` stores them in *destination* (so *left-operand* can be pointer to a non-modifiable array or an arithmetic expression that is not an lvalue). If length of *destination* is n and n is less than length of *right-operand*, then only the first n elements of *right-operand* are used; all elements of *right-operand* are utilized.

NOTE If *left-operand* array is shorter than *right-operand* array, then elements will be reused from *left-operand* in a round-robin sequence (going back to the first element after using the last one); however, `omni_4_` stores the return values in *destination*, so the earlier results will be used instead of the original value(s) from *left-operand*.

5.9.5 `op_`

The header `<op._>` defines the macro `OP__` that configures the behavior of `op_` family; `OP__` records the `defined` state of `DEBUG` macro whenever `<op._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

```
# include <op._>
op_   ( unary-operator , operand )
op_   ( [destination ,] left-operand , binary-operator , right-operand )
op_2_ ( unary-operator , operand )
op_3_ ( left-operand , binary-operator , right-operand )
op_4_ ( destination , left-operand , binary-operator , right-operand )
```

Constraints

operand, *destination*, *left-operand*, and *right-operand* shall have scalar types: if the type is a pointer type, it shall be pointer to a complete array; otherwise the type shall be an arithmetic type. *operand* (for `op_2_`), *left-operand* (for `op_3_`), and *destination* (for `op_4_`) shall be pointer to a modifiable array, or a modifiable lvalue whose address can be obtained with the `&` operator, interpreted as pointer to an array having a single element of arithmetic type.

unary-operator shall be applicable on each element of *operand* without causing any semantic violation. Similarly, it shall be possible to use *binary-operator* with two operands: an element of *left-operand* (for `op_3_`) or *destination* (for `op_4_`) placed before the operator, and an element of *right-operand* placed after the operator.

Semantics

`op_` invokes `op_`n`_` if the expanded argument sequence contains n arguments. These features are modeled after `op__` family from the ellipsis framework (provided by the header `<templates._>`), and their operational semantics are identical (the precise behavior is described in chapter 4). If *operand*, *left-operand*, or *right-operand* has an arithmetic type, it is interpreted as an array having a single element. `op_2_` applies unary operator on each element of *operand*, which is then replaced with the result of the operation. `op_3_` and `op_4_` are analogous to `op_3_` and `op_4_`, except that the elements of *left-operand* and *right-operand* are used with *binary-operator* instead of function call arguments. `op_3_` stores the return values in *left-operand* itself, whereas `op_4_` stores them in *destination*. If length of *destination* is n and n is less than length of *right-operand* (as inferred from the array types), then only the first n elements of *right-operand* are used; in any case, all elements of *right-operand* are utilized at least once.

For `op_3_`, if *binary-operator* is not an assignment operator, then it is considered to be one. For `op_4_`, even if *binary-operator* is an assignment operator, the results are stored in *destination*, and *left-operand* remains unchanged.

5.9.6 `rel_`

The header `<rel._>` defines the macro `REL__` that configures the behavior of `rel_` family; `REL__` records the `defined` state of `DEBUG` macro whenever `<rel._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

```
# include <rel._>
rel_   ( flag , unary-operator , operand )
rel_   ( flag , left-operand , binary-operator , right-operand )
rel_3_ ( flag , unary-operator , operand )
rel_4_ ( flag , left-operand , binary-operator , right-operand )
```

Constraints

$flag$ shall be a modifiable lvalue of arithmetic type. *left-operand* and *right-operand* shall have scalar types: if the type is a pointer type, it shall be pointer to a complete array; otherwise the type shall be an arithmetic type.

For `rel_3_`, it shall be possible to use *binary-operator* with two consecutive elements of *operand*; for `rel_4_`, *binary-operator* shall not cause semantic error when an element of *left-operand* is placed before the operator, and an element of *right-operand* is placed after the operator. An expression with *binary-operator* shall have scalar type.

NOTE *binary-operator* is intended to be a relational or equality operator, but it is not imposed as a constraint.

Semantics

`rel_` invokes `rel_`n`_` if the expanded argument sequence contains n arguments. These features are modeled after `rel__` family from the ellipsis framework (provided by the header `<templates._>`), and their operational semantics are identical (the precise behavior is described in chapter 4). If *operand*, *left-operand*, or *right-operand* has an arithmetic type, then it is interpreted as an array having a single element. Only the lvalue $flag$ is modified.

If *operand* has a single element, then `rel_3_` sets $flag$ to `TRUE_()` (one). If the length of *operand* is inferred to be n and n is greater than one, then every pair of consecutive elements is used as the operands of *binary-operator* as long as the outcome of the operation is non-zero. If an outcome is zero, then rest of the iterations are skipped, and $flag$ is set to `FALSE_()` (zero); otherwise all the $n - 1$ outcomes are non-zero, and $flag$ is set to `TRUE_()`.

If both *left-operand* and *right-operand* have a single element each, then `rel_4_` sets $flag$ to `TRUE_()`; otherwise let n be the maximum length of the two arrays. At most n iterations are performed, and in each iteration, *binary-operator* is used with an element of *left-operand* and an element of *right-operand*: if the outcome is zero, then subsequent iterations are skipped and $flag$ is set to `FALSE_()`; otherwise all the n outcomes are non-zero and $flag$ is set to `TRUE_()`. Whenever a shorter array is exhausted, its elements are reused in a round-robin sequence.

NOTE A non-trivial assignment to $flag$ is the logical conjunction of all the outcomes for *binary-operator*.

5.9.7 `filter_`

`<filter._>` defines the macro `FILTER__` that configures the behavior of `filter_` family; `FILTER__` records the `defined` state of `DEBUG` every time `<filter._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

`# include <filter._>`
`filter_` (*destination* , *predicate* , *source* [, *range*] , *key*)
`filter_4_` (*destination* , *predicate* , *source* , *key*)
`filter_5_` (*destination* , *predicate* , *source* , *range* , *key*)

Constraints

Both *source* and *destination* shall be pointers to complete arrays. *destination* array shall be modifiable, and its element type shall be suitable to store an element of *source* array without any type cast.

predicate shall be a function type expression that can be called with an element of *source* as the first argument, with *key* being the subsequent argument(s) without any type cast; return type of *predicate* shall be a scalar type.

range shall be an expression having a `Range` type.

If *key* needs to specify multiple arguments for calling *predicate*, it shall be a fully parenthesized list for `filter_`.

Semantics

`filter_` invokes `filter_`n`_` if the expanded argument sequence contains n arguments. *key* is subjected to the `peel_` macro before counting the number of elements in it using `COUNT_`: if the count is one, then *key* is used without peeling; otherwise the resulting text after applying `peel_` is used, which can be a list of arguments.

`filter_4_` calls *predicate* for each element of *source*: in each invocation, the element is the first argument, followed by *key* (peeled in case it expands to multiple arguments). If the return value is non-zero, then that element is copied to *destination*; if *destination* gets full before reaching the end of *source*, subsequent iterations are skipped.

`filter_5_` calls *predicate* only for elements in the indices specified by *range*, as long as *destination* is not full.

5.9.8 `search_`

`<search._>` defines the macro `SEARCH__` that configures the behavior of `search_` family; `SEARCH__` records the `defined` state of `DEBUG` every time `<search._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

`# include <search._>`
`search_` (*found-at* , *predicate* , *source* [, *range*] , *key*)
`search_4_` (*found-at* , *predicate* , *source* , *key*)
`search_5_` (*found-at* , *predicate* , *source* , *range* , *key*)

Constraints

found-at shall be a modifiable lvalue of arithmetic type, and *source* shall be pointer to a complete array.

predicate shall be a function type expression that can be called with an element of *source* as the first argument, with *key* being the subsequent argument(s) without any type cast; return type of *predicate* shall be a scalar type.

range shall be an expression having a `Range` type.

If *key* needs to specify multiple arguments for calling *predicate*, it shall be a fully parenthesized list for `search_`.

Semantics

`search_` invokes `search_`n`_` if the expanded argument sequence contains n arguments. *key* is subjected to the `peel_` macro before counting the number of elements in it using `COUNT_`: if the count is one, then *key* is used without peeling; otherwise the resulting text after applying `peel_` is used, which can be a list of arguments.

`search_4_` calls *predicate* for each element of *source*, starting at index zero and moving towards higher indices: in each invocation, the element is the first argument, followed by *key* (peeled in case it expands to multiple arguments). If a return value is non-zero, then that index is copied to *found-at*, and rest of the iterations are skipped; otherwise *predicate* returns zero for all elements of *source*, and *found-at* is set to the length of *source*.

`search_5_` calls *predicate* only for elements in the indices specified by *range*: *found-at* is set to the first index (as per the sequence specified by *range*) for which *predicate* returns non-zero, skipping rest of the index sequence; otherwise *predicate* returns zero for all elements in the given *range*, and *found-at* is set to the length of *source*.

NOTE If *predicate* returns non-zero, *found-at* is set to a non-negative index, even if *range* has negative values.

5.9.9 `permute_`

`<permute._>` defines the macro `PERMUTE__` that configures the behavior of `permute_` family; `PERMUTE__` records the `defined` state of `DEBUG` every time `<permute._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

Syntax

`# include <permute._>`
`permute_` ([*destination* ,] *permutation* , *source* [, *range*])
`permute_2_` (*permutation* , *source*)
`permute_3_` (*destination* , *permutation* , *source*)
`permute_4_` (*destination* , *permutation* , *source* , *range*)

Constraints

source, *permutation*, and *destination* shall be pointers to complete arrays.

For `permute_2_`, *source* array shall be modifiable; for `permute_3_` and `permute_4_`, *destination* array shall be modifiable. It shall be possible to copy an element of *source* array to *destination* array without any type cast.

permutation shall point to an array whose elements have integer type.

range shall be an expression having a `Range` type.

Semantics

`permute_` invokes `permute_`n`_` if the expanded argument sequence contains n arguments.

For `permute_2_`, let n be length of the smaller array. For each index i from 0 through $n-1$, let i' be the element at index i of *permutation*. The element at index i' of the initial *source* array is copied to index i in the same array.

For `permute_3_`, let n be the smallest length out of the three arrays. For each index i from 0 through $n-1$, let i' be the element at index i of *permutation*. The element at index i' of the initial *source* array is copied to index i in the *destination* array. Let n' be length of the smaller array out of *source* and *destination*: if n is smaller than n', then the elements from index n through $n'-1$ are directly copied from the initial *source* array to *destination*.

For `permute_4_`, let n be the smallest length out of the three arrays. For each index i in the sequence specified by *range*, if i is less than n, let i' be the element at index i of *permutation*. The element at index i' of the initial *source* array is copied to index i in the *destination* array; rest of the elements are not modified in *destination*.

When compiled with `PERMUTE__` as `1`, if a required element of *permutation* array is not a valid *source* index, one of the following diagnostic messages is written to `stderr`, and the process terminates as if by calling `exit(1)`.

`Assertion failed: (*(`*permutation*`))[`*index*`] >= -length_(`*source*`),`
`function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

`Assertion failed: (*(`*permutation*`))[`*index*`] < length_(`*source*`),`
`function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

function-identifier, *file-name*, and *line-number* are respectively obtained from `__func__`, `__FILE__`, and `__LINE__`.

When compiled with `PERMUTE__` as `0`, if *permutation* array contains an invalid *source* index, then it is ignored.

NOTE The reference implementation creates a copy of the initial *source* array, in case the arrays overlap; the copy is freed after the permutation. If memory allocation fails, the following diagnostic message is printed:

`Assertion failed: new_(*(`*source*`)) != NULL, function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

This message is written to the standard error stream `stderr`, and the process terminates as if by calling `exit(1)`. This defensive check for allocation failure is always performed, regardless of whether `PERMUTE__` expands to `0` or `1`.

5.9.10 Joining

`<join._>` defines the macro `JOIN__` that configures the behavior of `join__` and `join_*` families; `JOIN__` records the `defined` state of `DEBUG` every time `<join._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise.

5.9.10.1 Synonyms

`<join._>` defines the type synonyms `Sentence` and `WSentence`: `Sentence` is a non-modifiable array of `Ptr(Char)`, and `WSentence` is a non-modifiable array of `Ptr(WChar)`; the modifiable counterparts are `Sentence_` and `WSentence_`.

NOTE For both array types, the element type is pointer to non-modifiable data (`Char` or `WChar`); in particular, other similar arrays of pointer to modifiable data (`Char_` or `WChar_`) will not be compatible by type.

5.9.10.2 `join__`

Syntax

`# include <join._>`
`join__` (*sentence* [, *separator* [, *range*]])
`join__1__` (*sentence*)
`join__2__` (*sentence* , *separator*)
`join__3__` (*sentence* , *separator* , *range*)

Constraints

sentence shall be a pointer to `Sentence` or `WSentence`.

If *sentence* is a pointer to `Sentence`, then *separator* shall be of type `String` or pointer to `Char`; otherwise *sentence* is a pointer to `WSentence`, and *separator* shall be of type `WString` or pointer to `WChar`.

range shall be an expression whose type is `Range (Int)`.

Semantics

`join__` invokes `join__`*n*`_` if the expanded argument sequence contains n arguments.

`join__1__` concatenates the (wide) strings pointed to by the elements of *sentence*, with a space between every pair of consecutive elements. `join__3__` concatenates only the elements at indices specified by *range*, and in the given order. For `join__2__` and `join__3__`, *separator* is placed between every pair of consecutive elements. In all cases, a newline is placed after the concatenated (wide) string, followed by a null (wide) character to mark the end.

If *sentence* is a pointer to `Sentence`, the outcome is a pointer to `String_` (modifiable array of characters); otherwise *sentence* is a pointer to `WSentence`, and the outcome is a pointer to `WString_` (modifiable array of wide characters). In both cases, the array allocation is obtained with `malloc` (or equivalent), and it can be deallocated by passing the pointer to `free`; the pointer is null if the required allocation is not available. The array is of incomplete type, and if the outcome is not null, then the array length can be obtained by calling `strlen` or `wcslen` function.

If an element of *sentence* is a null pointer, that element is ignored. The behavior is undefined if *separator* or any non-null element of *sentence* points to a (wide) character array that is not terminated by a null (wide) character.

When compiled with `JOIN__` expanding to `1`, only the argument pointers are asserted to be not null, as done by `notnull__`. When compiled with `JOIN__` expanding to `0`, the outcome is null if *sentence* is null. If *separator* is null, a (wide) space is placed between elements. If *range* is null, all non-null elements of *sentence* array are joined.

NOTE If the element type of *sentence* array is a pointer to `Char_` or `WChar_`, then *sentence* should be cast to `Ptr (Sentence)` or `Ptr (WSentence)`; the cast is safe and it is used to avoid qualifier mismatch in element type.

5.9.10.3 `join_*`

Syntax

`# include <join._>`
`join_` (*sentence* [, *separator* [, *range*]])
`join_1_` (*sentence*)
`join_2_` (*sentence* , *separator*)
`join_3_` (*sentence* , *separator* , *range*)

Constraints

The `join_` family shall have precisely the same constraints as those applicable for the `join__` family.

Semantics

`join_` family evaluates each expression exactly only once; rest of the semantics are same as `join__` family.

NOTE A "two-dimensional" array of characters or wide characters cannot be used with the `join__` and `join_` families, which expect pointer to an array of pointers (an "array of arrays" is different from an "array of pointers").

Chapter 6

Methods

A C_ method is essentially a pair of function pointers: protocol and procedure. The primary motivation behind this design is to separate behavior from implementation: protocol describes "*what*" needs to be done, whereas procedure specifies "*how*" it is actually accomplished. As an analogy, let us consider the example of making a cake: a protocol would describe only the externally observable features of the cake that are of interest to buyers; a procedure would be a detailed step-by-step recipe to prepare the cake from its ingredients, that is of concern to the baker.

On similar lines, the primary advantage of isolating behavior from implementation is that a caller needs to be concerned only with the protocol, and the exact implementation details of the procedure "*should* be" irrelevant. If a functionality or transformation is conceptually imagined as an opaque box, then protocol is an abstract specification of the external behavior, whereas procedure deals with the concrete machinery that operates inside the box.

6.1 `public` and `private`

`public` and `private` are object-like macros; the reference implementation defines them in `<specifiers._>` header. `public` expands to the keyword `inline`, and `private` expands to the keyword `static` (for internal linkage).

NOTE These macros are nothing but alternative names, and they do not add anything new to the language. However, one advantage of macros is that they can be easily undefined, which can be convenient for certain purposes; for example, the reference implementation provides inline definitions for several functions in header files, using the macro `public` instead of the keyword `inline`. The benefit is that the inline definitions can be made visible in multiple translation units, which can be used by compilers for static analysis and optimizations of function calls. An external definition is still required for each such function, so the file `lib.c_` first includes the file `<specifiers._>`, then undefines the macro `public`, and redefines it with an empty replacement text. This ensures that when the header files containing `public` function definition are included in `lib.c_`, they are no longer `inline` definitions.

6.2 `no_inline_`

The `inline` keyword is used to suggest that the code generated for calling a function should be "efficient". For most purposes, efficiency is desirable in terms of runtime, and if the generated code for a function definition is small, the code can be substituted at call site(s), thereby avoiding the overhead of a function call. However, substituting the code of another function at multiple call sites can increase size of the object file; also, too many inline substitutions inside a function can bloat its code to an extent that the resulting code itself becomes unsuitable for inlining.

The inline keyword is only a hint to the compiler (similar to the register keyword); however, even in the absence of an explicit hint from the programmer, translators can still perform inline substitutions at their own discretion, if the function definition is visible in the same translation unit as the function call. Increasing code size to gain speed (space-time tradeoff) may not always work as expected, and large code expansion can actually degrade performance (depending on several factors of the execution environment, such as instruction caching). In particular, minimizing the size of an executable is an important concern for memory constrained devices, even if that comes at the cost of a tolerable increase in execution time. In several contexts, it may be desirable to have a portable mechanism to explicitly disable inline substitutions at specific call sites, without the use of compiler-specific flags.

Syntax

no_inline_ (*function*)

Semantics

If *function* is a function name or a function pointer, then invoking the outcome of no_inline_ ensures that the call is not inlined, even if an inline definition is visible in the translation unit. The outcome is an expression that compares equal with *function*, and it has the same function type as *function*; if *function* is a function pointer, then type of the outcome is the corresponding function type obtained on dereferencing the pointer.

6.3 Contracts

The concept of protocols is nothing but a natural extension of the idea behind function declarations: a declaration only specifies the return type and parameter types of a function, for static type checking of function calls and performing implicit argument promotions or conversions during translation. A protocol describes additional checks to be performed on the arguments and return value during execution. In a sense, a protocol is a form of contract that expects the caller to ensure certain prerequisites for the arguments, and promises that the return value will meet certain criteria. These requirements are established with the help of pre-conditions and post-conditions.

The reference implementation provides <contract._> which includes two other headers: <pre._> and <post._>. <pre._> provides facilities for specifying pre-conditions, and <post._> provides facilities for writing post-conditions.

NOTE Pre-conditions are input-oriented, whereas post-conditions are output-oriented.

6.3.1 Pre-conditions

Syntax

pre_ (*condition* [, *text*="*condition*" [, *site*=SITE]])
pre_1_ (*condition*)
pre_2_ (*condition* , *text*)
pre_3_ (*condition* , *text* , *site*)

Constraints

condition shall be a scalar expression. *text* shall be a string. *site* shall be of type Site.

The pre_ family can only be used inside blocks where the identifier _site refers to a variable of type Site.

NOTE Site_ is a synonym for struct Site having three members: func, file, and line; func and file are pointers to Char, whereas line is of type Int. Both the type and the synonym pair are defined in <assert._>.

Semantics

pre_ invokes pre_*n*_ if the expanded argument sequence contains *n* arguments.

If *condition* compares equal to zero, pre_1_ prints a diagnostic message of the following form:

Pre-condition failed: *condition*, function *function-identifier*, file *file-name*, line *line-number*.
Called from function *caller-function-identifier*, file *call-site-file-name*, line *call-site-line-number*.

caller-function-identifier, *call-site-file-name*, and *call-site-line-number* are respectively obtained from `_site.func`, `_site.file`, and `_site.line`; if either `_site.func` or `_site.file` is a null pointer, an empty string is used instead.

`pre_2_` and `pre_3_` print *text* instead of *condition* in the message. `pre_1_` and `pre_2_` obtain *function-identifier*, *file-name*, and *line-number* from `__func__`, `__FILE__`, and `__LINE__` (respectively), whereas `pre_3_` obtains these "source code coordinates" from *site*; if `(`*site*`).func` or `(`*site*`).file` is null, an empty string is used instead.

In all cases, the diagnostic message is written to the standard error stream `stderr`, and the process terminates as if by calling `exit(1)`. *text* and *site* are always evaluated once; their results are discarded if *condition* is non-zero.

6.3.2 Post-conditions

Syntax

`post_` (*condition* [, *text*="*condition*"] [, *site*=`SITE`])
`post_1_` (*condition*)
`post_2_` (*condition* , *text*)
`post_3_` (*condition* , *text* , *site*)

Constraints

condition shall be a scalar expression. *text* shall be a string. *site* shall be of type `Site`.

Semantics

`post_` invokes `post_`*n*`_` if the expanded argument sequence contains *n* arguments.

If *condition* compares equal to zero, `post_1_` prints a diagnostic message of the following form:

`Post-condition failed:` *condition*`, function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

`post_2_` and `post_3_` print *text* instead of *condition* in the message. `post_1_` and `post_2_` obtain *function-identifier*, *file-name*, and *line-number* from `__func__`, `__FILE__`, and `__LINE__` (respectively), whereas `post_3_` obtains these "source code coordinates" from *site*; if `(`*site*`).func` or `(`*site*`).file` is null, an empty string is used.

In all cases, the diagnostic message is written to the standard error stream `stderr`, and the process terminates as if by calling `exit(1)`. *text* and *site* are always evaluated once; their results are discarded if *condition* is non-zero.

6.3.3 Example

Searching is a basic functionality that is required for most applications. A bare minimum design would require two inputs: a collection and an element to find in the collection. If the element is found and the collection is a sequence, it may be desirable to know the position at which element was found; usually the first occurrence is returned.

```
#include <c._>

Size_ finder
(   let Size len,
    let Ptr (Int [len]) arr,
    let Int  key
)
begin
    loop_(0, len - 1)
        stop_((*arr)[_i_] == key, _i_)
    end
    return len;
end
```

The given code looks through an array in increasing order of indices, stopping when the element is found, and index of the first occurrence is returned; if return value equals array length, it indicates the element was not found.

The function `finder` is an example of solver that actually does the work of going through an array and comparing each of its elements with the given search value. It assumes that `arr` points to a valid `Int` array having (at least) `len` elements. The implementation uses an iterative approach to access the array elements from low to high indices.

To express the behavior of searching without specifying any particular implementation detail, we can write another function that verifies the outcome of `finder` for a given pair of arguments: this would be a post-condition. Additionally, this function would also be responsible for argument validation: in this example, it can check that the array pointer is not null, which would be a pre-condition. The following function `find` concretizes this specification.

```
#include <c._>

Size_ finder(Size len, Ptr (Int [len]) arr, Int key);

Size_ find
(   let Site _site,
    let Size len,
    let Ptr (Int [len]) arr,
    let Int  key
)
begin
    pre_(len != 0);
    pre_(arr != NULL);
    Var pos = finder(len, arr, key);
    post_(pos <= len);
    if (pos) loop_(0, pos - 1)
        post_((*arr)[_i_] != key);
    end
    post_((pos < len  implies   (*arr)[pos] == key));
    return pos;
end
```

In a sense, both pre-conditions and post-conditions are a form of guard clauses that allow execution to continue only if the given condition is satisfied. In this example, there are two pre-conditions: validating that the length is non-zero and the array pointer is not null (note that null is just one invalid pointer that can be easily diagnosed; there is no general mechanism in C to detect whether a pointer refers to a valid object for the given type).

Fulfilling the pre-conditions is a responsibility of the calling function; a function can be called from multiple locations, and if any pre-condition is unsatisfied, it is important to know the precise invocation that caused a violation. The limitation of writing pre-conditions as assertions (such as with the `assert` macro from `<assert.h>`) is that it only reports a violation for an argument, not the offending invocation that supplied the bad argument. In our example, the first parameter `_site` is meant to capture call site details, which is used by the `pre_` family to provide additional information in the diagnostic message; for an invocation of `find`, the non-modifiable lvalue `SITE` can be used as the first argument to capture the values of `__func__`, `__FILE__`, and `__LINE__` at the call site.

After the pre-conditions are found to be satisfied, `find` invokes the solver `finder` and verifies that the return value satisfies the given post-conditions. The first post-condition verifies that the outcome is within a valid range. The second post-condition verifies that for all indices less than the outcome, none of the elements are equal to the search value; this describes the specification of finding the first occurrence of search value when moving from low to high indices. If outcome is less than array length, it is a valid index, and the third post-condition verifies that the corresponding element is equal to search value. A verifier should not alter the return value, and treat it as read-only: once the post-conditions are satisfied, verifier simply returns the unmodified outcome given by solver.

The following code shows how the invocation can be simplified with a wrapper macro: the first argument would typically provide call site details with `SITE`, and length of a complete array can be inferred with `length__`. Most invocations would provide these details in the same manner, so they need not be specified for each call: the macro `find__` eliminates such boilerplate code in source files by pushing extra arguments necessary for the function call.

```
#include <c._>

Size_ find(Site _site, Size len, Ptr (Int [len]) arr, Int key);
#define find__(arr, key) find(SITE, length__(arr), arr, key)

Int_ main()
begin
    Int arr[] = {10, 20, 30, 40, 50,};
    Var pos = find__(&arr, 40);
    if (pos == length__(&arr)) puts("Element not found");
    else print_("Element found at index", pos);
end
```

NOTE Throughout the rest of this documentation, we shall use the term "validation" for pre-conditions on the arguments supplied by the caller, and "verification" for post-conditions on the return value given by the solver.

Recommended practice

Diagnostic messages become meaningful and precise when multiple conditions are specified separately; if they are written as a single logical conjunction, it becomes more tedious to find out which condition was violated.

6.4 Prototype

There are four polymorphic families for declaring, defining, and invoking C_ methods: `prototype_`, `protocol_`, `procedure_`, and `call_`. The `prototype_` family is primarily used in header files for method declarations. Similar to conventional function declarations, `prototype_` declarations are used for static type checking and argument promotions or conversions for method calls; however, parameter names are significant for `prototype_` declarations, and they can be referred at call sites for out of order associations of actual arguments with formal parameters.

Syntax

`prototype_` ([*return-type* ,] (*prefix* , *method-name*) [, *parameter-tuples*])

`prototype_0_` ([*return-type* ,] (*prefix* , *method-name*) , *parameter-tuples*)
`prototype_0_1_` (*return-type* , (*prefix* , *method-name*) , *parameter-tuples*)
`prototype_0_2_` ((*prefix* , *method-name*) , *parameter-tuples*)

`prototype_1_` ((*prefix* , *method-name*))

`prototype_2_` (*return-type* , (*prefix* , *method-name*))
`prototype_2_1_` (*return-type* , (*prefix* , *method-name*))
`prototype_2_` ((*prefix* , *method-name*) , (*parameter-type* , *parameter-name*))
`prototype_2_2_` ((*prefix* , *method-name*) , (*parameter-type* , *parameter-name*))

Constraints

return-type shall not be an array type, an incomplete type, or a function type. *parameter-tuples* shall be a comma-separated list of parenthesized tuples, each of them having the form `(` *parameter-type* `,` *parameter-name* `)`.

Type qualifiers (if any) in *return-type* are ignored. *parameter-type* shall not contain any storage-class specifier.

A *parameter-name* shall not be used to specify array length in another *parameter-type*, or for any other purpose.

If *parameter-type* has type “array of T”, it is adjusted as “pointer to T”, while preserving the qualifiers (if any) of element type T. If *parameter-type* is of function type, it is adjusted as pointer to a function of the same type.

NOTE The two adjustments on *parameter-type* are also performed for conventional function parameters in C.

Semantics

`prototype_` invokes `prototype_0_` if the expanded argument sequence has more than two arguments; otherwise it invokes `prototype_`n`_` if the expanded argument sequence contains n arguments, with n not exceeding two.

The first argument is peeled and expanded: if the resulting list has a single element, it is considered as *return-type*; otherwise it shall have two elements, *prefix* and *method-name*, which are used to generate method identifiers.

`prototype_0_` invokes `prototype_0_`n`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `prototype_0_1_` declares a method that accepts a sequence of arguments as specified by *parameter-tuples*, and returns a value of type *return-type*. `prototype_0_2_` uses `Void_` as the return type.

`prototype_1_` declares a method that can accept a variable number of arguments (zero or more) whose types are not specified in the declaration, and the method does not return anything (return type is `Void_`).

`prototype_2_` invokes `prototype_2_`n`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `prototype_2_1_` declares a method that can accept a variable number of arguments having unspecified types, and returns a value of type *return-type*. Return type of a `prototype_2_2_` declaration is `Void_`.

In all cases, an extra argument named `_site` of type `Site` is pushed before the parameter list, as per parameter tuples or ellipsis (`...`) for variable arguments; `_site` is intended to store call site details during method invocations.

NOTE There is no mechanism to specify a method with named parameters along with variable argument list.

Recommended practice

prefix and *method-name* should not start or end with an underscore, and neither of them should contain two consecutive underscores; the latter is reserved for name mangling that can be performed by C_ implementations.

EXAMPLE The following prototype declares a method for sorting an array of integers. It has three parameters: the first is an implicit parameter named `_site` of type `Site`, followed by `len` and `arr`; the return type is `Void_`.

```
#include <c._>

prototype_((Integers, sort), (Size, len), (Ptr (Int_ []), arr))
```

6.4.1 Identifiers

Each `prototype_` declaration introduces a set of identifiers that are generated from *prefix* and *method-name* through name mangling; the precise naming scheme used by the reference implementation is described in appendix B.

NOTE An implementation is also permitted to declare additional identifiers using reserved names.

6.4.1.1 `Method_`

Syntax

`Method_ (` *prefix* `,` *method-name* `)`

Semantics

`Method_` specifies the function type declared by `prototype_`; in all cases, the first parameter is of type `Site`, followed by the *parameter-tuples* specified in `prototype_`, or ellipsis (`...`) if none were given.

6.4.1.2 `method_`

Syntax

`method_` (*prefix* , *method-name*)

Semantics

`method_` specifies a non-modifiable array of two function pointers: the function type is given by `Method_` with the same *prefix* and *method-name*. These are pointers to the proxy and verifier functions, which are described next.

NOTE `typeof (method_(`*prefix*`,` *method-name*`))` is same as `Ptr (Method_(`*prefix*`,` *method-name*`)) [2]`.

6.4.1.3 `verifier_`

Syntax

`verifier_` (*prefix* , *method-name*)

Semantics

`verifier_` gives the mangled name of the protocol: it is the function that describes the behavior ("*what*" requires to be done) by establishing pre-conditions on the arguments and post-conditions on the return value.

6.4.1.4 `proxy_`

Syntax

`proxy_` (*prefix* , *method-name*)

Semantics

`proxy_` gives pointer to a function having the same type as specified by `Method_`.

The header `<method._>` defines an object-like macro `METHOD__` that records the `defined` state of `DEBUG` macro every time `<method._>` is included: it expands to `1` if `DEBUG` was defined, and `0` otherwise. Conventionally, both parts of a method (protocol and procedure) are compiled in one translation unit: if the macro `METHOD__` expands to `1` when the protocol is defined, the function pointer specified by `proxy_` refers to the protocol; otherwise `METHOD__` shall expand to `1`, and the proxy points to a `private` function (internal linkage) that ignores the first argument of type `Site` and invokes the procedure with rest of the arguments. Any return value is forwarded to the caller.

NOTE Unlike a protocol, function declaration of a procedure is not modified to push an extra `Site` parameter at the beginning: this causes a mismatch between their function types. `proxy_` is an intermediary that provides a common type for both the protocol and the procedure. However, the protocol can be bypassed only if the translation unit containing the method has been compiled in non-debugging mode, *i.e.* proxy points to a function that directly calls the procedure. The only purpose of this extra function call overhead is to remove the `Site` argument.

6.5 Protocol

A protocol is nothing but a layer of abstraction between the caller and the procedure. A protocol itself does not generate the return value: if all the argument values satisfy their respective pre-conditions, then the protocol invokes a procedure, which gives the return value to the protocol; the protocol then verifies if the return value meets the given post-conditions, and on success, it forwards the same return value to the caller without modifying it.

Syntax

```
# include <method._>

protocol_     ( [return-type ,] ( prefix , method-name ) [, parameter-tuples] )

protocol_0_   ( [return-type ,] ( prefix , method-name )  , parameter-tuples  )
protocol_0_1_ (  return-type ,  ( prefix , method-name )  , parameter-tuples  )
protocol_0_2_ ( ( prefix , method-name ) , parameter-tuples )

protocol_1_   ( ( prefix , method-name ) )

protocol_2_   (  return-type ,  ( prefix , method-name ) )
protocol_2_1_ (  return-type ,  ( prefix , method-name ) )
protocol_2_   ( ( prefix , method-name ) , ( parameter-type , parameter-name ) )
protocol_2_2_ ( ( prefix , method-name ) , ( parameter-type , parameter-name ) )
```

Constraints

return-type shall not be an array type, an incomplete type, or a function type. *parameter-tuples* shall be a comma-separated list of parenthesized tuples, each of them having the form `(` *parameter-type* `,` *parameter-name* `)`.

Type qualifiers (if any) in *return-type* are ignored.

If *parameter-type* has type "array of T", it is adjusted as "pointer to T", while preserving the qualifiers (if any) of element type T. If *parameter-type* is of function type, it is adjusted as pointer to a function of the same type.

Each member of the `protocol_` family starts a block that shall be lexically closed with `end` (or its equivalent).

NOTE Names used for parameters can be redefined inside a `protocol_` block, without creating an inner block.

Semantics

The `protocol_` family is used to provide function definition for a verifier; it also declares the procedure prior to its definition. The proxy function pointer is defined in the same translation unit: if the macro `METHOD__` expands to `1`, then the proxy points to the protocol; otherwise `METHOD__` shall expand to `0`, and the proxy points to a function with internal linkage that directly calls the procedure (without `_site` parameter) and forwards any return value.

`protocol_` invokes `protocol_0_` if the expanded argument sequence has more than two arguments; otherwise it invokes `protocol_`*n*`_` if the expanded argument sequence contains n arguments, with n not exceeding two.

The first argument is peeled and expanded: if the resulting list has a single element, it is considered as *return-type*; otherwise it shall have two elements, *prefix* and *method-name*, which are used to generate method identifiers.

`protocol_0_` invokes `protocol_0_`*n*`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `protocol_0_1_` defines a method that accepts a sequence of arguments as specified by *parameter-tuples*, and returns a value of type *return-type*. `protocol_0_2_` uses `Void_` as the return type.

`protocol_1_` defines a method that can accept a variable number of arguments (zero or more) whose types are not specified in the declaration, and the method does not return anything (return type is `Void_`).

`protocol_2_` invokes `protocol_2_`*n*`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `protocol_2_1_` defines a method that can accept a variable number of arguments having unspecified types, and returns a value of type *return-type*. Return type of a `protocol_2_2_` definition is `Void_`.

In all cases, an extra argument named `_site` of type `Site` is pushed before the parameter list, as per parameter tuples or ellipsis (`...`) for variable arguments; `_site` is intended to store call site details during method invocations.

A protocol definition always has external linkage. If `private` (equivalent to the keyword `static`) is specified before `protocol_`, internal linkage is applicable for the procedure, not the protocol.

EXAMPLE The basic idea behind any sorting algorithm is to move around some elements to achieve a certain ordering. The following protocol defines a verifier to check that the outcome is sorted in non-decreasing order.

```
#include "Integers._"

private
protocol_((Integers, sort),
(let Size, len),
(let Ptr (Int_ [len]), arr))
    pre_(len != 0);
    pre_(arr != NULL);
    solver_(Integers, sort)(len, arr);
    guard_(len > 1)
    loop_(1, len - 1)
        post_((*arr)[_i_ - 1] <= (*arr)[_i_]);
    end
end
```

The protocol has three parameters: the first is an implicit parameter named `_site` of type `Site`, followed by `len` and `arr`; the return type is `Void_`. There are two pre-conditions: validating that the length is non-zero and the pointer is not null; if any pre-condition fails, call site information for the current invocation is obtained from the implicit parameter named `_site`. The post-condition is within a loop, checking that each element does not exceed its next element. The procedure is invoked after validating pre-conditions and before verifying post-conditions.

6.6 Procedure

Procedure defines the solver that operates on the arguments and possibly generates a return value (or side effects). Unlike protocols, function definition (and declaration) of a procedure does not push an extra `Site` parameter at the start: this is because a procedure is neither required nor supposed to know from where it is invoked. The information provided by `_site` is primarily meant for use by the `pre_` family whenever a pre-condition is not satisfied, in order to track the call site that caused a violation; however, pre-conditions (and also post-conditions) should be established only within protocols, so there is no need to forward the original call site details to the procedure.

NOTE Should the need arise, programmers can supply call site details to the procedure by explicitly specifying a parameter of type `Site`, and also providing a corresponding argument for the same when the function is called.

Syntax

`procedure_` ([*return-type* ,] (*prefix* , *method-name*) [, *parameter-tuples*])

`procedure_0_` ([*return-type* ,] (*prefix* , *method-name*) , *parameter-tuples*)
`procedure_0_1_` (*return-type* , (*prefix* , *method-name*) , *parameter-tuples*)
`procedure_0_2_` ((*prefix* , *method-name*) , *parameter-tuples*)

`procedure_1_` ((*prefix* , *method-name*))

`procedure_2_` (*return-type* , (*prefix* , *method-name*))
`procedure_2_1_` (*return-type* , (*prefix* , *method-name*))
`procedure_2_` ((*prefix* , *method-name*) , (*parameter-type* , *parameter-name*))
`procedure_2_2_` ((*prefix* , *method-name*) , (*parameter-type* , *parameter-name*))

Constraints

The `procedure_` family shall have precisely the same constraints as those applicable for the `protocol_` family.

Semantics

`procedure_` family is used to provide function definition for a solver; if `private` (or equivalently `static`) is specified before the associated `protocol_` definition, it means that the `procedure_` definition has internal linkage.

`procedure_` invokes `procedure_0_` if the expanded argument sequence has more than two arguments; otherwise it invokes `procedure_`n`_` if the expanded argument sequence contains n arguments, with n not exceeding two.

The first argument is peeled and expanded: if the resulting list has a single element, it is considered as *return-type*; otherwise it shall have two elements, *prefix* and *method-name*, which are used to generate procedure identifier.

`procedure_0_` invokes `procedure_0_`n`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `procedure_0_1_` defines a function that accepts a sequence of arguments as specified by *parameter-tuples*, and returns a value of type *return-type*; `procedure_0_2_` uses `Void_` as the return type.

`procedure_1_` defines a `Void_` function with a single parameter `_args_` of type `VA_list_` (declared in `<stdarg._>` as a synonym for `va_list`): it provides access to a variable number of arguments given to the protocol or the proxy.

`procedure_2_` invokes `procedure_2_`n`_` if the first argument expands to a list with n elements on being subjected to the `peel_` macro. `procedure_2_1_` defines a function having a single parameter `_args_` of type `VA_list_`, and it returns a value of type *return-type*. Return type of a `procedure_2_2_` definition is `Void_`.

EXAMPLE The following procedure implements a well-known adaptive version of bubble sort algorithm.

```
#include "Integers/sort._"

procedure_((Integers,sort),
(let Size, len),
(let Ptr (Int_ [len]), arr))
    guard_(len && arr)
    Var_ last_ = len - 1;
    Var_ omega_ = (Size)0;
    do for (Var_ i_ = omega_ = 0; i_ < last_; i_++)
        if_((*arr)[i_] > (*arr)[i_ + 1])
            Var tmp = (*arr)[omega_ = i_];
            (*arr)[i_] = (*arr)[i_ + 1];
            (*arr)[i_ + 1] = tmp;
        end
    while ((last_ = omega_));
end
```

In the above code, the array is conceptually divided into two parts: the left side is assumed to be unsorted, and the right side is sorted. The variable `last_` marks the last index of the left side; `last_` is initialized to `len - 1`, so the left side spans the entire array, and the right side is initially empty. The inner `for` loop "bubbles up" the maximum element of the left side to its end, by swapping consecutive elements that are out of order. Once the maximum value is moved to the `last_` index, it is augmented to the right side by decreasing the value of `last_`; this element will not exceed any previous maximum that was bubbled up earlier, so the right side remains sorted.

As an enhancement, another variable `omega_` keeps track of the latest (highest) index at which a swapping was performed: This implies that elements from `omega_` to `last_` are already sorted, so they can be combined with the right side. Doing this is trivial: we simply set `last_` to `omega_` after completing each round of bubbling up the maximum. Note that the assignment `(last_ = omega_)` also acts the condition of the `do-while` loop, which stops when `omega_` remains zero: this indicates that the left side is empty, and the right side now spans the entire array.

The optional enhancement makes the algorithm "adaptive" by achieving linear time complexity in the best case: if the array is already sorted, then no swapping is performed, and `omega_` remains zero after the first round of bubbling. This indicates that elements from index `omega_` (zero) to `last_` (`len - 1`) are sorted, spanning the entire array. After the first round, `last_` is set to `omega_`, which stops the `do-while` loop as its condition becomes zero.

6.6.1 `solver_`

Syntax
`solver_` (*prefix* , *method-name*)
Semantics
`solver_` gives mangled name of the function defined by `procedure_`; this function is also declared by `protocol_`.
NOTE `prototype_` family does not declare the identifier given by `solver_`: this is because a procedure can be defined with internal linkage, in which case the function will not be visible outside of its own translation unit.

6.6.2 Multiple procedures

The source file `"Integers/sort._"` containing the protocol definition is located in `include/` directory. Examples in this documentation follow a practice of keeping prototype declarations and protocol definitions in `include/` directory: files having prototype declarations are included for method invocations, whereas a file having a protocol definition is included only by those files that provide a corresponding procedure definition. Files having procedure definitions are named with `.c_` as filename extension, and these files are placed within the `compile/` directory.

The following subsections describe two endpoint-based strategies for sorting: their source codes are available in the files `hourglass.c_` and `burrow.c_`, both located in the `Integers/` subdirectory within the `compile/` directory. Both of these translation units include the header file `"Integers/sort._"` containing the protocol definition.

NOTE As these files provide definitions for the same functions, at most one of their object codes can be linked.

6.6.2.1 Hourglass partitioning scheme

Most partitioning schemes for quicksort algorithm are unbalanced, as one of the partitions can be significantly larger than the other. The hourglass scheme achieves a balanced partitioning by dividing the array at the middle as the first step: elements of the left half are organized as an inverted binary max heap, whereas elements of the right half are organized as a binary min heap. An inverted binary max heap is constructed backwards, where the last element (of the left half) is the maximum value acting as the root element, with the elements before it (if any) acting as children and successors. The right side is organized as a binary min heap, so it starts with the minimum value as the root element. If the maximum value of left side (root of inverted max heap) is less than minimum value of right side (root of min heap), then these two (adjacent elements) are interchanged, and each side is fixed to maintain the heap property, by sinking the incoming root element “downwards”, i.e. towards the leaf elements.

The process is repeated until the left root does not exceed the right root, indicating that all elements of the left side do not exceed any element on the right side, so no further swapping is required from one side to the other; once this is established, each side is recursively sorted using the same mechanism. The following functions provide a modular implementation of this algorithm, intended as an alternative procedure to our earlier bubble sort approach.

```
#include "Integers/sort._"

private Void_ fix_inv_max_heap
(   let Size len,
    let Ptr (Int_ [len]) arr,
    let Size_ i_
)
begin
    Var_ last  = len - 1;
    Var_ next_ = (Size)0;
```

```
    begin
        Var left  =  i_<<1 | 1;
        guard_(left < len)
        Var right = left+1;
        if((right<len implies  (*arr)[last - left] >= (*arr)[last - right]))
            if ((*arr)[last - left] > (*arr)[last - i_]) next_ = left;
            else break;
        elif  ((*arr)[last - right] > (*arr)[last - i_]) next_ = right;
        else break;
        Var temp = (*arr)[last - i_];
        (*arr)[last - i_] = (*arr)[last - next_];
        (*arr)[last - (i_ = next_)] = temp;
    again
end

private Void_ fix_min_heap
(   let Size len,
    let Ptr (Int_ [len]) arr,
    let Size_ i_
)
begin
    Var_ next_ = (Size)0;
    begin
        Var left  =  i_<<1 | 1;
        guard_(left < len)
        Var right = left+1;
        if((right<len implies  (*arr)[left] <= (*arr)[right]))
            if ((*arr)[left] < (*arr)[i_]) next_ = left;
            else break;
        elif  ((*arr)[right] < (*arr)[i_]) next_ = right;
        else break;
        Var temp   = (*arr)[i_];
        (*arr)[i_] = (*arr)[next_];
        (*arr)[i_  = next_] = temp;
    again
end

private Void_ make_inv_max_heap
(   let Size len,
    let Ptr (Int_ [len]) arr
)
begin
    Var_ i_ = len>>1;
    do fix_inv_max_heap(len, arr, i_); while (i_--);
end
```

```
private Void_ make_min_heap
(   let Size len,
    let Ptr (Int_ [len]) arr
)
begin
    Var_ i_ = len>>1;
    do fix_min_heap(len, arr, i_); while (i_--);
end

procedure_((Integers, sort),
(let Size, len),
(let Ptr (Int_ [len]), arr))
    guard_(len>1 &&  arr)
    Var left  = arr;
    Var llen  = len >> 1;
    Var right = (Ptr (Int_ []))(*arr + llen);
    Var rlen  = len - llen;
    make_inv_max_heap(llen, left);
    make_min_heap(rlen, right);
    Var max_left  = *right - 1;
    Var min_right = *right;
    until_(*max_left <= *min_right)
        Var  temp = *max_left ;
        *max_left = *min_right;
        *min_right = temp;
        fix_inv_max_heap(llen, left, 0);
        fix_min_heap(rlen, right, 0);
    end
    solver_(Integers, sort)(llen - 1, left);
    solver_(Integers, sort)(rlen - 1, (Ptr (Int_ []))(*right + 1));
end
```

EXAMPLE If the "heapified" partitions are drawn one above the other, it visually resembles an hourglass.

```
14 13 11 10
 15     12
     16
     17
 21     18
23 22 20 19
```

The above "hourglass" would be physically represented in the linear sequence `14 13 11 10 15 12 16` followed by `17 21 18 23 22 20 19`. Neither of them are sorted from the inside, which can be done using similar hourglass formations recursively on each side. Note that the root elements `16` and `17` are always placed in their correct positions as per the final sorted array, so they need not be included in the recursive calls for sorting each side.

NOTE An asymptotic upper bound on the running time of this algorithm can be obtained from the recurrence relation $T(n) = n+n\log(n/2)+2T(n/2)$. n denotes the complexity of heap formation for each partition $(n/2+n/2)$. $n\log(n/2)$ is the worst-case complexity of element exchange, when $n/2$ swaps are needed; $\log(n/2)$ is the heap height. $T(n/2)$ denotes the time taken to sort one side. This approach is non-adaptive even if the array is already sorted, since a heap formation does not confirm a total linear order; hence all recursive calls have to be made in any case.

6.6.2.2 Burrowing merge strategy

A simpler and more efficient alternative is to sort each side individually, and then merge them into a sorted array. The conventional approach for merging two sorted arrays requires a buffer array for storing the sorted array, which is later copied back to the original. We shall implement a recursive merge strategy that eliminates the need of an auxiliary array. This approach is a natural extension of the hourglass strategy of comparing only the endpoint elements: instead of looking at just the extremities, our new approach would gradually "burrow" into each side, starting at middle. As both sides are sorted before merging, there is no need for heaps (they are implicitly present).

We start by positioning a left index at the end of left partition, and a right index at the start of right partition; these refer to the maximum element on left side and minimum element on right side. If left side maximum is greater than right side minimum, then left index is decremented (moved further left), and right index is incremented (moved further right). This process is repeated until the element at left index does not exceed the element at right index, or one of the sides is exhausted. At this point, pairwise swapping is started from the current position of left index (it may have to be incremented once) and original right index (start of right side); after each swapping between left and right partitions, both indices are incremented. The pairwise exchange continues till the end of each side.

At the end of this stage, we end up with identical scenarios in each partition: each side has two sub-partitions, which are individually sorted; these can be merged recursively by the same strategy on each side, as shown below.

```
#include "Integers/sort._"

private Void_ burrow
(   let Ptrdiff alpha,
    let Ptr (Int_ []) arr,
    let Ptrdiff mid,
    let Ptrdiff omega
)
begin
    guard_(alpha <= mid  &&   mid < omega)
    Var_  left_ = mid;
    Var_ right_ = mid+1;
    until   ((alpha<=left_ &&  right_<=omega
    implies (*arr)[left_] <= (*arr)[right_]))
        left_--, right_++;
    Var left = left_;
    for_(right_ = mid+1; ++left_ <= mid; right_++)
        Var tmp = (*arr)[left_];
        (*arr)[ left_] = (*arr)[right_];
        (*arr)[right_] = tmp;
    end
    burrow(alpha, arr, left, mid);
    burrow(mid+1, arr, right_-1, omega);
end

private Void_ sort
(   let Size  alpha,
    let Ptr (Int_ []) arr,
    let Size  omega
)
```

```
begin
    guard_(alpha < omega)
    Var mid = (alpha + omega)>>1;
    sort(alpha, arr, mid);
    sort(mid+1, arr, omega);
    burrow(alpha, arr, mid, omega);
end

procedure_((Integers, sort),
(let Size, len),
(let Ptr (Int_  [len]), arr))
    guard_(len &&  arr)
    sort(0, arr, len - 1);
end
```

Correctness of this approach can be proved based on the following three observations:

- In the burrowing phase, each pair of index movement characterizes a swap operation for out-of-order elements without actually performing it. If such a hypothetical exchange is immediately done at left index i (having element l_i) and right index j (having element r_j), then incoming element l_i at right index r will be greater than all elements on the left side: this is because l_i is trivially greater than all elements l_0 through l_{i-1}, and l_i was found to be greater than r_j, so l_i is transitively greater than all elements r_0 through r_{i-1}, which were earlier swapped (hypothetically) to the left side through burrowing inwards. A similar argument can be made for the incoming element r_j at the left index i, so both elements l_i and r_j are moved to the correct partitions.
- If burrowing stops at left index i (having element l_i) and right index j (having element r_j) then $l_i \le r_j$. Transitively, an element l_{i-1} before l_i and r_{j+1} after r_j follow the ordering $l_{i-1} \le l_i \le r_j \le r_{i+1}$; this is also trivially true for all other elements before l_i and after r_j. Additionally, the current r_{j-1} was swapped earlier from the left side occurring after l_i, so l_i and all elements before it cannot exceed r_{j-1} (as both sides were already sorted to begin with). The same argument also applies for all right side elements before r_j, which were swapped from the left side while burrowing inwards, all of them originally occurring after l_i.
- In the hypothetical "eager exchange" model, elements of the left side are guaranteed to not exceed elements on the right side at the end of burrowing; the disadvantage is that the incoming elements will be in reverse order on either side. The correctness will not be affected if we postpone the swappings to a separate phase after finding where the burrowing stops; until then only the indices are shifted without moving elements. The exchange phase preserves the order of the transferred elements: from an abstract perspective of this "lazy exchange" model, a sorted block from the right side of left partition is swapped with an equal-sized sorted block from the left side of right partition. Remaining elements were already sorted, so after the exchange phase, each side gets sub-divided into two sorted sub-parts (though not necessarily into equal-sized halves).

NOTE Despite the absence of an additional buffer array, this approach cannot be regarded as truly "in-place", which only permits a constant amount of space overhead that does not depend on the number of array elements. Due to the recursive nature of our code, practical implementations would require additional call stack space for each (non-tail) function call, so the actual space complexity is $O(\log n)$ (where n is the number of elements). However, the merge strategy is naturally adaptive: if the overall array is already sorted, then the burrowing does not proceed inwards, and the recursive calls for merging are not performed; the initial calls for sorting each partition require linear time, as given by the recurrence relation $T(n) = 2T(n/2) + 1$ (one comparison between endpoint elements at the start of burrowing phase). Another benefit worth highlighting is that the approach is highly parallelizable, which facilitates efficient practical implementations on environments where multiple processing cores are available.

6.7 Invocation

The `call_` family is used to invoke methods that have been declared with the `prototype_` family. The header <call._> is used to configure debugging behavior of the `call_` family; this header also defines the object-like macro `CALL__`, which records the `defined` state of `DEBUG` macro every time <call._> is included. If `DEBUG` remains defined before the most recent inclusion of <call._>, then `CALL__` expands to `1`; otherwise `CALL__` expands to `0`.

Syntax

```
# include <call._>

call_     ( method [, argument-list] )
call_     ( ( prefix , method-name ) [, argument-list] )

call_0_   ( method , argument-list )
call_0_1_ ( method , argument-list )
call_0_   ( ( prefix , method-name ) , argument-list )
call_0_2_ ( ( prefix , method-name ) , argument-list )

call_1_   ( method )
call_1_1_ ( method )
call_1_   ( ( prefix , method-name ) )
call_1_2_ ( ( prefix , method-name ) )
```

Constraints

method shall be an array of two function pointers, and the function type shall accept a `Site` as the first argument. If *method* is given, then *argument-list* shall not specify any named argument of the form `.`*parameter* `=` *argument*.

A prototype declaration for the method identified by `(` *prefix* `,` *method-name* `)` shall occur prior to its use in `call_`. If it accepts a variable number of arguments, then `(` *prefix* `,` *method-name* `)` cannot be directly used for invocation: in such cases, the *method* array shall be specified as `method_ (` *prefix* `,` *method-name* `)` (or equivalent).

argument-list shall be a non-empty sequence of expressions, and if the method does not accept a variable number of arguments, then the values in *argument-list* are subjected to default argument promotions as per the parameter types declared in the prototype; none of the arguments shall not violate any type constraints. If the argument corresponding to the last parameter is followed by another argument, the latter shall be a named argument of the form `.`*parameter* `=` *argument*, where *parameter* shall be the name of some parameter in the prototype declaration.

NOTE If *method* is a comma expression, it needs to be doubly parenthesized for `call_`, `call_0_`, and `call_1_`.

Semantics

`call_` invokes `call_1_` if the expanded argument sequence is a singleton; otherwise it invokes `call_0_`.

The first argument is peeled and expanded: if the resulting list has a single element, it is considered as *method*; otherwise it shall have two elements, *prefix* and *method-name*, which are used to generate method identifiers.

`call_0_` invokes `call_0_`*n*`_` if the first argument expands to a list with *n* elements on being subjected to the `peel_` macro. `call_1_` invokes `call_1_`*n*`_` if the first argument expands to a list with *n* elements on being subjected to `peel_`. `call_0_` is used when some explicit argument is given for the invocation, and `call_1_` is used otherwise.

In all cases, the `call_` family pushes a value of type `Site` as the implicit first argument: this value contains call site information obtained from `__func__`, `__FILE__`, and `__LINE__`, passed to the `_site` parameter of protocols.

When compiled with `CALL__` expanding to `1`, protocol function is invoked; additionally, if *method* array is given, then the pointer obtained (through array-to-pointer decay) is asserted to be not null, as if by using `notnull_`, and if the assertion works, then second element of the array (protocol at index `1`) is similarly asserted to be not null. Otherwise `CALL__` shall expand to `0`, and proxy function is invoked instead (first element of *method* at index zero).

6.7.1 Named arguments

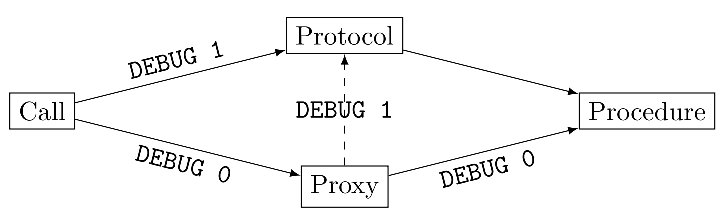
For `(` *prefix* `,` *method-name* `)` style invocation of a method whose prototype permits a fixed number of arguments, a named argument of the form `.`*parameter* `=` *argument* is used to associate an argument with a parameter. The idea is influenced by keyword arguments in the Python programming language, and it can be used to provide arguments out of order, i.e. in a different sequence as compared to the parameter list declared in the prototype. If a named argument is followed by a conventional unnamed argument, the latter corresponds to the subsequent parameter; further unnamed arguments after that are assigned in the declared order of parameters as per the prototype.

If the argument corresponding to a parameter is provided multiple times, only the last expression is considered, and rest of them are not evaluated; if an argument is omitted for some parameter, the default value `0` is used instead (null for pointer type parameters). The latter property can be convenient for augmenting a method with additional parameters, without having to refactor the source code: this is because existing invocations with `call_` will still compile successfully, with each newly added parameter at the end receiving the default argument `0`.

NOTE A crucial point is that existing object files should not be linked with a method extended with additional parameters, but their source codes should be recompiled to generate object files; otherwise the default argument `0` will not be used for the extra parameters, potentially causing undefined behavior at runtime. The `call_` family comes with a mixed blessing that only avoids refactoring the source code if more parameters are added to a method. It still requires recompiling all translation units that invoke the method, based on the updated prototype; failing to do so can silently break existing code. As a general recommendation, an existing method prototype in wide use should never be updated except in very rare circumstances, and it is best to avoid such breaking changes altogether.

6.7.2 Workflow

The following diagram illustrates the workflow of a method invocation using the `call_` family, depending on how the caller and the method are compiled. When `CALL__` expands to `1` at the call site, the `call_` family is configured to invoke the protocol, which in turn invokes the procedure (possibly after validating pre-conditions on the arguments). When `CALL__` expands to `0` at the call site, the `call_` family is configured to invoke the proxy, which is a function pointer declared by the prototype and defined along with the protocol. If the translation unit defining the method (protocol and procedure) is compiled with `METHOD__` expanding to `1`, then the proxy is defined as a pointer to the protocol; otherwise `METHOD__` shall expand to `0`, and the proxy points to a private function (internal linkage) that simply calls the procedure, ignoring the `_site` parameter.



NOTE The dashed line signifies that the proxy is essentially a pointer to the protocol when the method is compiled with `METHOD__` expanding to `1`, so the dashed line does not incur any additional function call overhead.

Effectively, the protocol can be bypassed only when the caller and callee (method) jointly agree that debugging is not necessary, such as when both are compiled with `DEBUG` defined as `0`. This can be done to reduce code size and improve runtime performance when the called method has been tested enough times to instill a reasonable confidence that the code is *possibly* free of bugs. However, it is always important to remember that "*testing shows the presence, not the absence of bugs*" (quote by Edsger Wybe Dijkstra); several classes of bugs cannot be detected using conventional pre-conditions and post-conditions, and certain bugs manifest only with specific compiler flags.

EXAMPLE The following code invokes the method for sorting a list of integers: when compiled with `DEBUG` as `1`, `CALL__` expands to `1` and the protocol is invoked, whose name is given by `verifier_(Integers, Sort)`. With `DEBUG` as `0`, `CALL__` expands to `0`, and the function pointer `proxy_(Integers, sort)` is used for the invocation.

```
#include "Integers._"

Int_ main()
begin
    Int_ arr_[] = {8, 6, 4, 2, 0, 9, 7, 5, 3, 1};
    call_((Integers, sort), puts("not printed"),
    .arr = &arr_, .len = length__(&arr_));
    print_("Sorted array is");
    loop_(0, length__(&arr_) - 1)
        print_(arr_[_i_]);
    end
end
```

NOTE The first argument is later superseded by `.len = length__(&arr_)`; for compiling with `gcc`, the flag `-Wno-override-init-side-effects` may be required to suppress a warning about non-evaluation of `puts` call.

EXERCISE The `private` function `burrow` that implements the recursive merge algorithm uses the signed type `Ptrdiff` for its parameters accepting index values. An earlier version of the code used the `Size` type for parameters, which violated the post-condition in the protocol (when compiled in debugging mode). On further investigation, the resulting array turned out to be `4 0 1 2 3 5 6 7 8 9`, which is clearly not sorted. Stepping through the code then revealed that the array was also being accessed outside its bounds, one position before the base element.

As a small exercise, try changing the parameter types to `Size` and observe the results for the same input array. Find out which part of `burrow` function is affected by this subtle bug caused by unsigned arithmetic wraparound.

6.8 Design strategies

We conclude this chapter with some guidelines for writing protocol and procedure definitions. Capturing the precise behavior in a protocol can be challenging for non-trivial problems, and can often involve more time and effort than writing a procedure for the same task. We also stress upon the fact that it is impossible to write a protocol that can correctly identify all possible bugs: one common example is invalid memory access due to a bad pointer argument.

Protocols should always be sound, but are seldom required to be complete: writing an exhaustive set of conditions is often not desirable, and an excessive level of detail can be counter-productive; besides increasing the development time, extra code also means a greater chance of bugs in the protocol itself, and imposes a burden on maintenance. Protocols should check only the essential aspects; for instance, our sorting protocol is sound but not complete: it only verifies the non-decreasing order, without checking that the resulting array is a permutation of the original.

It is recommended that protocols should be written using an “offensive” design strategy: if a method documentation explicitly disallows some arguments or leaves the behavior undefined, then the protocol should establish the necessary pre-conditions to catch instances of bad invocations. The goal of testing phase is to detect a bug as soon as it occurs, not after it has propagated through several layers of abstraction, making it harder to trace the origin.

Readers may note that in our sorting example, the same checks are also performed by the procedures, but using guard clauses. As opposed to the offensive approach of immediate termination, procedures should adopt a “defensive” strategy and take a reasonable action if some pre-condition is not satisfied. Developers write programs to work, not to fail, and in production mode compilation it is desirable that an application should run as long as possible, taking corrective actions whenever necessary. Except in the case of memory-constrained devices, careful defensive code in procedures can go a long way in improving end-user experience, even in the presence of bugs.

Chapter 7

Classes

A class is used to associate well-defined functionalities with data objects: the behavior is described by protocols and implemented by procedures. A class defines a concrete data type, which means that the names and types of data members are declared along with the class type, and this information can be used to directly modify an attribute. Every class is associated with a set of methods, which provide a controlled mechanism for accessing and updating the state of an instance; however, concrete data types can be modified directly as their member details are known.

Each value of a class type is called an "instance" of that class: it is (pointer to) a structure that contains the member attributes of that class type. Every instance has an implicit attribute that specifies the dynamic type of that object: it is nothing but a pointer to a non-modifiable structure of function pointers, which is same for all instances of the same class, but different across multiple classes. Additionally, the type pointer also creates a static relationship between a class and its base type, establishing a type lineage all the way to the root ancestor `Object`.

Influenced by existing object-oriented languages, instances are typically manipulated via references, guaranteeing a small constant overhead (independent of structure size) when passed as function arguments or copied as return value. Another existing convention we shall follow is to start the name of an object-oriented type with an uppercase letter, and have at least one lowercase letter in the name (to differentiate them from object-like macros usually named entirely in uppercase); this practice is consistent with the notion of classes and interfaces being user defined data types, hence we follow the same naming scheme as for `C_` type synonyms (a trailing underscore means modifiable).

NOTE We shall use the term "object-oriented types" for collectively referring to both classes and interfaces.

7.1 Type structure

`struct Type` is a collection of pointers that contain information about identity, lineage, name, and implementation of a class (concrete type) or interface (abstract type); here "implementation" refers to procedures for a fixed set of protocols. The name `Type_` is a synonym for `Ptr_(const struct Type)`, which means a modifiable pointer to a non-modifiable structure; its counterpart `Type` specifies that the pointer is also non-modifiable, which is same as `Ptr (const struct Type)`. Each class (concrete type) and interface (abstract type) is characterized by a non-modifiable `Type` property associated with it, which is declared as an identifier with external linkage.

7.1.1 self

`self` is a pointer to non-modifiable `struct Type`. `self` is the first member of `Type` structure, and for a given `Type` instance, if `self` points to its own address, it indicates a basic type; otherwise the `Type` instance is a concrete implementation of an extended type (interface), and `self` points to basic `Type` instance of the implementing class.

7.1.2 `base`

Every object-oriented type (class or interface) extends another type called its base type. For a given `Type` instance, the member `base` points to the `Type` structure of the base type which has been extended by the given type.

NOTE The member `base` is used to establish lineage all the way to the `Object` class, the topmost ancestor.

7.1.3 `name`

The member `name` gives a non-modifiable string that stores the class or interface identifier; its type is `Ptr_(Char)`. For extended types, `name` gives the identifier of the implementing class, not the interface that is being implemented.

7.2 `Object` class structure

The `Object` class declares `struct Object` with only one member named `type`. All classes are related to the `Object` class through inheritance: each class has exactly one base class, and `Object` is the root ancestor of all classes.

NOTE The name `Object_` is a synonym for `struct Object`, and `Object` is its non-modifiable counterpart.

7.2.1 `type`

The member `type` is a pointer to a non-modifiable `Type` structure; its data type is `Type_`, which is a synonym for `Ptr_(const struct Type)`. An instance of any class "is an" `Object`, and due to the `type` attribute inherited from `Object` class, every instance "has a" `type`, which points to the `Type` structure of the class being instantiated.

NOTE Instantiating an interface gives a valid instance of `Abstract` type, which itself extends `Object` class.

7.3 Obtaining the type

Syntax

`type_ (` *class-or-interface* `)`

Constraints

class-or-interface shall be the name of an object-oriented type that has been declared prior to its use of `type_`.

Semantics

`type_` gives a pointer to the non-modifiable `Type` structure associated with the object-oriented type declared with the name *class-or-interface*. The outcome is an expression that is not an lvalue.

EXAMPLE `type_(Object)` gives a pointer to the `Type` structure associated with the `Object` class.

7.4 Type inheritance

C_ provides support for structural inheritance, which is implemented by "embedding" an instance of the base type as the first member of its derived type. This well-known design technique works in C due to the following two rules:

- All structure pointers have identical representation, so pointer to an instance of a derived type can be interpreted as pointer to an instance of its base type, without causing loss of information or undefined behavior.
- There is no padding before the first member of a structure: since every class structure embeds the structure of its base class as the first member, pointer to a derived structure is also a valid pointer to the base structure.

Both the rules can be applied recursively, which implies that a valid instance of any class is also structurally valid for any of its ancestor classes (including the `Object` class, whose sole attribute is the member `type`).

7.4.1 Establishing inheritance

In most object-oriented programming languages, the term "inheritance" broadly means that members of a base type are also implicitly members of any of its derived types, which includes both member attributes and member methods. The derived type can suppress an inherited method with its own implementation, which is usually known as "method overriding"; this technique does not forbid accessing the base class method via other mechanisms.

C_ relies on macros to establish inheritance between two object-oriented types: this is not merely limited to the reference implementation, but is part of the specification. The macro and replacement text need to be defined as:

`# define` *Derived*`_EXTENDS` *Base* [, *override-list*]

In the above syntax, a type named as *Derived* inherits from another type named as *Base*. The optional *override-list* can only specify a comma-separated list of method names that are part of the `Type` structure.

NOTE C_ does not specify any mechanism to disallow inheritance, though implementations can support it.

7.4.2 Base array

Declaration of an object-oriented type named *Derived* requires a prior macro named *Derived*`_EXTENDS`, and the first element in its replacement text is considered to be the name of the base type. The first member in the structure of the derived type is a singleton array named `base`, whose element type is same as an instance of the base type.

NOTE The instance attribute `base` is not to be confused with the pointer member `base` of `Type` structure.

7.4.3 Type validation

The function `is_type` performs some basic validation checks on (pointer to) a `Type` structure.

Declaration

```
Bool_ is_type(Type);
```

Description

The function `is_type` returns `true` if and only if the following conditions are satisfied:

- The `Type` argument is not a null pointer.
- On dereferencing the pointer, each member of the `Type` structure is not null.
- The previous rules are recursively satisfied for each ancestor class, reachable via the array member `base`.
- The root ancestor `Object` class is reached within a maximum height of `PP_MAX`.

If any of the above conditions fails, then it is not considered to be a valid `Type`, and the return value is `false`.

7.4.4 Liskov substitution

C_ supports Liskov substitution, allowing a derived class object to be used wherever its base class object is expected: this rule works recursively, so the instance of any class can be treated as an instance of each of its ancestor classes, up to the root ancestor `Object` class. The common practice is to operate on instances via pointers, and since each class structure implicitly contains its base structure as the first member, the pointer representation can be directly interpreted as pointing to an instance of the base structure, or recursively to an instance of any ancestor class.

The function `is_type` only checks whether a pointer refers to a valid `Type` structure or not. To check whether an object can be considered as a valid instance of a given class, the required conditions are tested by the `validate` method provided by that class (or inherited from its base class). However, for a C_ program to work correctly with Liskov substitution, it is necessary that each valid instance of a derived class is also a valid instance of all of its ancestor classes; this is done by the `validate` function, whose precise behavior is described in a later section.

7.5 Type relationships

Consider the following two class hierarchies:

- A class `Person` has several derived classes, one of which is called `Programmer` class.
- `Phone` class is inherited by `Landline` and `Mobile`; the latter is sub-classed by `FeaturePhone` and `SmartPhone`.

We shall use these examples to illustrate type relationships in the following subsections.

7.5.1 Sub-type checking

7.5.1.1 Prototype

```
prototype_(Bool_, (Type, is), (Type, descendant), (Type, ancestor))
```

7.5.1.2 Pre-conditions

- The call `is_type(descendant)` must return `true`.
- The call `is_type(ancestor)` must return `true`.

7.5.1.3 Procedure

If `ancestor->self` can be reached from `descendant->self` by traversing through `base` pointer of the latter (until `Object` type is reached), then the return value is `true`; otherwise the return value is `false`.

7.5.1.4 Invocation

The `is` family is used to check for "is-a" relationship among object-oriented types.

Syntax

`is` ([`. descendant =`] *descendant* `,` [`. ancestor =`] *ancestor*)
`is_` (*oo-identifier-list*)
`is__` (*oo-type-list*)

Constraints

Both *descendant* and *ancestor* shall be expressions of type `Type`, optionally specifying parameter names.
oo-identifier-list shall be a comma-separated list of names for object-oriented types whose declarations are visible.
oo-type-list shall be a comma-separated list of expressions, each of which shall be of type `Type`.
The number of elements in *oo-identifier-list* or *oo-type-list* shall be less than `PP_MAX` after macro expansions.

Semantics

`is (`*descendant* `,` *ancestor*`)` is equivalent to the more verbose `call_((Type, is) ,` *descendant* `,` *ancestor*`)`.
`is_` accepts a list of object-oriented type names, and checks if each type is a descendant of the type after it.
`is__` accepts a list of object-oriented type expressions, checking if each type is a descendant of the type after it.
Both `is` and `is_` evaluate each argument exactly once; for `is__`, if the number of expanded arguments in *oo-type-list* is more than two, then all arguments except the first and last can be evaluated more than once.
In each case, the outcome in an expression of type `Bool_`, and it is not an lvalue.

NOTE "is-a" relationship establishes a partial ordering, being reflexive, anti-symmetric, and transitive.

EXAMPLE `is(type_(Person), type_(Object))` is `true`, but `is(type_(Object), type_(Person))` is `false`.
`is_(FeaturePhone, Mobile, Phone, Object)` is `true`, which is equivalent to the following conjunction:

```
is_(FeaturePhone, Mobile) && is_(Mobile, Phone) && is_(Phone, Object)
```

7.5.2 Nearest common ancestor

C_ uses the term "super" in a slightly different sense as compared to most other object-oriented programming languages. Super type refers to the nearest common ancestor of two (or more) object-oriented types; intuitively speaking, super type represents the largest or most precise intersection between multiple object-oriented types.

7.5.2.1 Prototype

`prototype_(Type_, (Type, super), (Type, this), (Type, that))`

7.5.2.2 Pre-conditions

- The call `is_type(this)` must return `true`.
- The call `is_type(that)` must return `true`.

7.5.2.3 Procedure

If the return value is *type*, then both the invocations `is(this,` *type*`)` and `is(that,` *type*`)` return `true`. Additionally, if *des* is any descendant of *type*, then at least one of `is(this,` *des*`)` or `is(that,` *des*`)` is `false`.

NOTE If the arguments `this` and `that` compare as equal pointers, then it is also the return value.

7.5.2.4 Invocation

Syntax

`super` ([`. this =`] *this* , [`. that =`] *that*)
`super_` (*oo-identifier-list*)

Constraints

Both the arguments *this* and *that* shall be expressions of type `Type`, optionally specifying parameter names.
oo-identifier-list shall be a comma-separated list of names for object-oriented types whose declarations are visible.
The number of elements in *oo-identifier-list* shall be less than `PP_MAX` after macro expansions.

Semantics

`super (`*this* , *that*`)` is equivalent to the more verbose invocation `call_((Type, super) ,` *this* , *that*`)`.

`super_` accepts a list of object-oriented type names, and returns their nearest common ancestor type; it essentially provides a combination of map and fold/reduce, by applying `type_` on each name in *oo-identifier-list*, and then using fold/reduce with `super` as the aggregator. If *oo-identifier-list* is a singleton, the outcome is `type_(`*oo-identifier-list*`)`.

NOTE `super` is commutative, so the order of arguments does not affect the return value; however, the order of evaluation of arguments is unspecified, so side-effects should generally be avoided in argument expressions.

EXAMPLE The outcome of `super (type_(Programmer), type_(Phone))` is same as `type_(Object)`.
The outcome of `super_(Landline, FeaturePhone, SmartPhone)` is same as `type_(Phone)`.

7.5.3 Type composition

If an instance of some type A contains a member that is (pointer to) an instance of another type B, then A is said to be related to B through composition; in other words, every instance of type A "has-an" instance of type B.

For example, if the class structure of `Person` has a member that is pointer to an instance of class `Phone`, then we can say that "`Person` 'has-a' `Phone`" (the sentence "`Person` 'is-a' `Phone`" would incorrectly suggest inheritance).

NOTE C predates the advent of object-oriented design, and C_ uses composition for structural inheritance.

7.6 Type methods

During instantiation, the `type` attribute inherited from `Object` class is initialized to point to the `Type` structure associated with the class or interface being instantiated. The following subsections describe those `Type` methods for which a corresponding function pointer is declared in the `Type` structure, whose value is used for callback.

Except for `validate`, each `Type` method defines a pair of functions, protocol and procedure: protocol describes the expected behavior with pre-conditions and post-conditions, whereas its associated procedure invokes a callback function given by the corresponding member of a `Type` parameter, or from the `type` attribute of an `Object` pointer.

Except for `comparable`, each subsection lists three declarations. Prototype declarations uses generic `Void/Void_` pointers for parameters and return type, which is done to avoid boilerplate type casts during invocations. Procedure associated with a prototype invokes a callback function obtained from a type structure. The callback pointer need not have the same function type as the procedure, and in all cases, their parameter and return types are different: prototypes use `Void/Void_` pointers, whereas function pointers in the `Type` structure use `Object/Object_` pointers.

When a new object-oriented type is created, one procedure is declared for each function pointer in the type structure. If the new type (class or interface) is named as T, then its associated procedures use $T/T\_$ pointers instead of `Object/Object_` pointers for parameters and return type. This is done to allow precise type checking of arguments and return value when these procedures are directly invoked; if they are used as callback via function pointers in type structure, the arguments and return value are expected to be `Object/Object_` pointers, which works correctly because both T and `Object` identify structure types, and their pointers have identical representation.

7.6.1 `validate`

`validate` is declared as a function instead of a method, as it does not specify any pre-conditions or post-conditions.

7.6.1.1 Declarations

Declaration
`Bool_ validate(Type type, Void *this);`
Type member
`Bool_ (*validate)(Object *);`
For type T
`Bool_ solver_(`T`, validate)(`T` *);`

7.6.1.2 Description

`validate` returns `true` if and only if all of the following conditions are satisfied:

- `this` is not null, and `is_type(((Object *) this)->type)` is `true`.
- If `type` is not null, then `is_type(type)` and `is(((Object *) this)->type, type)` are `true`; in other words, `this` should point to an instance whose `type` attribute is a descendant or sub-type of the `type` parameter.
- The `validate` procedure is invoked for each `Type` in the lineage from `Object` class to the `type` attribute of `this` instance, *i.e.* starting from the `Object` class and ending with `((Object *) this)->type)`; if any of them returns `false`, then that is the outcome, and successive calls are not performed. Also, if a derived type inherits `validate` procedure from its base type without overriding it, then that procedure is called only once.

NOTE The order of invoking `validate` procedures is intended to be aligned with Liskov substitution principle: every valid instance of a derived type is required to be a valid instance for its base type, and this applies transitively.

7.6.1.3 Invocation

Syntax

`validate_` ([*type*=`NULL` ,] *this*)
`validate_1_` (*this*)
`validate_2_` (*type* , *this*)

Semantics

`validate_` invokes `validate_`n`_` if the expanded argument sequence contains n arguments.
`validate_1_` calls `validate` with `type` as null pointer; `validate_2_` is a trivial wrapper over `validate`.

7.6.2 `init`

7.6.2.1 Declarations

Prototype

`prototype_(Void_ *, (Type, init), (Type, type), (Tape, tape))`

Type member

`Object_ *(*init)(Type, Tape);`

For type T

T`_ *solver_(`T`, init)(Type, Tape);`

7.6.2.2 Pre-conditions

- `is_type(type)` must be `true`.
- `tape` must not be null.

NOTE `tape` is expected to be pointer to an array whose last element is null pointer (similar to `argv` in `main`).

7.6.2.3 Post-conditions

- If the procedure returns *instance* and it is not null, then `validate(type,` *instance*`)` must be `true`.

7.6.2.4 Procedure

- If `is_type(type)` is `false`, then a null pointer is returned.
- If `tape` is null, it is adjusted to a singleton array whose only member is a null pointer.
- The return value is same as that of `type->init` when it is called with `type` and `tape` (adjusted) as arguments.

7.6.2.5 Invocation

Syntax

`init_` (*oo-type-name*)
`init_1_` (*oo-type-name*)
`init_` ([`. type =`] *type* , [`. tape =`] *tape*)
`init_2_` ([`. type =`] *type* , [`. tape =`] *tape*)

`init__` ([`. type =`] *type* , *argument-list*)
`init__0__` ([`. type =`] *type* , *argument-list*)
`init__` (*oo-type-name*)
`init__1_` (*oo-type-name*)

Semantics

`init_` invokes `init_`*n*`_` if the expanded argument sequence contains *n* arguments. `init_1_(`*oo-type-name*`)` is equivalent to the verbose call `((`*oo-type-name*`_ *) call_((Type, init), type_(`*oo-type-name*`), (Tape){NULL}))`.

`init_2_(`*type*`, `*tape*`)` is equivalent to `call_((Type, init), `*type*`, `*tape*`)`; the outcome is of type `Void_ *`.

`init__` invokes `init__1_` if the expanded argument sequence is a singleton; otherwise it invokes `init__0__`.

`init__1_` is equivalent to `init_1_`.

`init__0__(`*type*`, `*argument-list*`)` is same as `init_2_(`*type*`, (Tape){map__(&lvalue__, `*argument-list*`), NULL})`. In other words, each value in *argument-list* is converted into an lvalue (as if with `lvalue__`), pointers to these lvalues are packed into an array, and a terminating null pointer is appended to it (to mark the end); each lvalue and the array itself have automatic storage duration, whose lifetime ends after the innermost block where the invocation is performed. Each value in *argument-list* can be evaluated more than once only if it has a variably modified type.

7.6.3 `free`

7.6.3.1 Declarations

Prototype

`prototype_((Type, free), (Void_ *, this))`

Type member

`Void_ (*free)(Object_ *);`

For type *T*

`Void_ solver_(`*T*`, free)(`*T*`_ *);`

7.6.3.2 Pre-conditions

- If `this` is not null, then `is_type(((Object *) this)->type)` must be `true`.

7.6.3.3 Procedure

- If `this` is a null pointer, then the function returns immediately.
- Let *type* be the value of `((Object *) this)->type`; if `is_type(`*type*`)` is `false`, then return immediately.
- Otherwise `is_type(`*type*`)` is `true` and *type*`->free(this)` is invoked.

7.6.3.4 Invocation

Syntax

`free_ (` [`. this =`] *this* `)`

Semantics

`free_(`*this*`)` is equivalent to the more verbose invocation `call_((Type, free), `*this*`)`.

7.6.4 `compare`

7.6.4.1 Declarations

Prototype

`prototype_(LLong_, (Type, compare), (Void *, this), (Void *, that))`

Type member

`LLong_ (*compare)(Object *, Object *);`

For type *T*

`LLong_ solver_(`*T*`, compare)(`*T* `*, `*T* `*);`

7.6.4.2 Pre-conditions

- `validate_(this)` must be `true`.
- `validate_(that)` must be `true`.

7.6.4.3 Procedure

- If `this` and `that` compare as equal pointers, then the function returns zero.
- If `validate_(this)` or `validate_(that)` is `false`, then the function returns one.
- Otherwise, let *super* denote the value of `super(((Object *) this)->type, ((Object *) that)->type)`. Then the return value is same as the result of *super*`->compare(this, that)`.

7.6.4.4 Invocation

Syntax

`compare_ (` [`. this =`] *this* `,` [`. that =`] *that* `)`

Semantics

`compare_(`*this*`,` *that*`)` is equivalent to the more verbose invocation `call_((Type, compare),` *this*`,` *that*`)`.

7.6.4.5 Recommended practice

The result of comparison between two distinct instances should follow the same convention as prescribed for the comparator function required by standard library functions `bsearch` and `qsort` (both are declared in <stdlib.h>).

- If the first instance is considered to be less than the second instance, then the outcome should be negative.
- If both instances are considered to be equal (though not necessarily same), then the outcome should be zero.
- If the first instance is considered to be more than the second instance, then the outcome should be positive.
- Otherwise both the instances are considered to be incomparable, and the outcome can be any positive value.

If a derived type overrides `compare` procedure of its base type, it should be a refinement in the following sense:

- If two instances can be compared with the base type, they should remain comparable for the derived type.
- If two instances are considered unequal by the base type, they should remain unequal for the derived type.

7.6.5 `comparable`

7.6.5.1 Declarations

Prototype

```
prototype_(Bool_, (Type, comparable), (Void *, this), (Void *, that))
```

7.6.5.2 Pre-conditions

- `validate_(this)` must be `true`.
- `validate_(that)` must be `true`.

7.6.5.3 Procedure

- If `this` and `that` compare as equal pointers, then the function returns `true`.
- If `validate_(this)` or `validate_(that)` is `false`, then the function returns `false`.
- Otherwise, let *super* denote the value of `super(((Object *) this)->type, ((Object *) that)->type)`. If the result of *super*`->compare(this, that)` is zero, then the function returns `true`. Otherwise, the function returns `true` iff *super*`->compare(this, that)` and *super*`->compare(that, this)` have opposite signs.

7.6.5.4 Invocation

Syntax
`comparable_` ([`. this =`] *this* , [`. that =`] *that*)
Semantics
`comparable_(`*this*`, `*that*`)` is equivalent to the verbose invocation `call_((Type, comparable), `*this*`, `*that*`)`.

7.6.6 `copy`

7.6.6.1 Declarations

Prototype
```
prototype_(Void_ *, (Type, copy), (Void_ *, this), (Void *, that))
```
Type member
```
Object_ *(*copy)(Object_ *, Object *);
```
For type T
T`_ *solver_(`T`, copy)(`T`_ *, `T` *);`

7.6.6.2 Pre-conditions

- `this` and `that` must not compare as equal pointers.
- `validate_(that)` must be `true`.
- If `this` is not a null pointer, then `is_type(((Object *) this)->type)` must be `true`.
- If `this` is not null pointer, then `is(((Object *) this)->type, ((Object *) that)->type)` must be `true`. In other words, a non-null `this` must be an instance of a sub-type of `that->type` (if not the same type).

NOTE The first pre-condition is meant to catch bugs where both argument expressions evaluate to same value.

7.6.6.3 Post-conditions

Let *copy* be the pointer returned on invoking the procedure. If *copy* is not null, then the following must be satisfied:
- `validate(((Object *) that)->type, `*copy*`)` must be `true`.
- `solver_(Type, compare)(`*copy*`, that)` must be zero; in other words, *copy* and `that` must compare equal.

7.6.6.4 Procedure

- If `this` and `that` compare as equal pointers, then the function returns `this`.
- If `validate_(that)` is `false`, then the function returns a null pointer.
- If `this` is not a null pointer and `is_type(((Object *) this)->type)` is `false`, then `NULL` is returned.
- If `this` is not null and `is(((Object *)this)->type, ((Object *)that)->type)` is `false`, `NULL` is returned.
- Let *type* denote the value of `((Object *) that)->type`; then the function returns *type*`->copy(this, that)`.

7.6.6.5 Invocation

Syntax
`copy_` ([[`. this =`] *this* ,] [`. that =`] *that*)
`copy_1_` ([`. that =`] *that*)
`copy_2_` ([`. this =`] *this* , [`. that =`] *that*)
Semantics
`copy_` invokes `copy_`n`_` if the expanded argument sequence contains n arguments.
`copy_1_(`*that*`)` is equivalent to `copy_2_(NULL, `*that*`)`.
`copy_2_(`*this*`, `*that*`)` is equivalent to `( (typeof (unqual__(`*that*`))) call_((Type, copy), `*this*`, `*that*` )`.

7.6.7 `read`

7.6.7.1 Declarations

Prototype
`prototype_(Void_ *, (Type, read), (Void_ *, this), (Stream, in))`
Type member
`Object_ *(*read)(Object_ *, Stream);`
For type T
T`_ *solver_(`T`, read)(`T`_ *, Stream);`

7.6.7.2 Pre-conditions

- `this` must not be a null pointer.
- `in` must not be a null pointer.
- `is_type(((Object *) this)->type)` must be `true`.

7.6.7.3 Post-conditions

Let *type* be the value of `((Object *)this)->type` before calling the procedure and *res* be the value returned by it

- If *res* is not a null pointer, then `validate(`*type*`, `*res*`)` must be `true`.

7.6.7.4 Procedure

- If `this` or `in` is a null pointer, then the function returns a null pointer.
- Let *type* be the value of `((Object *) this)->type`; if `is_type(`*type*`)` is `false`, the function returns `NULL`.
- Otherwise the function returns the value of *type*`->read(this, in)`.

7.6.7.5 Invocation

Syntax
`read_` ([`. this =`] *this* [, [`. in =`] *in*])
`read_1_` ([`. this =`] *this*)
`read_2_` ([`. this =`] *this* , [`. in =`] *in*)
Semantics
`read_` invokes `read_`n`_` if the expanded argument sequence contains n arguments.
`read_1_(`*this*`)` is equivalent to `read_2_(`*this*`, stdin)`.
`read_2_(`*this*`, `*in*`)` is equivalent to `( (typeof_(`*this*`)) call_((Type, read), `*this*`, `*in*`) )`.

7.6.8 `write`

7.6.8.1 Declarations

Prototype
`prototype_(LLong_, (Type, write), (Void *, this), (Stream, out))`
Type member
`LLong_ (*write)(Object *, Stream);`
For type T
`LLong_ solver_(`T`, write)(`T` *, Stream);`

7.6.8.2 Pre-conditions

- `validate_(this)` must be `true`.
- `out` must not be a null pointer.

7.6.8.3 Procedure

- If `validate_(this)` is `false` or `out` is a null pointer, then the function returns `-1`.
- Otherwise, let *type* be the value of `((Object *) this)->type`; the return value is *type*`->write(`*this*`, `*out*`)`.

7.6.8.4 Invocation

Syntax

`write_` ([`. this =`] *this* [, [`. out =`] *out*])
`write_1_` ([`. this =`] *this*)
`write_2_` ([`. this =`] *this* , [`. out =`] *out*)

Semantics

`write_` invokes `write_`n`_` if the expanded argument sequence contains n arguments. `write_1_(`*this*`)` is equivalent to `write_2_(`*this*`, stdout)`. `write_2_(`*this*`, `*out*`)` is equivalent to `call_((Type, write), `*this*`, `*out*`)`.

7.6.9 parse

7.6.9.1 Declarations

Prototype

`prototype_(Void_ *, (Type, parse), (Void_ *, this), (Size_ *, length), (Void *, in))`

Type member

`Object_ *(*parse)(Object_ *, Size_ *, Void *);`

For type T

T`_ *solver_(`T`, parse)(`T`_ *, Size_ *, Void *);`

7.6.9.2 Pre-conditions

- `this` must not be a null pointer.
- `in` must not be a null pointer.
- `is_type(((Object *) this)->type)` must be `true`.
- `length` must not be a null pointer.
- `*length` must not be equal to zero.

7.6.9.3 Post-conditions

Let *type* be the value of `((Object *)this)->type` and *capacity* be the value of `*length` before invoking the procedure. Let *instance* be the pointer returned by the procedure.

- After the procedure returns, the updated value of `*length` must not exceed *capacity*.
- If *instance* is not a null pointer, then `validate(`*type*`, `*instance*`)` must be `true`.

7.6.9.4 Procedure

- If `this` or `in` is a null pointer, then the function returns a null pointer.
- Let *type* be the value of `((Object *) this)->type`; if `is_type(`*type*`)` is `false`, the function returns `NULL`.
- If `length` is a null pointer or `*length` is equal to zero, then the function returns a null pointer.
- Otherwise, the function returns *type*`->parse(this, length, in)`.

7.6.9.5 Invocation

Syntax

parse__ ([. this =] *this* [, [. length =] *length*] , [. in =] *in*)
parse__2__ ([. this =] *this* , [. in =] *in*)
parse__3_ ([. this =] *this* , [. length =] *length* , [. in =] *in*)

parse_ ([. this =] *this* [, [. length =] *length*] , [. in =] *in*)
parse_2_ ([. this =] *this* , [. in =] *in*)
parse_3_ ([. this =] *this* , [. length =] *length* , [. in =] *in*)

Semantics

`parse__` invokes `parse__2__` if the argument sequence expands to two elements.
Otherwise the argument sequence shall expand to three elements, and `parse__3_` is invoked.
`parse__2__(`*this*`,` *in*`)` is equivalent to `parse__3_(`*this*`, &lvalue__(length__(`*in*`)),` *in*`)`.
`parse__2__` can evaluate the argument *in* more than once only if it has a variably modified type.
`parse__3_(`*this*`,` *length*`,` *in*`)` is equivalent to `( (typeof_(`*this*`)) call_((Type, parse),` *this*`,` *length*`,` *in*`) )`.

The `parse_*` family evaluates each argument exactly once; rest of the semantics are identical to `parse__` family. More precisely, `parse_2_*` has the same functionality as `parse__2__`, and `parse_3_` is equivalent to `parse__3_`.

7.6.10 `text`

7.6.10.1 Declarations

Prototype

`prototype_(Void_ *, (Type, text), (Void *, this), (Size_ *, length), (Void_ *, out))`

Type member

`Void_ *(*text)(Object *, Size_ *, Void_ *);`

For type *T*

`Void_ *solver_(`*T*`, text)(`*T*` *, Size_ *, Void_ *);`

7.6.10.2 Pre-conditions

- `validate_(this)` shall return `true`.
- `length` must not be a null pointer.
- `*length` must not be equal to zero.

7.6.10.3 Post-conditions

Let *capacity* be the value of `*length` before invoking the procedure, and let *text* be the pointer returned by it.

- The updated value of `*length` must not be equal to zero.
- If *text* and *out* are not null, and updated `*length` does not exceed *capacity*, then *text* and *out* must be equal.

7.6.10.4 Procedure

- If `validate_(this)` is `false`, then the function returns a null pointer.
- If `length` is a null pointer or `*length` is equal to zero, then the function returns a null pointer.
- Let *type* be the value of `((Object *)this)->type`; then the function returns *type*`->text(this, length, out)`.

7.6.10.5 Invocation

Syntax

text__ ([. this =] *this* [[, [. length =] *length*] , [. out =] *out*])
text__1_ ([. this =] *this*)
text__2__ ([. this =] *this* , [. out =] *out*)
text__3_ ([. this =] *this* , [. length =] *length* , [. out =] *out*)

text_ ([. this =] *this* [[, [. length =] *length*] , [. out =] *out*])
text_1_ ([. this =] *this*)
text_2_ ([. this =] *this* , [. out =] *out*)
text_3_ ([. this =] *this* , [. length =] *length* , [. out =] *out*)

Semantics

`text__` invokes `text__1_` if the argument sequence expands to a singleton. If the argument sequence expands to two elements, then `text__2__` is invoked; otherwise there shall be three arguments, and `text__3_` is invoked.

`text__1_` (*this*) is equivalent to `text__3_(`*this*`, &(Size_){1}, NULL)`.

`text__2__(`*this*`, `*out*`)` is equivalent to `text__3_(`*this*`, &lvalue__(length__(`*out*`)), `*out*`)`.

`text__2__` can evaluate the argument *out* more than once only if it has a variably modified type.

`text__3_(`*this*`, `*length*`, `*out*`)` is equivalent to the verbose invocation `call_((Type, text), `*this*`, `*length*`, `*out*`)`.

The `text_`* family evaluates each argument exactly once; rest of the semantics are identical to `text__` family. So `text_1_` is equivalent to `text__1_`, `text_2_`* is similar to `text__2__`, and `text_3_` is equivalent to `text__3_`.

7.6.11 decode

7.6.11.1 Declarations

Prototype

`prototype_(Void_ *, (Type, decode), (Void_ *, this), (Size_ *, length), (Encoding *, in))`

Type member

`Object_ *(*decode)(Object_ *, Size_ *, Encoding *);`

For type *T*

T`_ *solver_(`*T*`, decode)(`*T*`_ *, Size_ *, Encoding *);`

7.6.11.2 Pre-conditions

- `this` must not be a null pointer.
- `in` must not be a null pointer.
- `is_type(((Object *) this)->type)` must be `true`.
- `length` must not be a null pointer.
- `*length` must not be equal to zero.

7.6.11.3 Post-conditions

Let *type* be the value of `((Object *)this)->type` and *capacity* be the value of `*length` before invoking the procedure. Let *instance* be the pointer returned by the procedure.

- After the procedure returns, the updated value of `*length` must not exceed *capacity*.
- If *instance* is not a null pointer, then `validate(`*type*`, `*instance*`)` must be `true`.

7.6.11.4 Procedure

- If `this` or `in` is a null pointer, then the function returns a null pointer.
- Let *type* be the value of `((Object *) this)->type`; if `is_type(`*type*`)` is `false`, the function returns `NULL`.
- If `length` is a null pointer or `*length` is equal to zero, then the function returns a null pointer.
- Otherwise, the function returns *type*`->decode(this, length, in)`.

7.6.11.5 Invocation

Syntax

```
decode__   ( [. this =] this [, [. length =] length] , [. in =] in )
decode__2__ ( [. this =] this  , [. in =] in )
decode__3_  ( [. this =] this  , [. length =] length  , [. in =] in )

decode_    ( [. this =] this [, [. length =] length] , [. in =] in )
decode_2_  ( [. this =] this  , [. in =] in )
decode_3_  ( [. this =] this  , [. length =] length  , [. in =] in )
```

Semantics

`decode__` invokes `decode__2__` if the argument sequence expands to two elements.
Otherwise the argument sequence shall expand to three elements, and `decode__3_` is invoked.
`decode__2__(`*this*`, `*in*`)` is equivalent to `decode__3_(`*this*`, &lvalue__(length__(`*in*`)), `*in*`)`.
`decode__2__` can evaluate the argument *in* more than once only if it has a variably modified type.
`decode__3_(`*this*`, `*length*`, `*in*`)` is equivalent to `((typeof_(`*this*`)) call_((Type, decode), `*this*`, `*length*`, `*in*`))`.

The `decode_`* family evaluates each argument only once; rest of the semantics are identical to `decode__` family. Specifically, `decode_2_`* has the same functionality as `decode__2__`, and `decode_3_` is equivalent to `decode__3_`.

7.6.12 encode

7.6.12.1 Declarations

Prototype

```
prototype_(Encoding_*, (Type, encode), (Void *, this), (Size_*, length), (Encoding_*, out))
```

Type member

```
Encoding_ *(*encode)(Object *, Size_ *, Encoding_ *);
```

For type T

```
Encoding_ *solver_(T, encode)(T *, Size_ *, Encoding_ *);
```

7.6.12.2 Pre-conditions

- `validate_(this)` shall return `true`.
- `length` must not be a null pointer.
- `*length` must not be equal to zero.

7.6.12.3 Post-conditions

Let *capacity* be the value of `*length` before invoking the procedure, and let *enc* be the pointer returned by it.

- The updated value of `*length` must not be equal to zero.
- If *enc* and *out* are not null, and updated `*length` does not exceed *capacity*, then *enc* and *out* must be equal.

7.6.12.4 Procedure

- If `validate_(this)` is `false`, then the function returns a null pointer.
- If `length` is a null pointer or `*length` is equal to zero, then the function returns a null pointer.
- Let *type* be the value of `((Object *)this)->type`; the function returns *type*`->encode(this, length, out)`.

7.6.12.5 Invocation

Syntax

`encode__` ([`. this =`] *this* [[, [`. length =`] *length*] , [`. out =`] *out*])
`encode__1_` ([`. this =`] *this*)
`encode__2__` ([`. this =`] *this* , [`. out =`] *out*)
`encode__3_` ([`. this =`] *this* , [`. length =`] *length* , [`. out =`] *out*)

`encode_` ([`. this =`] *this* [[, [`. length =`] *length*] , [`. out =`] *out*])
`encode_1_` ([`. this =`] *this*)
`encode_2_` ([`. this =`] *this* , [`. out =`] *out*)
`encode_3_` ([`. this =`] *this* , [`. length =`] *length* , [`. out =`] *out*)

Semantics

`encode__` invokes `encode__1_` if the argument sequence expands to a singleton. If it expands to two arguments, then `encode__2__` is invoked; otherwise there shall be three arguments, and `encode__3_` is invoked.

`encode__1_ (`*this*`)` is equivalent to `encode__3_(`*this*`, &(Size_){1}, NULL)`.

`encode__2__(`*this*`, `*out*`)` is equivalent to `encode__3_(`*this*`, &lvalue__(length__(`*out*`)), `*out*`)`.

`encode__2__` can evaluate the argument *out* more than once only if it has a variably modified type.

`encode__3_(`*this*`, `*length*`, `*out*`)` is equivalent to the invocation `call_((Type, encode), `*this*`, `*length*`, `*out*`)`.

`encode_`* family evaluates each argument only once; other semantics are same as `encode__` family. `encode_1_` is equivalent to `encode__1_`, `encode_2_`* is similar to `encode__2__`, and `encode_3_` is equivalent to `encode__3_`.

7.6.12.6 Recommended practice

If a valid encoding can be generated for an instance, then it should be possible to reconstruct that instance by invoking the `decode` method with the generated encoding. Unlike the `text` method for generating human-readable textual representation of an instance, an encoding is typically binary data and it should characterize an instance. A single instance can have multiple encodings, but two distinguishable instances should have different encodings.

7.6.13 `add`

7.6.13.1 Declarations

Prototype

`prototype_(Void_ *, (Type, add), (Void_ *, sum), (Void *, augend), (Void *, addend))`

Type member

`Object_ *(*add)(Object_ *, Object *, Object *);`

For type *T*

T`_ *solver_(`*T*`, add)(`*T*`_ *, `*T*` *, `*T*` *);`

7.6.13.2 Pre-conditions

- `validate_(augend)` must be `true`.
- `validate_(addend)` must be `true`.

Let *super* denote the value of `super(((Object *) augend)->type, ((Object *) addend)->type)`. If `sum` is not a null pointer, let *type* denote the value of `((Object *) sum)->type`.

- `is_type(`*type*`)` must be `true`.
- `is(`*type*`, `*super*`)` must be `true`.

7.6.13.3 Post-conditions

- Let *object* be returned by procedure; if *object* is not null, then `validate(`*super*`, `*object*`)` must be `true`.

7.6.13.4 Procedure

- If `validate_(augend)` is `false`, then the function returns a null pointer.
- If `validate_(addend)` is `false`, then the function returns a null pointer.

Let *super* denote the value of `super(((Object *) augend)->type, ((Object *) addend)->type)`. If `sum` is not a null pointer, let *type* denote the value of `((Object *) sum)->type`.

- If `is_type(`*type*`)` is `false`, then the function returns a null pointer.
- If `is(`*type*`, `*super*`)` is `false`, then the function returns a null pointer.
- Otherwise the function returns the value of *super*`->add(sum, augend, addend)`.

7.6.13.5 Invocation

Syntax

`add_` ([[`. sum =`] *sum* ,] [`. augend =`] *augend* , [`. addend =`] *addend*)
`add_2_` ([`. augend =`] *augend* , [`. addend =`] *addend*)
`add_3_` ([`. sum =`] *sum* , [`. augend =`] *augend* , [`. addend =`] *addend*)

Semantics

`add_` invokes `add_`n`_` if the expanded argument sequence contains n arguments.
`add_2_(`*augend*`, `*addend*`)` is equivalent to `add_3_(NULLPTR, `*augend*`, `*addend*`)`.
`add_3_(`*sum*`, `*augend*`, `*addend*`)` is equivalent to the following verbose invocation:

`( (typeof_(`*sum*`)) call_((Type, add), `*sum*`, `*augend*`, `*addend*`) )`

7.6.14 sub

7.6.14.1 Declarations

Prototype
`prototype_(Void_*, (Type, sub), (Void_*, difference), (Void*, minuend), (Void*, subtrahend))`
Type member
`Object_ *(*sub)(Object_ *, Object *, Object *);`
For type *T*
T`_ *solver_(`*T*`, sub)(`*T*`_ *, `*T*` *, `*T*` *);`

7.6.14.2 Pre-conditions

- `validate_(minuend)` must be `true`.
- `validate_(subtrahend)` must be `true`.

Let *super* denote the value of `super(((Object *) minuend)->type, ((Object *) subtrahend)->type)`.
If `difference` is not a null pointer, let *type* denote the value of `((Object *) difference)->type`.

- `is_type(`*type*`)` must be `true`.
- `is(`*type*`, `*super*`)` must be `true`.

7.6.14.3 Post-conditions

- Let *object* be returned by procedure; if *object* is not null, then `validate(`*super*`, `*object*`)` must be `true`.

7.6.14.4 Procedure

- If `validate_(minuend)` is `false`, then the function returns a null pointer.
- If `validate_(subtrahend)` is `false`, then the function returns a null pointer.

Let *super* denote the value of `super(((Object *) minuend)->type, ((Object *) subtrahend)->type)`.
If `difference` is not a null pointer, let *type* denote the value of `((Object *) difference)->type`.

- If `is_type(`*type*`)` is `false`, then the function returns a null pointer.
- If `is(`*type*`, `*super*`)` is `false`, then the function returns a null pointer.
- Otherwise the function returns the value of *super*`->sub(difference, minuend, subtrahend)`.

7.6.14.5 Invocation

Syntax
`sub_` ([[`. difference =`] *difference* ,] [`. minuend =`] *minuend* , [`. subtrahend =`] *subtrahend*)
`sub_2_` ([`. augend =`] *minuend* , [`. subtrahend =`] *subtrahend*)
`sub_3_` ([`. difference =`] *difference* , [`. minuend =`] *minuend* , [`. subtrahend =`] *subtrahend*)
Semantics
`sub_` invokes `sub_`*n*`_` if the expanded argument sequence contains *n* arguments.
`sub_2_(`*minuend*`, `*subtrahend*`)` is equivalent to `sub_3_(NULLPTR, `*minuend*`, `*subtrahend*`)`.
`sub_3_(`*difference*`, `*minuend*`, `*subtrahend*`)` is equivalent to the following verbose invocation:

`( (typeof_(`*difference*`)) call_((Type, sub), `*difference*`, `*minuend*`, `*subtrahend*`) )`

7.6.15 `mul`

7.6.15.1 Declarations

Prototype
`prototype_(Void_*, (Type,mul), (Void_*,product), (Void*,multiplier), (Void*,multiplicand))`
Type member
`Object_ *(*mul)(Object_ *, Object *, Object *);`
For type T
T`_ *solver_(`T`, mul)(`T`_ *, `T` *, `T` *);`

7.6.15.2 Pre-conditions

- `validate_(multiplier)` must be `true`.
- `validate_(multiplicand)` must be `true`.

Let *super* denote the value of `super(((Object *) multiplier)->type, ((Object *) multiplicand)->type)`. If `product` is not a null pointer, let *type* denote the value of `((Object *) product)->type`.

- `is_type(`*type*`)` must be `true`.
- `is(`*type*`, `*super*`)` must be `true`.

7.6.15.3 Post-conditions

- Let *object* be returned by procedure; if *object* is not null, then `validate(`*super*`, `*object*`)` must be `true`.

7.6.15.4 Procedure

- If `validate_(multiplier)` is `false`, then the function returns a null pointer.
- If `validate_(multiplicand)` is `false`, then the function returns a null pointer.

Let *super* denote the value of `super(((Object *) multiplier)->type, ((Object *) multiplicand)->type)`. If `product` is not a null pointer, let *type* denote the value of `((Object *) product)->type`.

- If `is_type(`*type*`)` is `false`, then the function returns a null pointer.
- If `is(`*type*`, `*super*`)` is `false`, then the function returns a null pointer.
- Otherwise the function returns the value of *super*`->mul(product, multiplier, multiplicand)`.

7.6.15.5 Invocation

Syntax
`mul_` ([[`. product =`] *product* `,`] [`. multiplier =`] *multiplier* , [`. multiplicand =`] *multiplicand*)
`mul_2_` ([`. multiplier =`] *multiplier* , [`. multiplicand =`] *multiplicand*)
`mul_3_` ([`. product =`] *product* , [`. multiplier =`] *multiplier* , [`. multiplicand =`] *multiplicand*)
Semantics
`mul_` invokes `mul_`n`_` if the expanded argument sequence contains n arguments.
`mul_2_(`*multiplier*`, `*multiplicand*`)` is equivalent to `mul_3_(NULLPTR, `*multiplier*`, `*multiplicand*`)`.
`mul_3_(`*product*`, `*multiplier*`, `*multiplicand*`)` is equivalent to the following verbose invocation:

`( (typeof_(`*product*`)) call_((Type, mul), `*product*`, `*multiplier*`, `*multiplicand*`) )`

7.6.16 div

7.6.16.1 Declarations

Prototype
`prototype_(Void_ *, (Type, div), (Void_ *, result), (Void *, dividend), (Void *, divisor))`
Type member
`Object_ *(*div)(Object_ *, Object *, Object *);`
For type *T*
T`_ *solver_(`*T*`, div)(`*T*`_ *, `*T*` *, `*T*` *);`

7.6.16.2 Pre-conditions

- `validate_(dividend)` must be `true`.
- `validate_(divisor )` must be `true`.

Let *super* denote the value of `super(((Object *) dividend)->type, ((Object *) divisor)->type)`. If `result` is not a null pointer, let *type* denote the value of `((Object *) result)->type`.

- `is_type(`*type*`)` must be `true`.
- `is(`*type*`, `*super*`)` must be `true`.

7.6.16.3 Post-conditions

- Let *object* be returned by procedure; if *object* is not null, then `validate(`*super*`, `*object*`)` must be `true`.

7.6.16.4 Procedure

- If `validate_(dividend)` is `false`, then the function returns a null pointer.
- If `validate_(divisor )` is `false`, then the function returns a null pointer.

Let *super* denote the value of `super(((Object *) dividend)->type, ((Object *) divisor)->type)`. If `result` is not a null pointer, let *type* denote the value of `((Object *) result)->type`.

- If `is_type(`*type*`)` is `false`, then the function returns a null pointer.
- If `is(`*type*`, `*super*`)` is `false`, then the function returns a null pointer.
- Otherwise the function returns the value of *super*`->div(result, dividend, divisor)`.

7.6.16.5 Invocation

Syntax
`div_` ([[`. result =`] *result* ,] [`. dividend =`] *dividend* , [`. divisor =`] *divisor*)
`div_2_` ([`. dividend =`] *dividend* , [`. divisor =`] *divisor*)
`div_3_` ([`. result =`] *result* , [`. dividend =`] *dividend* , [`. divisor =`] *divisor*)
Semantics
`div_` invokes `div_`*n*`_` if the expanded argument sequence contains *n* arguments.
`div_2_(`*dividend*`, `*divisor*`)` is equivalent to `div_3_(NULLPTR, `*dividend*`, `*divisor*`)`.
`div_3_(`*result*`, `*dividend*`, `*divisor*`)` is equivalent to the following verbose invocation:

`( (typeof_(`*result*`)) call_((Type, div), `*result*`, `*dividend*`, `*divisor*`) )`

7.7 Object class procedures

The Object class provides procedures for each Type method that has a corresponding member in the Type structure; these functions are basic stubs with minimal code, just enough to satisfy the post-conditions imposed by protocols.

The expression `type_(Object)` is a pointer to the Type structure of Object class: members of this structure provide access to procedures implemented by Object class, as described in the subsections. Any class using Object as base class inherits these procedures in its own Type structure, unless it overrides them with other functions.

The object-like macros `Object_EXTENDS` and `SELF_C_EXTENDS` are reserved for implementations of C_.

7.7.1 validate

Declaration

```
Bool_ solver_(Object, validate)(Object *this);
```

Description

The procedure returns `true` if `this` is not null and `is_type(this->type)` is `true`; otherwise it returns `false`.

7.7.2 init

Declaration

```
Object_ *solver_(Object, init)(Type type, Tape tape);
```

Description

The procedure always returns a null pointer.

7.7.3 free

Declaration

```
Void_ solver_(Object, free)(Object_ *this);
```

Description

The procedure calls `free(this)`.

7.7.4 compare

Declaration

```
LLong_ solver_(Object, compare)(Object *this, Object *that);
```

Description

The procedure returns zero if `this` and `that` compare as equal pointers; otherwise the return value is one.

7.7.5 copy

Declaration

```
Object_ *solver_(Object, copy)(Object_ *this, Object *that);
```

Description

The procedure returns `this` if it compares equal to `that`; otherwise the return value is a null pointer.

NOTE If a class provides a non-trivial implementation of `copy`, then the Object class `compare` procedure needs to be overridden as well (either by the same class or by an ancestor). This is because `solver_(Object, compare)` returns 1 for two unequal pointers, which would cause a post-condition violation in the Type protocol for `copy`.

7.7.6 `read`

Declaration
`Object_ *solver_(Object, read)(Object_ *this, Stream in);`
Description
The procedure always returns a null pointer.

7.7.7 `write`

Declaration
`LLong_ solver_(Object, write)(Object *this, Stream out);`
Description
Let *ret* be the return value for the invocation `fprintf(out, "%s@%p\n", this->type->name, (Void *)this)`. If `fflush(out)` is successful, then *ret* is the return value; otherwise the return value is `EOF` (a negative integer).

7.7.8 `parse`

Declaration
`Object_ *solver_(Object, parse)(Object_ *this, Size_ *length, Void *in);`
Description
The procedure sets `*length` to zero and returns a null pointer.

7.7.9 `text`

Declaration
`Void_ *solver_(Object, text)(Object *this, Size_ *length, Void_ *out);`
Description
Let *count* denote the return value of `snprintf(NULL, 0, "%s@%p\n\0", this->type->name, (Void *)this)`. Let *capacity* denote the existing value of `*length`; `*length` is updated to *count* before the procedure returns.

If `out` is not null and *count* does not exceed *capacity*, then `sprintf(out, "%s@%p\n", this->type->name, (Void *)this)` is called and `out` is returned. Otherwise `malloc(`*count*`)` is called, and let *des* be its return value. If *des* is not null, `sprintf(`*des*`, "%s@%p\n", this->type->name, (Void *)this)` is called. *des* is always returned.

NOTE `*length` is always set to *count* before the procedure returns; a non-null return value indicates success.

7.7.10 `decode`

Declaration
`Object_ *solver_(Object, decode)(Object_ *this, Size_ *length, Encoding *in);`
Description
The procedure sets `*length` to zero and returns a null pointer.

7.7.11 `encode`

Declaration
`Encoding_ *solver_(Object, encode)(Object *this, Size_ *length, Encoding_ *out);`
Description
The procedure always returns a null pointer.

7.7.12 `add`

Declaration

`Object_ *solver_(Object, add)(Object_ *sum, Object *augend, Object *addend);`

Description

The procedure always returns a null pointer.

7.7.13 `sub`

Declaration

`Object_ *solver_(Object, sub)(Object_ *difference, Object *minuend, Object *subtrahend);`

Description

The procedure always returns a null pointer.

7.7.14 `mul`

Declaration

`Object_ *solver_(Object, mul)(Object_ *product, Object *multiplier, Object *multiplicand);`

Description

The procedure always returns a null pointer.

7.7.15 `div`

Declaration

`Object_ *solver_(Object, div)(Object_ *result, Object *dividend, Object *divisor);`

Description

The procedure always returns a null pointer.

7.8 Creating a class

7.8.1 Declaration

Syntax

`# define` *class-name*`_EXTENDS` *base-class* [`,` *override-list*]
`class_` `(` *class-name* [`,` *interface-list*] `)` *member-declarations* `fin`
`class_0_` `(` *class-name* `,` *interface-list* `)` *member-declarations* `fin`
`class_1_` `(` *class-name* `)` *member-declarations* `fin`

Constraints

base-class shall be the name of another class that has been declared prior to the declaration of *class-name*.
interface-list shall be a comma-separated list of interface names, declared prior to the declaration of *class-name*.
The number of elements in the expanded *interface-list* shall be less than `PP_MAX`.
The identifiers `base` and `type` shall not be used as member names in *member-declarations*.
The optional *override-list* shall be a comma-separated list of method names from the `Type` structure.

Semantics

`class_` invokes `class_0_` if *interface-list* is specified; otherwise *interface-list* is omitted and `class_1_` is invoked.

`class_1_` declaration provides member details of a class structure whose tag is *class-name*. `class_1_` declares two synonyms: *class-name*`_` is a synonym for `struct` *class-name*, and *class-name* is its non-modifiable counterpart.

The first implicit member of a class is named as `base`, which is an instance of *base-class* as specified in the replacement text of *class-name*`_EXTENDS`; more precisely, the member `base` is declared before *member-declarations* as a singleton array whose element type is `struct` *base-class* (or simply *base-class*`_`). Each class structure also has a member named `type` that is inherited from the `Object` class: it is a pointer to a non-modifiable `Type` structure.

`class_1_` also declares procedures for the `Type` methods prefixed with *class-name*, along with a non-modifiable external array whose name is given by `type_(`*class-name*`)`: the array is a singleton whose element is a `Type` structure.

In addition to the facilities provided by `class_1_`, `class_0_` also declares a non-modifiable external array for each interface specified in *interface-list*. For an interface identified by *interface-name*, name of the external array is given by `typex_(`*interface-name*`,` *class-name*`)`: this array is a singleton whose element type is an extended `Type` structure identified as `struct Typex (`*interface-name*`)`, having function pointers for each method of that interface.

NOTE Interfaces provide a mechanism to extend the basic `Type` structure, discussed in the next chapter.

EXAMPLE A wrapper class provides structural encapsulation of a basic data type, and describes fundamental operations supported on it with the help of `Type` methods. As our first example, we shall declare a simple wrapper class named `Unsigned`, whose only non-trivial member is `ULLong_ value` (`base` and `type` are trivially present).

```
#ifndef UNSIGNED__
#define UNSIGNED__

#include <c._>

#define UNSIGNED_MAX 18446744073709551615ULL

#define Unsigned_EXTENDS Object,\
 validate, init, compare, copy, read, write,\
 parse, text, decode, encode, add, sub, mul, div

class_ (Unsigned)
    ULLong_ value;
fin

#endif
```

The above class declaration is available in the file `Unsigned._` located in `examples/include/class` directory. The macro `Unsigned_EXTENDS` is required by the `class_` declaration, which establishes an inheritance relationship between `Unsigned` class and `Object` class; more precisely, the member `base` of `struct Unsigned` is of type `Object_`. Successive names in the replacement text of `Unsigned_EXTENDS` comprise the optional *override-list*, specifying the basic `Type` methods that are implemented by the `Unsigned` class (more precisely, it is the procedures that are being implemented, not the protocols). Note that `free` is missing in the list, which means that `Unsigned` uses the same procedure that is inherited from `Object` class (recall that the `free` procedure implemented by `Object` class calls the standard library function `free` for memory deallocation). The word `fin` marks the end of the class declaration.

The macro `UNSIGNED_MAX` is later used in validation; for maximum portability, the value of $2^{64} - 1$ is used.

NOTE For naming structure and union members, we shall relax the C`_` convention of a trailing underscore.

7.8.2 Definition

Declarations of object-oriented types are typically placed in header files, as they are required in multiple translation units (wherever the type is used). On the contrary, every such type must have exactly one definition that can be compiled: this is because every object-oriented type has an associated `Type` structure declared as a singleton array (non-modifiable), which must be defined in exactly one translation unit (source file that is given to the compiler).

Syntax

`define_` (*name* [, *interface-list*])
`define_0_` (*class-name* , *interface-list*)
`define_1_` (*oo-type-name*)

Constraints

interface-list shall be a comma-separated list of interface names, declared prior to the declaration of *class-name*.

The number of elements in the expanded *interface-list* shall be less than `PP_MAX`.

An object-like macro named as *class-name*`_EXTENDS` or *oo-type-name*`_EXTENDS` shall remain defined, and for each *ancestor* type up to the `Object` class, a similarly named macro *ancestor*`_EXTENDS` shall also remain defined.

For each *interface-name* in *interface-list*, an object-like macro shall be defined in the following form:

`# define` *class-name*`_IMPLEMENTS_`*interface-name* *base-implementation* `,` *methods-list*

In the replacement text, *base-implementation* shall be name of the class that implements the base type of *interface-name*, as specified by the macro named as *interface-name*`_EXTENDS`; as a special rule, if *interface-name* directly extends the `Abstract` type, then *base-implementation* shall be specified as `SELF`. If the macro *interface-name*`_EXTENDS` specifies *base-interface* as the base type and *base-interface* is not `Abstract`, then an object-like macro named *base-implementation*`_IMPLEMENTS_`*base-interface* shall be defined in the same form as described above.

methods-list shall be a list of method names that are specified in the extended type structure declared as `struct Typex (`*interface-name*`)`; for each *method-name* in *methods-list*, an external function named as `solver_(`*class-name*`,` *method-name*`)` shall be declared, and this also applies for the implementation of *base-interface*.

NOTE `is(type_(`*class-name*`), type_(`*base-implementor*`))` should be `true`, but not enforced as a constraint.

Semantics

`define_` invokes `define_0_` if *interface-list* is specified; else *interface-list* is omitted and `define_1_` is invoked.

`define_1_` defines a non-modifiable external array named `type_(`*oo-type-name*`)`: this array is a singleton whose sole element is a `Type` structure, containing function pointers for the basic procedures of an object-oriented type.

For each procedure name specified in the optional *override-list* in the replacement text of the macro *oo-type-name*`_EXTENDS`, the function pointer is used to initialize the corresponding member in `type_(`*oo-type-name*`)`; otherwise the process is repeated for each ancestor type, and procedures defined by the nearest ancestor are used to initialize remaining function pointers in the non-modifiable `Type` structure, defined with static storage duration.

In addition to the facilities provided by `define_1_`, `define_0_` also defines a non-modifiable external array for each interface specified in *interface-list*. For an interface identified by *interface-name*, name of the external array is given by `typex_(`*interface-name*`,` *class-name*`)`: this array is a singleton whose element type is an extended `Type` structure identified as `struct Typex (`*interface-name*`)`, having function pointers for each method of that interface.

EXAMPLE The wrapper class `Unsigned` is defined in the file `examples/compile/class/Unsigned.c_`

```
#include "Unsigned._"

 define_ (Unsigned)
```

7.8.3 Properties

Syntax

`property_` (*oo-type-name* , *property-name*)

Semantics

`property_` can be used to create identifiers for external variables associated with an object-oriented type.

NOTE The `Type` structure associated with every object-oriented type can also be considered as a property.

Recommended practice

Objects declared with external linkage should be non-modifiable; they are generally stored in read-only segment.

7.9 Implementing procedures

To complete our example on the wrapper class `Unsigned`, we need to provide definitions for its procedures that override the default behavior inherited from `Object` class; the procedures are specified in *override-list* for the macro named as `Unsigned_EXTENDS` in the header file `Unsigned._`, and they are declared by the `class_` declaration.

NOTE We have organized each procedure in its own source file; however, this practice is not necessary.

7.9.1 validate

examples/compile/class/Unsigned/validate.c_

```
#include "Unsigned/validate._"

procedure_(Bool_, (Unsigned, validate),
(let Ptr (Unsigned), this))
    return validator(this->value);
end
```

An inline definition for `validator` function is available in `examples/include/class/Unsigned/validate._`

```
#ifndef   UNSIGNED__VALIDATE__
#define   UNSIGNED__VALIDATE__

#include "Unsigned._"

private inline Bool_ validator(let ULLong value)
begin
    return value <= UNSIGNED_MAX;
end

#endif
```

It only checks that `value` does not exceed `UNSIGNED_MAX`, which was earlier defined with the value of $2^{64} - 1$.

7.9.2 init

examples/compile/class/Unsigned/init.c_

```
#include  "Unsigned/validate._"

procedure_(Unsigned_ *, (Unsigned, init),
(let Type, type),
(let Tape, tape))
    Var value = *tape? *(ULLong *)*tape : 0;
    guard_(validator(value), NULL)
    Var object = new__(Unsigned);
    guard_(object, NULL)
    object->type  = type ;
    object->value = value;
    return object;
end
```

If the first element of `tape` array is not null, then it is interpreted as a pointer to `ULLong`, and the same is dereferenced; otherwise the value zero is used. If the dereferenced value exceeds `UNSIGNED_MAX`, then a null pointer is returned. Otherwise a new `Unsigned` object is allocated; if the allocation fails, a null pointer is returned. For a successful allocation, the members `type` and `value` are initialized, and a pointer to the new instance is returned.

7.9.3 compare

`examples/compile/class/Unsigned/compare.c_`

```
#include "Unsigned._"

procedure_(LLong_  , (Unsigned, compare),
(let Ptr (Unsigned), this),
(let Ptr (Unsigned), that))
    return this->value < that->value
    ? -1 : this->value > that->value;
end
```

Following the established conventions expected by functions like `bsearch` and `qsort`, the return value is negative if left value is less than right value, zero if they are equal, and positive if left value is greater than right value.

NOTE Directly returning the difference between `this->value` and `that->value` can cause signed overflow.

7.9.4 copy

`examples/compile/class/Unsigned/copy.c_`

```
#include  "Unsigned._"

procedure_(Unsigned_*, (Unsigned, copy),
(let  Ptr (Unsigned_), this),
(let  Ptr (Unsigned ), that))
    Var  copy = need__(this);
    guard_(copy, NULL)
    if (!this) copy->type = type_(Unsigned);
    copy->value = that->value;
    return copy;
end
```

A new allocation is done only if `this` is not null; if the allocation fails, a null pointer is returned. Otherwise `copy->value` is set to `that->value`, and if a new allocation was performed, then its `type` member is initialized to `type_(Unsigned)`. The return value is a pointer to the instance.

7.9.5 read

`examples/compile/class/Unsigned/read.c_`

```
#include  "Unsigned/validate._"
```

```
procedure_(Unsigned_*, (Unsigned, read),
(let Ptr  (Unsigned_), this),
(let Stream, in))
    return input__(in, this->value) == 1
    && validator(this->value) ? this : NULL;
end
```

If data can be successfully read from the input stream `in` and the resulting value does not exceed `UNSIGNED_MAX`, then the pointer `this` is returned; otherwise the return value is `NULL`.

7.9.6 write

examples/compile/class/Unsigned/write.c_

```
#include "Unsigned._"

procedure_(LLong_, (Unsigned, write),
(let Ptr (Unsigned), this),
(let Stream, out))
    return output__(out, this->value);
end
```

The integer `this->value` is written to the output stream `out`, and the number of characters written is returned.

7.9.7 parse

examples/compile/class/Unsigned/parse.c_

```
#include  "Unsigned/validate._"

#include  <errno._>

procedure_(Unsigned_*, (Unsigned, parse),
(let  Ptr (Unsigned_), this),
(let  Ptr (Size_), length),
(let  Ptr (Void ), in))
    Void_ *memchr(Void *, Int, Size);
    guard_(memchr(in, '\0', *length), NULL)
    Auto_  endptr_ = (Char_ *)unqual__(in);
    errnum_ = 0;
    ULLong_ strtoull(String, Char_ **, Int);
    Var value = strtoull(in, &endptr_, 0);
    guard_((*length = endptr_-(Char *)in) &&  !errnum
    && validator(value), (errnum_ = 0, NULL))
    this->value = value;
    return this;
end
```

Our implementation of parse expects in to be a valid string; the initial value of *length is expected to be capacity of the character array, and if none of its elements match the null byte, then the input is assumed to be invalid, and a null pointer is returned. Otherwise the string is interpreted as an unsigned integer using the standard library function strtoull (declared in <stdlib.h>); the third argument corresponding to base is specified as zero, which supports octal (prefixed with 0) and hexadecimal (prefixed with 0X or 0x) notations along with decimal form.

The modifiable pointer endptr_ records the address of the first invalid character that could not be converted; consequently, its offset relative to the base pointer in gives the number of valid characters that were correctly interpreted, which is stored in *length. errnum_ (an alternative name for errno defined in the C_ extension header <errno._>) is set to zero before calling strtoull, and if it becomes non-zero after the call, or if zero characters were read, or the converted value exceeds UNSIGNED_MAX, then errnum_ is reset to zero and a null pointer is returned. Otherwise the converted value is assigned to this->value, and a pointer to the instance is returned.

7.9.8 text

examples/compile/class/Unsigned/text.c_

```
#include "Unsigned._"

procedure_(Void_ *, (Unsigned, text),
(let Ptr (Unsigned), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    Var buflen = output__((String_){0}, this->value) + 1U;
    Var buf = (out && *length >= buflen)? (String_*)out : new__(Char_[buflen]);
    *length = buflen;
    guard_(buf, NULL)
    output__(*buf, this->value);
    return buf;
end
```

The first invocation of output__ counts the number of characters that would be present in the decimal form of this->value; 1U is added to make room for a terminating null byte. If out is not a null pointer, it is expected to be a character array having a capacity of (at least) *length elements; if *length is not less than the required buffer length, then out is used to store the text representation. Otherwise a new character array of length buflen is allocated; if the allocation fails, then a null pointer is returned. However, *length is always updated to buflen regardless of whether the allocation succeeds or not; if it fails, then *length indicates the capacity that would be required. The second invocation of output__ actually writes the text representation of this->value to the array, and the latter is returned by the function (pointer to an array has the same address as base element of the array).

7.9.9 decode

examples/compile/class/Unsigned/decode.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, decode),
(let Ptr (Unsigned_), this),
(let Ptr (Size_), length),
(let Ptr (Encoding), in))
    guard_(8 <= *length, NULL)
```

```
    *length = 8;
    this->value =
     ((*in)[0] & 255)   +
    (((*in)[1] & 255U   ) << 8 ) +
    (((*in)[2] & 255UL  ) << 16) +
    (((*in)[3] & 255UL  ) << 24) +
    (((*in)[4] & 255ULL) << 32) +
    (((*in)[5] & 255ULL) << 40) +
    (((*in)[6] & 255ULL) << 48) +
    (((*in)[7] & 255ULL) << 56) ;
    return this;
end
```

`in` points to an array of `UByte`, whose data is interpreted as the little-endian representation of an unsigned integer. The initial value of `*length` is expected to be length of the encoding array; only the first eight bytes are interpreted, and if the array is shorter than that, then a null pointer is returned. Otherwise `*length` is set to eight, indicating the number of bytes that were interpreted. The bitwise-AND with `255` is meant for systems where `CHAR_BIT` is more than eight; for most execution environments, a byte is an octet of bits, and the bitwise-AND is likely to be optimized away by compilers. However, the suffixes `UL` and `ULL` are significant, as they promote the left operand of the shift to (possibly) wider type, ensuring that there is enough room to accommodate the shifted bits.

7.9.10 encode

examples/compile/class/Unsigned/encode.c_

```
#include "Unsigned._"

procedure_(Encoding_*, (Unsigned, encode),
(let Ptr (Unsigned), this),
(let Ptr (Size_) , length),
(let Ptr (Encoding_), out))
    Var enc = (out && 8 <= *length)? out : new__(UByte_ [8]);
    *length = 8;
    guard_(enc, NULL);
    Var value = this->value;
    (*enc)[0] = value      & 255;
    (*enc)[1] = value>>8  & 255;
    (*enc)[2] = value>>16 & 255;
    (*enc)[3] = value>>24 & 255;
    (*enc)[4] = value>>32 & 255;
    (*enc)[5] = value>>40 & 255;
    (*enc)[6] = value>>48 & 255;
    (*enc)[7] = value>>56 & 255;
    return enc;
end
```

`*length` is expected to be the capacity of `out` array. If `out` is not null and `*length` is not less than eight, then it is used for storing the encoding; otherwise an array of eight bytes is allocated. The length is always set to eight, but if the allocation fails, then a null pointer is returned. Otherwise the bits of `this->value` are divided into octets and stored in little-endian order (less significant octet in lower address), and a pointer to the encoding is returned.

7.9.11 add

examples/compile/class/Unsigned/add.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, add),
(let Ptr (Unsigned_), sum),
(let Ptr (Unsigned ), augend),
(let Ptr (Unsigned ), addend))
    Var result = need__(sum);
    guard_(result, NULL)
    if (!sum) result->type = type_(Unsigned);
    result->value = (augend->value + addend->value) & UNSIGNED_MAX;
    return result;
end
```

If `sum` is not null, then it is used for storing the result; otherwise a new allocation is performed. If the allocation fails, a null pointer is returned; otherwise the member `type` of the new object is initialized to `type_(Unsigned)`. The sum of `augend->value` and `addend->value` is stored in `result->value`, and pointer to the instance is returned.

NOTE Unsigned arithmetic follows implicit wraparound, so there is no possibility of integer overflow; however, if `width_(ULLong)` is greater than 64 on some execution environment, then the sum can be greater than `UNSIGNED_MAX`, which would violate the post-condition that invokes `validate` on a non-null return value. To avoid such a mishap, bitwise-AND with `UNSIGNED_MAX` (64 bits set to 1) is used to truncate the result (most systems support exactly 64 bits in `ULLong`, so the bitwise-AND is redundant and likely to be optimized away by compilers).

7.9.12 sub

examples/compile/class/Unsigned/sub.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, sub),
(let Ptr (Unsigned_), difference),
(let Ptr (Unsigned ), minuend),
(let Ptr (Unsigned ), subtrahend))
    Var result = need__(difference);
    guard_(result, NULL)
    if (!difference) result->type = type_(Unsigned);
    result->value = minuend->value - subtrahend->value;
    return result;
end
```

Code for subtracting two `Unsigned` instances is similar to that of addition; bitwise-AND with `UNSIGNED_MAX` is not needed because difference of two unsigned integers (each not exceeding `UNSIGNED_MAX`) cannot be out of range.

7.9.13 mul

examples/compile/class/Unsigned/mul.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, mul),
(let Ptr (Unsigned_), product),
(let Ptr (Unsigned ), multiplier),
(let Ptr (Unsigned ), multiplicand))
    Var result = need__(product);
    guard_(result, NULL)
    if (!product) result->type = type_(Unsigned);
    result->value = (multiplier->value * multiplicand->value) & UNSIGNED_MAX;
    return result;
end
```

The code for `mul` is almost entirely identical to that of `add`, except that addition is replaced with multiplication.

7.9.14 div

examples/compile/class/Unsigned/div.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, div),
(let Ptr (Unsigned_), output),
(let Ptr (Unsigned ), dividend),
(let Ptr (Unsigned ), divisor))
    guard_(divisor->value, NULL)
    Var result = need__(output);
    guard_(result, NULL)
    if (!output) result->type = type_(Unsigned);
    result->value = dividend->value / divisor->value;
    return result;
end
```

`div` uses a guard clause to check that divisor is non-zero; rest of the code is similar to other arithmetic methods.

7.10 More examples

We have already seen how to create a user defined class, which broadly involves three steps: declaration (of class structure), definition (of type structure), and implementation of procedures. We can create simple wrapper classes for any basic data type; however, creating a new class for each and every C type would add little value and a lot of clutter. We have restrained our examples to only four wrapper classes: `Unsigned`, `Signed`, `Rational`, and `Text`.

`Signed` is a wrapper class for the fixed-point type `LLong_`, whereas `Rational` is a wrapper class for the floating-point type `Float_`. The wrapper class `Text` contains two non-trivial members: `buffer` is a pointer to `Char_`, and `length` is of type `Size_`. The implementation of these wrapper classes are mostly similar to that of the `Unsigned` class, and for the sake of brevity, their full implementation details have been moved to appendix A. So far we have implemented only the basic methods; in the next subsections, we shall associate additional methods with a class.

7.10.1 Linear linked list

A linked list is a concrete data structure for representing an ordered sequence of data. Unlike arrays, a linked list is not required to support direct access of elements: to access an element, each element that precedes it needs to be visited as well. This is because the elements are not required to be stored in physically contiguous locations, but each element stores a pointer to its next element. Each element is called a node and has two primary components: a data value, and a pointer to its successor; the latter establishes a logical continuation between successive elements. The first node of a list is called the head node, which is a starting point to reach any node in that list; similarly, the last node is called the tail node, which does not have any successor for a non-circular linked list implementation.

NOTE The tail node is useful for appending an element in constant time, which is often a frequent operation.

7.10.1.1 Declaration

The following code declares a linear linked list; a similar code is available in `examples/include/class/Chain._`

```
#ifndef CHAIN__
#define CHAIN__

#define Chain_EXTENDS Object,\
 validate, init, free, compare, copy, add

typedef_(Node, struct Node
{   Void  *data;
    struct Node *next;
})

class_ (Chain)
    Node_ *head;
    Size_  length;
    Node_ *tail;
fin

prototype_(Bool_ , (Chain, append),
(Chain_ *, this) , (Void *, data))

#endif
```

The macro `Chain_EXTENDS` establishes an inheritance relationship with the `Object` class, and specifies the methods that are overridden by the `Chain` class. `Node_` is a synonym for a modifiable structure with two members, `data` and `next`: `data` is a pointer to a non-modifiable object, and `next` is a pointer to a modifiable `Node_` object.

The class structure contains three non-trivial members: `head` and `tail` are respectively pointers to the first and last nodes, and `length` is used to maintain a count of the current number of elements in the list (in constant time).

The `append` method is meant to extend a list by adding a node after tail node (the new node becomes the tail).

7.10.1.2 Definition

The following code defines the `Type` structure for the class `Chain`, using the relationship specified in `Chain_EXTENDS`.

```
#include"Chain._"

define_ (Chain)
```

7.10.1.3 Validation

Implementation of `validate` for `Chain` is left as an exercise in this chapter. The following checks can be performed:

- Both `head` and `tail` must not be null, or both must be null (`head iff tail`).
- `length` must be zero if and only if both `head` and `tail` are null (`length iff head`).
- The count of distinct nodes from `head` through `tail` must be equal to `length`.
- If `tail` is not null, then its `next` member must be a null pointer (`tail implies ! tail->next`).

NOTE In the next chapter, we shall implement `validate` procedure of `Chain` class using `Iterable` interface.

7.10.1.4 Constructor

The following procedure code is available in the source file `examples/compile/class/Chain/init.c_`

```
#include  "Chain._"

procedure_(Chain_ *, (Chain, init),
(let Type, type),
(let Tape, tape))
    (Void) tape;
    Var chain = new__(Chain);
    if (chain) *chain = (Chain){.type = type};
    return chain;
end
```

The implementation first creates a new `Chain` object, and if the allocation is successful, then its `type` member is initialized to the argument for `type` parameter. Rest of the members are initialized as if with the integer constant `0`: `head` and `tail` are each set to the null pointer, and `length` is initialized to zero, which indicates an empty list.

NOTE `(Void) tape` is a pseudo-use of the parameter `tape`, for suppressing a warning on unused parameters.

7.10.1.5 Destructor

The following procedure code is available in the source file `examples/compile/class/Chain/free.c_`

```
#include "Chain._"

procedure_((Chain, free),
(let Ptr (Chain_), this))
    if (!validate_(this)) this->head = NULL;
    for (let Ptr_(Node_) node_,
    next_ = this->head; (node_ = next_); free(node_))
        next_ = node_->next;
    free(this);
end
```

If the parameter `this` does not point to a valid `Chain` object, then its `head` member is set to a null pointer. Otherwise, each node is deallocated from `head` through `tail`, and finally the `Chain` object itself is deallocated.

NOTE We have not used a guard clause for `this` to be non-null, as it is checked by `free` procedure of `Type`.

7.10.1.6 Appending

The protocol for `append` method is left as an exercise in this chapter. The following conditions can be established:

- *Pre-condition*: The parameter `this` must point to a valid `Chain` object.
- *Post-condition*: `this` must point to a valid `Chain` object after appending. If the procedure returns `false`, then the operation failed, and all members of `this` must remain unchanged; otherwise a new node was appended, whose member `data` must be same as the parameter `data`, and `this->length` must have been incremented.

In the next chapter, we shall design the `append` protocol for `Chain` class with the help of `Iterable` interface. For now, we shall discuss the following procedure available in the source file `examples/compile/class/Chain/append.c_`

```
#include "Chain/append._"

procedure_(Bool_ , (Chain, append),
(let Ptr (Chain_), this),
(let Ptr ( Void ), data))
    Var node = new__(Node);
    guard_(node, FALSE_())
    node->data = data;
    node->next = NULL;
    if  (this->length++) this->tail->next = node;
    else this->head = node;
    this->tail = node;
    return TRUE_();
end
```

If a new node cannot be allocated, then zero is returned; otherwise the member `data` is set to the parameter `data`, and the member `next` is set to a null pointer. The length is updated, and if its existing value is non-zero, then the new node is linked to the `next` member of current tail node; otherwise the chain is currently empty, and the new node becomes the head node. In any case, the newly appended node always becomes the tail node.

7.10.2 Typed linked list

The `Chain` class does not implement several procedures associated with `Type` structure. To overcome this limitation, we can extend the `Chain` class with an additional `Type` member, and impose a validation rule that each element is required to be a valid instance of the `Type` specified for a given list. This is done with another class named `List`.

7.10.2.1 Declaration

The following code declares a typed linked list; a similar code is available in `examples/include/class/List._`

```
#ifndef LIST__
#define LIST__

#include "Chain._"

#define List_EXTENDS Chain,\
 validate, init, compare, copy, read,\
 write, parse, text, decode, encode, add
```

```
class_ (List)
    Type_ species;
fin

prototype_(Bool_, (List, append),
(List_ *, this), (Void *, data))

prototype_(Type_, (List, species),
(List_ *, this), (Type, species))

#endif
```

List extends the Chain class with an additional member species, which is a pointer to a Type structure. For a given List instance, each of its elements must point to a data object that is a valid instance for the species of that list. Recall that due to Liskov substitution, each instance of a derived type is required to be a valid instance of its base type as well, so instances of types that extend the given species can be added to the list as well.

The macro List_EXTENDS establishes an inheritance relationship with the Chain class: more precisely, the implicit member base of List class is declared as a singleton array with element type as Chain. List_EXTENDS also specifies the procedure names that are overridden by the List class; rest are inherited from the Chain class.

7.10.2.2 Definition

The following code defines the Type structure for the class List, using the relationship specified in List_EXTENDS.

```
#include"List._"

define_ (List)
```

7.10.2.3 Constructor

The following procedure code is available in the source file examples/compile/class/List/init.c_

```
#include  "List._"

procedure_(List_ *, (List, init),
(let Type, type),
(let Tape, tape))
    Var species = (Type *)*tape;
    guard_((species implies  is_type(*species)), NULL)
    Var list  =  new__(List);
    if (list) *list = (List)
    {   .type    = type,
        .species = species? *species : type_(Object),
    };
    return list;
end
```

If the first element in the tape array is not null, it is expected to be pointer to a struct Type pointer, Type being a synonym for Ptr (const struct Type). If it is not null and does not refer to a valid Type structure, then a null pointer is returned. Otherwise a new List object is created, and on successful allocation, its type member is initialized to the value of type parameter. If the first element of tape array is not null, it is dereferenced to get a pointer to a Type structure, used to initialize the species of the new instance; otherwise type_(Object) is used.

7.10.2.4 Appending

The method append is used to extend a List instance at its tail node. A separate prototype has been declared only because the protocol specifies pre-conditions in addition to those established by append method of the Chain class; the procedure itself is trivial and simply calls the append procedure of Chain class, without any extra functionality.

The following procedure code is available in the source file examples/compile/class/List/append.c_

```
#include "List/append._"

procedure_(Bool_, (List, append),
(let Ptr (List_), this),
(let Ptr (Void ), data))
    Bool_  solver_(Chain, append)(Chain_ *, Void *);
    return solver_(Chain, append)(this->base, data);
end
```

NOTE A prototype does not declare procedure, so solver_(Chain, append) is declared prior to invocation.

7.10.2.5 Type relaxation

The method species is used to obtain and possibly relax the existing species of a List instance to an ancestor type. We have left the protocol definition as an exercise, which shall be implemented in the next chapter using Collection interface. Following are the basic pre-conditions that can be established by the protocol:

- this must point to a valid instance of List class.
- If species is not null, then it must point to a valid Type structure.

The following procedure code is available in the source file examples/compile/class/List/species.c_

```
#include "List._"

procedure_(Type_, (List, species),
(let Ptr (List_), this),
(let Type, species))
    guard_(species, this->species)
    return this->species = this->base->length?
     super(this->species, species) : species;
end
```

If the parameter species is a null pointer, then the existing value of this->species is returned.

If the list is empty, then this->species is directly updated to species; otherwise this->species is updated to the nearest common ancestor of species and the existing value of this->species. It ensures that this->species is always updated to an ancestor type, and due to Liskov substitution principle, each existing element in the list would be a valid instance of the super type. So a non-empty list would remain valid after relaxing the type.

7.10.2.6 Other procedures

In this chapter, we have discussed only the implementation of `init` for the `List` class; some other procedures will be implemented in the next chapter, demonstrating how inheritance and abstraction promote code reusability. We conclude this chapter with an overview of how these procedures can be implemented (can be done as an exercise).

- `validate` should check that `is_type(species)` is `true`, and the `data` member of each node in the list points to a valid instance of `species`. Recall that the general `validate` function does validation with respect to the base type first, so we need not check that the list structure is valid, as it is already done by the `Chain` class. To allow instances of any object-oriented type in a list, its `species` can be initialized to `type_(Object)`.
- `compare` should adopt a similar rule as `strcmp`: for a pair of elements, one from each list in identical positions, find their super type and invoke its `compare` procedure with the instances as arguments. If the outcome is non-zero, then that is also the return value of comparing the lists. Otherwise the same is repeated for successive pairs until one of the lists ends: if the other list ends as well, then the lists are considered to be equal, and the return value should be zero; otherwise the shorter list is considered to be "less than" the longer list.
- `copy` should perform a "shallow copy" of the source list: only the data pointers are copied to nodes of the destination list (allocated as required), as opposed to invoking `copy` procedure for each instance. The primary reason behind this is to avoid issues of memory allocation failure while creating a "deep copy" of each instance.
- `read`, `parse`, and `decode` should each invoke its corresponding procedure from the `Type` specified by `species`.
- `write`, `text`, and `encode` should each invoke its corresponding procedure from the `Type` given by each instance.
- `add` should concatenate the two lists: elements of the left list are to be followed by elements of the right list.

Chapter 8

Interfaces

"*The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.*" — Donald Knuth

An interface in C_ defines a structure of function pointers for operations permitted on an abstract data type. Unlike concrete data types implemented with classes whose data members are exposed by the declaration, abstract data types do not specify how the data is internally organized or represented. Interfaces provide an insulating layer over data that prevents direct access and modification: all operations need to be done via functions pointers that are part of the interface structure. In other words, the behavior is specified, but the representation is opaque.

As an analogy, the powers and functions of a national government are described by the constitution of that nation, which formalizes what a government can (or cannot) do. An interface is something similar, and in C_, the expected behavior is described with pre-conditions and post-conditions in protocols. A government is materialized by political parties, who must abide by the laws formulated in the constitution. Similarly, an interface is an abstract specification of behavior, and to actually perform these operations on an object instance, we need functions that know how the data is internally represented. These functions are provided by classes; more precisely, these functions are implemented as procedures of class methods. A class that provides the functions required by an interface structure is said to "concretize" or implement that interface, and instances of that class are said to materialize that interface. An interface can be concretized by multiple classes, and a single class can implement multiple interfaces.

Besides the aesthetic appeal of representation-agnostic code, interfaces promote modular design, low coupling, and provide opportunities of code reuse through dependency inversion. The usual requirement is that every interface depends on a concrete class that implements the necessary functions as procedures. However, once the basic operations have been defined for an abstract type, higher order functionalities can be directly implemented by the interface itself using the basic operations. If such an interface is concretized by multiple classes, then each class can utilize these higher order functionalities without having to re-implement the same logic. In other words, each implementing class that concretizes the basic operations depends on the interface for higher order functionalities that are provided by the interface. While we refer to this as "dependency inversion", it is part of a mutual interdependence between abstract and concrete types, and if done correctly, it can also reduce the chances of programming bugs.

Recall that each class has an associated `Type` structure that provides pointers to procedures implemented by that class (or inherited from its base class). The same benefit is also provided to each interface for the basic `Type` methods. Additionally, the structure of function pointers declared by an interface is actually an extension of the basic `Type` structure discussed in the previous chapter, which is done by inheriting from the `Abstract` type.

NOTE Unlike some object-oriented languages, C_ does not support multiple inheritance among interfaces.

8.1 Abstract type

`Abstract` type extends the `Object` class, and by virtue of inheritance, `struct Abstract` contains a member `base` defined as a singleton array whose element type is `Object_`. `Abstract_` is a synonym for `struct Abstract`, while `Abstract` is its non-modifiable counterpart. The member `type` is an alias for `base->type`, which is initialized by the `Abstract` constructor to point to the `Type` structure of `Abstract`, containing pointers to overridden procedures. `Abstract` structure also contains a pointer member that can be accessed with three names having different types:

- The name `concrete` is declared as pointer to `Void`, referring to a non-modifable object.
- The name `concrete_` is declared as pointer to `Void_`, referring to a modifiable object.
- The name `_concrete` is a pointer to an untagged union with two members, both aliased to the same pointer: `type` is a pointer to `struct Type`, and `typex` is a pointer to an extended `Type`, with `Typex_(Abstract)` being a synonym for the pointer type (`Type` extension using interfaces is documented in a later section).

When an interface is instantiated with `Type` of a concrete class, the above three names alias to the same member, pointing to an instance of the class that concretizes the interface. The member `type` (or `base->type`) provides the abstract lineage, while `_concrete->type` points to `Type` of the implementing class, providing the concrete lineage.

The macros `Abstract_EXTENDS` and `SELF_IMPLEMENTS_Abstract` are reserved for implementations of C_.

NOTE The generic pointer type aliases are used to avoid boilerplate casts during assignments and invocations.

8.1.1 concrete_

The names `concrete` and `concrete_` pointers to `Void` and `Void_` (respectively), whereas `_concrete` has a different type, which is pointer to an untagged union. The ISO C standard does not disallow union pointers to have a different representation than `void` pointer; as a purely artificial example, consider a hypothetical execution environment where pointers to union are represented with flipped bits, and the actual memory address can be obtained by toggling each bit, which is equivalent to taking one's complement of the pointer data interpreted as a binary integer. Therefore, at least in theory, a `void` pointer representation need not be identical to a union pointer, and interpreting one pointer type's binary representation as another pointer type would cause undefined behavior.

In practice, virtually all existing environments have identical representations for all object pointers; in fact, the POSIX standard has a stronger requirement that function pointers must also be representable as `void` pointers. This implies that if pointer to a concrete instance is stored using the names `concrete` or `concrete_`, the stored representation is that of a `void` pointer, and it can be safely interpreted as union pointer due to their identical representation (on real-world systems). However, out of purely pedantic concerns, it is recommended that the name `_concrete` should not be directly accessed, and the name `concrete` should be used for storing a concrete instance.

A non-lvalue expression having the same type as `_concrete` can be obtained with the macro `concrete_`.

Syntax

`concrete_ (` *expression* `)`

Constraints

Type of *expression* shall be pointer to `Abstract`, or pointer to an interface type that inherits from `Abstract`. Equivalently, *expression* shall be of type "pointer to T" or "pointer to T`_`", and `is_(`T`, Abstract)` shall be `true`.

NOTE `is(Abstract, Abstract)` is trivially satisfied due to the reflexive nature of "is-a" relationship.

Semantics

`concrete_` dereferences *expression* to read the member name `concrete_` and casts it to the type of `_concrete`.

`concrete_(`*expression*`)` is equivalent to `((typeof ((`*expression*`)->_concrete)) (`*expression*`)->concrete_)`.

NOTE The macro `concrete_` uses the name `_concrete` only to get its data type, not for reading the pointer. Nevertheless, the cast is a no-op for practically all systems, and the outcome is same as `(`*expression*`)->_concrete`.

8.2 `Abstract` procedures

The `Abstract` type overrides all procedures of the `Object` class, under the assumption that these functions will be called with valid instances of `Abstract`, or instances of interfaces that inherit from `Abstract` (Liskov substitution).

8.2.1 `validate`

Declaration

```
Bool_ solver_(Abstract, validate)(Abstract *this);
```

Description

The procedure returns `true` if `is(this->type, type_(Abstract))` and `validate_(this->concrete)` are `true`.

8.2.2 `init`

Declaration

```
Abstract_ *solver_(Abstract, init)(Type type, Tape tape);
```

Description

If `is(type, type_(Abstract)` is `false` or the first element of `tape` is null, a null pointer is returned. Otherwise the first element of `tape` is assumed to be a `Type *` (`Type` is a synonym for `Ptr (const struct Type)`), and if the dereferenced value is not a valid concrete type (as determined by calling `is_typex`), then a null pointer is returned.

After validating both arguments, a new `Abstract` instance is created, and if the allocation fails, then a null pointer is returned. Otherwise `init_` is called with two arguments: first is concrete type obtained by dereferencing the first element of `tape`, and second is `tape + 1`, which refers to the sub-array after excluding the first element at index zero. If `init_` returns null, then the new `Abstract` instance is deallocated and a null pointer is returned. Otherwise the `type` member of `Abstract` instance is initialized with the `type` parameter, and the `concrete` member is initialized with the outcome of `init_`. The return value is a pointer to the initialized `Abstract` instance.

8.2.3 `free`

Declaration

```
Void_ solver_(Abstract, free)(Abstract_ *this);
```

Description

If `this` is a null pointer or `is(this->type, type_(Abstract))` is `false`, then the procedure returns immediately. Otherwise if the concrete instance is not a null pointer and its `type` member is valid (as determined by `is_type`), then the concrete instance is deallocated by calling `concrete_(this)->type->free(this->concrete_)`.

In any case, `free(this)` is invoked to deallocate the abstract instance before returning.

8.2.4 `compare`

Declaration

```
LLong_ solver_(Abstract, compare)(Abstract *this, Abstract *that);
```

Description

If either `this->type` or `that->type` is not a descendant of `type_(Abstract)`, then the return value is one. Let *super* denote the nearest common ancestor given by `super(concrete_(this)->type, concrete_(that)->type)`. Then the return value of the comparison is the outcome of *super*`->compare(this->concrete, that->concrete)`.

8.2.5 copy

Declaration

`Abstract_ *solver_(Abstract, copy)(Abstract_ *this, Abstract *that);`

Description

As per the protocol of copy method, this is considered to be the destination pointer, and that is considered to be the source pointer. If is(that->type, type_(Abstract)) is false, then a null pointer is returned. Otherwise if that is not null and the pointer that->concrete compares equal to the pointer this->concrete, then this is returned. If this->concrete is not null and either the concrete type is not valid or not a sub-type of concrete_(that)->type, then a null pointer is returned. Otherwise the copy procedure of concrete_(that)->type is invoked with this->concrete_ and that->concrete as arguments. If its outcome is null, then a null pointer is returned; otherwise its outcome is a pointer to a copy of that->concrete, and if it is same as this->concrete, then this is returned. Otherwise a new Abstract instance is allocated, whose type member is initialized with the type parameter, and concrete member is initialized with the pointer that was returned by concrete_(that)->type->copy(this->concrete, that->concrete). The return value is a pointer to the initialized Abstract instance.

8.2.6 read

Declaration

`Abstract_ *solver_(Abstract, read)(Abstract_ *this, Stream in);`

Description

If is(this->type, type_(Abstract)) is false, or this->concrete is null, or is_type(concrete_(this)->type) is false, then a null pointer is returned. Otherwise the read procedure of concrete_(this)->type is invoked with this->concrete_ and in as arguments. If the outcome of the call is null, then a null pointer is returned; if the outcome is same as this->concrete, then this is returned. Otherwise a new Abstract instance is allocated, whose type member is initialized with this->type, and concrete member is initialized with the earlier outcome of calling concrete_(this)->type->read. The return value is a pointer to the initialized Abstract instance.

8.2.7 write

Declaration

`LLong_ solver_(Abstract, write)(Abstract *this, Stream out);`

Description

If is(this->type, type_(Abstract)) is false, then -1 is returned; otherwise the return value is the outcome of calling the write procedure of concrete_(this)->type with this->concrete and out as arguments.

8.2.8 parse

Declaration

`Abstract_ *solver_(Abstract, parse)(Abstract_ *this, Size_ *length, Void *in);`

Description

If is(this->type, type_(Abstract)) is false, or this->concrete is null, or is_type(concrete_(this)->type) is false, then a null pointer is returned. Otherwise the parse procedure of concrete_(this)->type is invoked with this->concrete_, length, and in as arguments. If the outcome of the call is null, then a null pointer is returned; if the outcome is same as this->concrete, then this is returned. Otherwise a new Abstract instance is allocated, whose type member is initialized with this->type, and concrete member is initialized with the earlier outcome of calling concrete_(this)->type->parse. The return value points to the initialized Abstract instance.

8.2.9 `text`

Declaration

`Void_ *solver_(Abstract, text)(Abstract *this, Size_ *length, Void_ *out);`

Description

If `is(this->type, type_(Abstract))` is `false`, a null pointer is returned; otherwise return value is the outcome on calling `text` procedure of `concrete_(this)->type` with `this->concrete`, `length`, and `out` as arguments.

8.2.10 `decode`

Declaration

`Abstract_ *solver_(Abstract, decode)(Abstract_ *this, Size_ *length, Encoding *in);`

Description

If `is(this->type, type_(Abstract))` is `false`, or `this->concrete` is null, or `is_type(concrete_(this)->type)` is `false`, then a null pointer is returned. Otherwise the `decode` procedure of `concrete_(this)->type` is invoked with `this->concrete_`, `length`, and `in` as arguments. If the outcome of the call is null, then a null pointer is returned; if the outcome is same as `this->concrete`, then `this` is returned. Otherwise a new `Abstract` instance is allocated, whose `type` member is initialized with `this->type`, and `concrete` member is initialized with the earlier outcome of calling `concrete_(this)->type->decode`. A pointer to the initialized `Abstract` instance is returned.

8.2.11 `encode`

Declaration

`Encoding_ *solver_(Abstract, encode)(Abstract *this, Size_ *length, Encoding_ *out);`

Description

If `is(this->type, type_(Abstract))` is `false`, a null pointer is returned; otherwise return value is the outcome on calling `encode` procedure of `concrete_(this)->type` with `this->concrete`, `length`, and `out` as arguments.

8.2.12 `add`

Declaration

`Abstract_ *solver_(Abstract, add)(Abstract_ *sum, Abstract *augend, Abstract *addend);`

Description

If `augend->type` or `addend->type` is not a sub-type of `type_(Abstract)`, then a null pointer is returned. Otherwise let *super* denote the nearest common ancestor of the concrete types for `augend` and `addend`, as given by `super(concrete_(augend)->type, concrete_(addend)->type)`. If `sum` and `sum->concrete` are not null, and `concrete_(sum)->type` is not a valid type or not a sub-type of *super*, then a null pointer is returned. Otherwise let *result* be the outcome of *super*`->add(sum->concrete_, augend->concrete, addend->concrete)` (if `sum` is null, then a null pointer is used instead of `sum->concrete_`). If *result* is null, then a null pointer is returned; if *result* is same as `sum->concrete`, then `sum` is returned. Otherwise a new `Abstract` instance is allocated, whose `type` member is initialized with `super(augend->type, addend->type)` (nearest common ancestor of abstract types), and `concrete` member is initialized with *result*. The return value is a pointer to the initialized `Abstract` instance.

8.2.13 `sub`

Declaration

```
 Abstract_ *solver_(Abstract, sub)
(Abstract_ *difference, Abstract *minuend, Abstract *subtrahend);
```

Description

Implementation of the procedure `solver_(Abstract, sub)` is identical to that of `solver_(Abstract, add)`.

8.2.14 `mul`

Declaration

```
 Abstract_ *solver_(Abstract, mul)
(Abstract_ *product, Abstract *multiplier, Abstract *multiplicand);
```

Description

Implementation of the procedure `solver_(Abstract, mul)` is identical to that of `solver_(Abstract, add)`.

8.2.15 `div`

Declaration

```
Abstract_ *solver_(Abstract, div)(Abstract_ *result, Abstract *dividend, Abstract *divisor);
```

Description

Implementation of the procedure `solver_(Abstract, div)` is identical to that of `solver_(Abstract, add)`.

8.3 Creating an interface

8.3.1 Declaration

Syntax

`# define` *interface-name*`_EXTENDS` *base-interface* [`,` *override-list*]
`Interface_ (` *interface-name* `)`
`prototype_ (` [*return-type* `,`] `(` *interface-name* `,` *member-name* `)` [`,` *parameter-tuples*] `)`
`interface_ (` *interface-name* `,` *members-list* `)`

Constraints

base-interface shall be the name of another interface that has been declared prior to the declaration of *class-name*.

members-list shall be a comma-separated list of names, and for each *member-name* in *member-list*, a `prototype_` declaration with the identifier tuple `(`*interface-name*`,` *member-name*`)` shall be visible prior to the use of `interface_`.

members-list shall be non-empty and the number of elements after macro expansions shall be less than `PP_MAX`.

The optional *override-list* shall be a comma-separated list of method names from the `Type` structure.

Semantics

Declaring an interface involves the following four steps that are required to be done in the given order:

1. An object-like macro named as *interface-name*`_EXTENDS` is defined first, which establishes an inheritance relationship with another interface named as *base-interface*, and if *interface-name* overrides any of the `Type` procedures inherited from *base-interface*, then those are required to be specified in *override-list*.

2. The next step is to declare the interface name with `Interface_`, which itself declares types and synonyms:

   - It creates a forward declaration of `struct Typex (`*interface-name*`)`, along with a pair of type synonyms: `Typex (`*interface-name*`)` is a synonym for a non-modifiable pointer to `const struct Typex (`*interface-name*`)`, whereas `Typex_(`*interface-name*`)` is its modifiable twin (the dereferenced object is non-modifiable).
   - It defines the type `struct` *interface-name* along with a pair of synonyms: *interface-name*`_` is declared a synonym for `struct` *interface-name*, and `interface-name` is its non-modifiable counterpart. The structure definition contains precisely the same member names as `struct Abstract`, with the following members having different types: `base` is declared as a singleton array whose element type is *base-interface*`_` (which is `struct` *base-interface*), and the member `typex` of `_concrete` is of type `Typex_(`*interface-name*`)`.

3. The third step is to declare prototypes for the methods associated with the *interface-name* being declared; the purpose of declaring the type names earlier is to be able to use them as parameters and return types.

4. After declaring the prototypes, `interface_` defines the members of `struct Typex` (*interface-name*), whose forward declaration was done by `Interface_`. Similar to a `class_` declaration, `interface_` also declares procedures for the `Type` methods prefixed with *interface-name*, along with a non-modifiable external array whose name is given by `type_`(*interface-name*): the array is a singleton whose element is a `Type` structure.

EXAMPLE As with classes, interfaces are typically declared in header files that are included in other headers and translation units that require the interface. As our first example in this chapter, we shall declare an interface named `Iterable`. The following interface declaration is available in the header `examples/include/interface/Iterable._`

```
#ifndef ITERABLE__
#define ITERABLE__

#include <c._>

#define Iterable_EXTENDS Abstract,\
 validate, compare, copy, add

Interface_(Iterable)

typedef_  (Iterator, struct Iterator)

prototype_(Bool_, (Iterable, append),
(Iterable *, this), (Void *, data))

prototype_(Size_, (Iterable, count),
(Iterable *, this))

prototype_(Iterator_ *, (Iterable, duplicate),
(Typex (Iterable), typex), (Iterator *, iterator))

prototype_(Void *, (Iterable, get_next),
(Typex (Iterable), typex), (Iterator_*, iterator))

prototype_(Bool_ , (Iterable, has_next),
(Typex (Iterable), typex), (Iterator *, iterator))

prototype_(Iterator_ *, (Iterable, iterator),
(Iterable *, this))

interface_(Iterable, append, count, duplicate, get_next, has_next, iterator)

#endif
```

The macro `Iterable_EXTENDS` sets `Abstract` as the base interface of `Iterable`, and specifies the procedures that are overridden by `Iterable`, namely `validate`, `compare`, `copy`, and `add`; the rest are inherited from the `Abstract` type. `Interface_(Iterable)` declares structures and type synonyms for use in subsequent prototypes.

Every instance of `Iterable` is an abstract data structure that acts as a container for data objects; as their concrete representation is not specified, we need a mechanism to iterate over the elements. `Iterator` is a synonym for an opaque structure that contains the necessary information for iterating over an instance of `Iterable`; the exact details depend on how the data is actually stored, which is decided by the concretizing class.

The `iterator` method is used to obtain an `Iterator` that is initialized to traverse through the instance of `Iterable` referred by the pointer `this`. Each `iterator` maintains an internal state about the current position in the `Iterable`: `has_next` checks whether the `iterator` is exhausted, and if not, then `get_next` can be used to fetch the next element (and also update the internal state of `iterator`). As `Iterator` is an opaque structure, the concretizing class needs to implement the required procedures for operating with iterators; the necessary function pointers for these procedures are supplied via the extended type `typex`. The `duplicate` method is used clone the internal state of an `Iterator` into another object, so that iterating through one of them does not affect the other.

As their names suggest, `append` is used to add a new element to an `Iterable` instance (though not necessarily at the end), and `count` is used to determine the number of elements currently present in an instance of `Iterable`.

NOTE `Iterable` is not required to be a sequence, so a concretizing class need not preserve the insertion order.

8.3.2 Definition

Unlike declarations typically placed in header files, each interface must be defined in exactly one translation unit.

Syntax

`define_` (*interface-name*)
`define_1_` (*interface-name*)

Constraints

An object-like macro named as *interface-name*`_EXTENDS` shall remain defined, and for each *ancestor* type up to the `Abstract` type, a similarly named macro *ancestor*`_EXTENDS` shall also remain defined.

Semantics

`define_(`*interface-name*`)` is equivalent to `define_1_(`*interface-name*`)`, which defines a non-modifiable external array named `type_(`*interface-name*`)`: this array is a singleton whose sole element is a `Type` structure, containing function pointers for the basic `Type` procedures associated with *interface-name* (either inherited or overridden).

EXAMPLE The following interface definition is available in the file `examples/compile/interface/Iterable.c_`

```
#include "Iterable._"

 define_ (Iterable)
```

8.3.3 Type extension

Every interface declaration creates a structure with the tag `Typex (`*interface-name*`)`, which is forward declared by `Interface_(`*interface-name*`)` and structure members are defined by `interface_(`*interface-name*`, `*members-list*`)`. `Typex` is a portmanteau of “`Type` e**x**tension”, as this structure extends the `Type` structure with additional members.

The first member is named as `base`, which is a singleton array with element type `struct Typex (`*base-interface*`)`. For each *member-name* in *members-list*, the corresponding structure member is declared with the same name, as pointer to a function having the same type as the procedure `solver_(`*interface-name*`, `*member-name*`)` (note that `prototype_` declares the procedure’s type only); members are declared in the same sequence as in *members-list*.

The `Abstract` type also creates a structure with the tag `Typex (Abstract)`; it is a slight misnomer to call it a `Type` “extension”, since it contains a single member `base` declared as a singleton array with element `struct Type`.

8.3.3.1 `is_typex`

The function `is_typex` is used to check whether a pointer refers to a valid extended `Type` or not.

Declaration

`Bool_ is_typex(Void *typex)`

Description

The function `is_typex` returns `true` if and only if `is_type(typex)` is `true`, `((Type) typex)->self` is not the same pointer as `typex`, and `is_type(((Type) typex)->self)` is also `true`; otherwise the return value is `false`.

8.4 Procedures and methods

We start with overriding implementations of basic procedures, followed by other methods of `Iterable` interface.

8.4.1 validate

examples/compile/interface/Iterable/validate.c_

```
#include "Iterable._"

procedure_(Bool_, (Iterable, validate),
(let Ptr (Iterable), this))
    Var    typex = concrete_(this)->typex;
    guard_(typex->append
        && typex->count
        && typex->duplicate
        && typex->get_next
        && typex->has_next
        && typex->iterator, FALSE_())
    Var concrete = this->concrete;
    Var_  count_ = typex->count(concrete);
    Var_   more_ = TRUE_();
{   Var iterator = typex->iterator(concrete);
    post_(iterator != NULL);
    Var has_next = typex->has_next;
    Var get_next = typex->get_next;
    for (; (more_ = has_next(typex, iterator)) &&  count_; count_--)
        get_next(typex, iterator);
    free(iterator);
}   return !(more_ ||  count_);
end
```

If any member of the extend `Type` structure is a null pointer, the instance is considered invalid and zero is returned. Otherwise the instance is considered valid iff the number of elements reported by `count` is found to be correct.

8.4.2 compare

examples/compile/interface/Iterable/compare.c_

```
#include "Iterable._"

procedure_(LLong_, (Iterable, compare),
(let Ptr (Iterable), this),
(let Ptr (Iterable), that))
    Var this_typex = concrete_(this)->typex;
    Var that_typex = concrete_(that)->typex;
    Var this_count = this_typex->count(this->concrete);
    Var that_count = that_typex->count(that->concrete);
```

```
    guard_(this_count == that_count, this_count - that_count)
    guard_(this_count, 0)
    Var this_array = new__(Void *[this_count]);
    post_(this_array != NULL);
{   Var   iterator = this_typex->iterator(this->concrete);
    post_(iterator != NULL);
    Var get_next = this_typex->get_next;
    for (Var_ i_ = this_count; i_--;)
        (*this_array)[i_] = get_next(this_typex, iterator);
    free (iterator);
}{  Var   iterator = that_typex->iterator(that->concrete);
    post_(iterator != NULL);
    Var get_next = that_typex->get_next;
    for_(Var_ count_ = that_count; count_--;)
        Var next = get_next(that_typex, iterator);
        Var_ i_ = count_;
        do  if (next == (*this_array)[i_])
        {   (*this_array)[i_] = (*this_array)[count_];
            break;
        }   while (i_--);
        guard_(i_ + 1, 1)
    end
    free (iterator);
}   return 0;
end
```

If both iterables do not contain equal number of elements, then the one with lesser elements is considered smaller. If both of them contain precisely the same elements, then they are considered equal, and the return value is zero. Otherwise they are considered incomparable, and the return value is one.

8.4.3 copy

examples/compile/interface/Iterable/copy.c_

```
#include  "Iterable._"

procedure_(Iterable_*, (Iterable, copy),
(let  Ptr (Iterable_), this),
(let  Ptr (Iterable ), that))
    Var typex = concrete_(that)->typex;
    Var iterator = typex->iterator(that->concrete);
    guard_(iterator, NULL)
    Var copy = validate_(this) &&  ! concrete_(this)->typex->count(this->concrete)
    ? this : (Iterable_ *) init__(type_(Iterable), typex);
    guard_(copy, (free(iterator), NULL))
    Var   append = concrete_(copy)->typex->append;
    Var concrete = copy->concrete_;
    Var get_next = typex->get_next;
```

```
    for_(Var_ count_ = typex->count(that->concrete); count_--;)
        continue_(append(concrete, get_next(typex, iterator)))
        free_(concrete);
        if (copy == this) copy->concrete = NULL;
        else free(copy);
        free (iterator);
        return NULL;
    end
    free(iterator);
    return copy;
end
```

If `this` is a valid empty iterable, then it is used to store the copied elements; otherwise a new instance is allocated. The loop iterates over the source iterable `that` and appends each element to the concrete instance of the destination.

8.4.4 add

examples/compile/interface/Iterable/add.c_

```
#include "Iterable._"

procedure_(Iterable_*, (Iterable, add),
(let Ptr (Iterable_), sum),
(let Ptr (Iterable ), augend),
(let Ptr (Iterable ), addend))
    Var aug = concrete_(augend)->typex->iterator(augend->concrete);
    guard_(aug, NULL)
    Var add = concrete_(addend)->typex->iterator(addend->concrete);
    guard_(add, (free(aug), NULL))
    Var cat = validate_(sum)
    &&  sum->concrete != augend->concrete
    &&  sum->concrete != addend->concrete
    ?   sum : (Iterable_ *) init__(type_(Iterable), concrete_(augend)->type);
    guard_(cat, (free(aug), free(add), NULL))
    Var   append = concrete_(cat)->typex->append;
    Var concrete = cat->concrete_;
{   Var    typex = concrete_(augend)->typex;
    Var get_next = typex->get_next;
    for_(Var_ count_ = typex->count(augend->concrete); count_--;)
        continue_(append(concrete, get_next(typex, aug)))
        free_(concrete);
        if (cat == sum) cat->concrete = NULL;
        else free(cat);
        free(aug);
        free(add);
        return NULL;
    end
}   free(aug);
```

```
{   Var typex  = concrete_(addend)->typex;
    Var get_next = typex->get_next;
    for_(Var_ count_ = typex->count(addend->concrete); count_--;)
        continue_(append(concrete, get_next(typex, add)))
        free_(concrete);
        if (cat == sum) cat->concrete = NULL;
        else free(cat);
        free(add);
        return NULL;
    end
}   free(add);
    return cat;
end
```

If `sum` is valid and its concrete instance is not the same as concrete instance of `augend` or `addend`, then `sum` is used as the destination. Each element of `augend` is appended to the destination, followed by the elements of `addend`.

8.4.5 append

The `append` method is used to insert an additional element in an existing instance of `Iterable`.

8.4.5.1 Protocol

examples/include/interface/Iterable/append._

```
#ifndef   ITERABLE__APPEND__
#define   ITERABLE__APPEND__

#include "Iterable._"

private
protocol_(Bool_, (Iterable, append),
(let Ptr (Iterable), this),
(let Ptr (Void), data))
    pre_ (validate_(Iterable, this));
    Var count   = concrete_(this)->typex->count;
    Var priori  = count(this->concrete);
    Var success = solver_(Iterable, append)(this, data);
    post_(validate_(Iterable, this));
    post_((!success implies  count(this->concrete) == priori));
    post_ ((success implies  count(this->concrete) == priori+1));
    return  success;
end

#endif
```

Pre-conditions

- `this` must be a valid instance of `Iterable` before appending.

Post-conditions

- `this` must remain a valid `Iterable` instance regardless of whether `data` was appended or not.
- If the procedure returns `false`, then the count of elements must remain unchanged.
- If the procedure returns `true` , then the count of elements must be one more than the previous count.

8.4.5.2 Procedure

`examples/compile/interface/Iterable/append.c_`

```
#include "Iterable/append._"

procedure_(Bool_, (Iterable, append),
(let Ptr (Iterable), this),
(let Ptr (Void), data))
    return concrete_(this)->typex->append(this->concrete, data);
end
```

`append` procedure provided by the concretizing class is called with the concrete instance and `data` as arguments.

8.4.6 iterator

The `iterator` method is used to obtain (pointer to) a new `Iterator_` for traversing through an `Iterable` instance.

8.4.6.1 Protocol

`examples/include/interface/Iterable/iterator._`

```
#ifndef   ITERABLE__ITERATOR__
#define   ITERABLE__ITERATOR__

#include "Iterable._"

private
protocol_(Iterator_ *, (Iterable, iterator),
(let Ptr (Iterable), this))
    pre_(validate_(Iterable, this));
    return solver_(Iterable, iterator)(this);
end

#endif
```

Pre-conditions

- `this` must be valid instance of `Iterable`.

8.4.6.2 Procedure

examples/compile/interface/Iterable/iterator.c_

```
#include "Iterable/iterator._"

procedure_(Iterator_ *, (Iterable, iterator),
(let  Ptr (Iterable), this))
    return concrete_(this)->typex->iterator(this->concrete);
end
```

iterator procedure implemented by the concretizing class is called with the concrete instance as argument.

8.4.7 has_next

The has_next method is used to check if there is any element that has not been traversed using the given Iterator.

8.4.7.1 Protocol

examples/include/interface/Iterable/has_next._

```
#ifndef   ITERABLE__HAS_NEXT__
#define   ITERABLE__HAS_NEXT__

#include "Iterable._"

private
protocol_(Bool_, (Iterable, has_next),
(let Typex (Iterable), typex),
(let Ptr (Iterator), iterator))
    pre_(is_typex(typex));
    pre_(typex->has_next != NULL);
    pre_(iterator != NULL);
    return solver_(Iterable, has_next)(typex, iterator);
end

#endif
```

Pre-conditions

- typex must be a valid extended Type.
- The member has_next of typex must not be a null pointer.
- iterator must not be a null pointer.

8.4.7.2 Procedure

examples/compile/interface/Iterable/has_next.c_

```
#include  "Iterable/has_next._"

procedure_(Bool_, (Iterable, has_next),
(let Typex(Iterable), typex),
(let Ptr  (Iterator), iterator))
    return typex->has_next(typex, iterator);
end
```

`has_next` procedure implemented by the concretizing class is called with `typex` and `iterator` as arguments.

8.4.8 get_next

The `get_next` method is used to fetch an element that has not been traversed yet (if any) using the given `Iterator_`.

8.4.8.1 Protocol

examples/include/interface/Iterable/get_next._

```
#ifndef   ITERABLE__GET_NEXT__
#define   ITERABLE__GET_NEXT__

#include "Iterable._"

private
protocol_(Void *, (Iterable, get_next),
(let Typex(Iterable), typex),
(let Ptr (Iterator_), iterator))
    pre_(is_typex(typex));
    pre_(typex->has_next != NULL);
    pre_(typex->get_next != NULL);
    pre_(iterator != NULL);
    Var hasnext = typex->has_next(typex, iterator);
    Var    next = solver_(Iterable, get_next)(typex, iterator);
    post_((next implies  hasnext));
    return next;
end

#endif
```

Pre-conditions

- `typex` must be a valid extended `Type`.
- The member `has_next` of `typex` must not be a null pointer.
- The member `get_next` of `typex` must not be a null pointer.
- `iterator` must not be a null pointer.

Post-conditions

- The return value can be non-null only if `has_next` returned `true` before calling the procedure.

8.4.8.2 Procedure

examples/compile/interface/Iterable/get_next.c_

```
#include  "Iterable/get_next._"

procedure_(Void *, (Iterable, get_next),
(let Typex(Iterable ), typex),
(let Ptr  (Iterator_), iterator))
    return typex->get_next(typex, iterator);
end
```

`get_next` procedure implemented by the concretizing class is called with `typex` and `iterator` as arguments.

8.4.9 duplicate

The `duplicate` method is used to obtain a new `Iterator_` whose internal state is identical to an existing `Iterator`.

8.4.9.1 Protocol

examples/include/interface/Iterable/duplicate._

```
#ifndef   ITERABLE__DUPLICATE__
#define   ITERABLE__DUPLICATE__

#include "Iterable._"

private
protocol_(Iterator_ *, (Iterable, duplicate),
(let Typex (Iterable), typex),
(let Ptr (Iterator), iterator))
    pre_(is_typex(typex));
    pre_(typex->duplicate != NULL);
    pre_(iterator != NULL);
    return solver_(Iterable, duplicate)(typex, iterator);
end

#endif
```

Pre-conditions

- `typex` must be a valid extended `Type`.
- The member `has_next` of `typex` must not be a null pointer.
- `iterator` must not be a null pointer.

8.4.9.2 Procedure

examples/compile/interface/Iterable/duplicate.c_

```
#include  "Iterable/duplicate._"

procedure_(Iterator_ *, (Iterable, duplicate),
(let Typex(Iterable), typex),
(let Ptr  (Iterator), iterator))
    return typex->duplicate(typex, iterator);
end
```

`duplicate` procedure implemented by the concretizing class is called with `typex` and `iterator` as arguments.

8.4.10 count

The `count` method is used to determine the number of elements currently present in a given instance of `Iterable`.

8.4.10.1 Protocol

examples/include/interface/Iterable/count._

```
#ifndef   ITERABLE__COUNT__
#define   ITERABLE__COUNT__

#include "Iterable._"

private
protocol_(Size_, (Iterable, count),
(let Ptr (Iterable), this))
    pre_ (validate_(Iterable, this));
    return solver_(Iterable, count)(this);
end

#endif
```

Pre-conditions

- `this` must be a valid instance of `Iterable`.

8.4.10.2 Procedure

examples/compile/interface/Iterable/count.c_

```
#include "Iterable/count._"

procedure_(Size_, (Iterable, count),
(let Ptr (Iterable), this))
    return concrete_(this)->typex->count(this->concrete);
end
```

`iterator` procedure implemented by the concretizing class is called with the concrete instance as argument.

NOTE While the required functionality can be directly implemented by counting how many times `get_next` can be called on an `Iterator` before `has_next` returns `false`, such an approach would have linear time complexity.

On the other hand, concrete implementations can maintain an internal counter for the number of elements in an instance, which would be incremented by the concrete implementation of `append` each time an additional element is inserted (needless to say, the internal counter would be initialized to zero when a new `Iterable` is instantiated). Relegating the responsibility of counting to the concretizing class provides it an opportunity to leverage knowledge about internal organization of the underlying concrete object, and return the count of elements in constant time.

8.5 Concretization

Concretization means defining a structure for materializing an abstract data type described by an interface, and providing functions that can operate on instances of the concrete structure. This structure is provided by a class declaration, and for each member in the `Typex` structure declared by an interface, the class declares a prototype for a corresponding method whose procedure implements the necessary functionality required by the interface.

A concretizing class is said to "implement" an interface: an interface can be implemented by multiple classes, and a single class can implement multiple classes. The implementation relationship between a class and an interface is established with the help of object-like macros, which are required by declarations as well as definitions.

8.5.1 Declaration

Syntax

`# define` *class-name*`_EXTENDS` *base-class* [`,` *override-list*]
`# define` *class-name*`_IMPLEMENTS_`*interface-list* *base-implementation* `,` *methods-list*
`class_` `(` *class-name* [`implements` *interface-list*] `)` *member-declarations* `fin`
`class_0_` `(` *class-name* `implements` *interface-list* `)` *member-declarations* `fin`
`prototype_` `(` [*return-type* `,`] `(` *class-name* `,` *member-name* `)` [`,` *parameter-tuples*] `)`

Constraints

base-implementation shall be name of the class that implements the base type of *interface-name*, as specified by another macro named as *interface-name*`_EXTENDS`; as a special constraint, if *interface-name* directly extends the `Abstract` type, then *base-implementation* shall be specified as the word `SELF`. If the macro *interface-name*`_EXTENDS` specifies *base-interface* as the base type and *base-interface* is not `Abstract`, then an object-like macro named *base-implementation*`_IMPLEMENTS_`*base-interface* shall be defined in the same form as described in the syntax.

methods-list shall be a comma-separated list of member names declared in `struct Typex (`*interface-name*`)`, and for each such member a corresponding prototype shall be declared, prefixed with *class-name*. Each element in *methods-list* shall be a *member-name* corresponding to a function pointer in `struct Typex (`*interface-name*`)`, or a parenthesized pair of the form `(` *typex-member* `,` *class-method* `)`, where *typex-member* is a member name declared in `struct Typex (`*interface-name*`)`, and *class-method* is the unprefixed name declared in prototype of a class method.

Other constraints are precisely the same as documented for class declarations in §7.8.1 of the previous chapter.

NOTE *member-name*, `(`*member-name*`)`, and `(`*member-name*`,` *member-name*`)` are equivalent in *members-list*.

Semantics

`implements` is defined as an object-like macro that expands to a single comma. When the argument sequence of `class_` expands to multiple arguments, it invokes `class_0_`: the first argument is considered as name of a concretizing class, and subsequent arguments are the names of interfaces. For each *interface-name* specified in *interface-list*, `class_0_` declares a non-modifiable external array named `typex_(`*interface-name*`,` *class-name*`)`, whose element type is `const struct Typex (`*interface-name*`)`. Rest of the semantics are identical to that of `class_1_` (see §7.8.1).

NOTE The macros named as *class-name*`_IMPLEMENTS_`*interface-list* are required only for class definitions; they are placed in header files because another macro *base-implementation*`_IMPLEMENTS_`*base-interface* is required by the class definition (recursively for each ancestor); the latter is made available in the header of *base-implementation*.

EXAMPLE The `Chain` class implementing a linked list is a good candidate for concretizing `Iterable` interface. The following class declaration is available in the header file `examples/include/class/Chain._`

```
#ifndef CHAIN__
#define CHAIN__

#include "Iterable._"

#define Chain_EXTENDS Object,\
 validate, init, free, compare, copy, add

#define Chain_IMPLEMENTS_Iterable SELF,\
 append, count, duplicate, get_next, has_next, iterator

typedef_(Node, struct Node
{   Void  *data;
    struct Node *next;
})

class_ (Chain implements Iterable)
    Node_ *head;
    Size_  length;
    Node_ *tail;
fin

prototype_(Bool_ , (Chain, append),
(Chain_ *, this) , (Void *, data))

prototype_(Size_ , (Chain, count),
(Chain  *, this))

prototype_(Node_*, (Chain, iterator),
(Chain  *, this))

prototype_(Node_*, (Chain, duplicate),
(Typex (Iterable), typex), (Node  *, node))

prototype_(Void *, (Chain, get_next),
(Typex (Iterable), typex), (Node_ *, node))

prototype_(Bool_ , (Chain, has_next),
(Typex (Iterable), typex), (Node  *, node))

#endif
```

We are already familiar with most of this code, as it is almost entirely similar to the version in previous chapter.

New additions are inclusion of the header `"Iterable._"`, along with the macro `Chain_IMPLEMENTS_Iterable` that specifies the members of `struct Typex (Iterable)`, and for each member, we have also declared a prototype for the class method whose procedure provides the concrete implementation. `SELF` is used as *base-implementation* because the `Iterable` interface directly extends `Abstract`, for which we do not have a separate concretizing class.

Recall that the header `"Iterable._"` declares `Iterator` as a synonym for the incomplete type `struct Iterator`, whose pointer type is used in the `Iterable` prototypes for `iterator`, `has_next`, `get_next`, and `duplicate`. The opaque type `Iterator` for traversing through an `Iterable` is concretized by `Node` for traversing through a `Chain`.

8.5.2 Definition

For a concretizing class *class-name*, the translation unit that defines the array `type_(`*class-name*`)` also defines `typex_(`*interface-name*`,` *class-name*`)` for each *interface-name* specified in the class definition; it should be precisely the same set of instances that are specified in the class declaration (sequence of interface names does not matter).

Syntax

`define_` (*class-name* [`implements` *interface-list*])
`define_0_` (*class-name* `implements` *interface-list*)
`define_1_` (*class-name*)

Constraints

The constraints are precisely the same as documented for class definitions in §7.8.2 of the previous chapter.

Semantics

The semantics are precisely the same as documented for class definitions in §7.8.2 (`implements` expands to a comma). In particular, for each *interface-name* in *interface-list*, `define_0_` defines the non-modifiable external array named as `typex_(`*interface-name*`,` *class-name*`)` that is also declared (but not defined) by the class declaration.

Additionally, a single definition line of the form `define_(`*class-name*`,` *interface-A*`,` *interface-B*`)` can be split as `define_(`*class-name*`,` *interface-A*`)` `define_(`*class-name*`,` *interface-B*`)`, since both forms are equivalent.

`define_(`*class-name*`,` *interface-name*`)` initializes members of the sole element of the array `typex_(`*interface-name*, *class-name*`)`, which is of type `struct Typex (`*interface-name*`)`. This initialization requires an object-like macro named as *class-name*`_IMPLEMENTS_`*interface-name*, whose replacement text specifies name of the *base-implementation* class that concretizes the base interface of *interface-name*, and *members-list* specifies the member names of `struct Typex (`*interface-name*`)` to be initialized. For each expanded element in *members-list*:

- If the element is an identifier of the form *member-name* (optionally parenthesized), then the structure member `.`*member-name* is initialized with function pointer for the procedure `solver_(`*class-name*`,` *member-name*`)`.
- Otherwise it is a parenthesized pair of identifiers of the form `(`*typex-member*`,` *class-method*`)`, which initializes the member `.`*typex-member* with function pointer for the procedure `solver_(`*class-name*`,` *class-method*`)`.

The characterizing distinction between `type_(`*class-name*`)` and `typex_(`*interface-name*`,` *class-name*`)` is that for the latter, the member `self` points to `type_(`*class-name*`)`, whereas `type_(`*class-name*`)->self` points to itself.

NOTE Required procedure names are internally declared by the `define_` family, using the function types specified by `prototype_` declarations. The parenthesized form `(`*typex-member*`,` *class-method*`)` is useful when the implementing class method is declared with a different name than name of the interface method being implemented. This situation is inevitable when a single class concretizes multiple interfaces and at least one member name is common between their respective `Typex` structures, but declared with incompatible function pointer types.

EXAMPLE The following definitions are available in the source file `examples/compile/class/Chain.c_`

```
#include "Chain._"

 define_ (Chain)

 define_ (Chain implements Iterable)
```

8.5.3 Abstraction

The `abstract_` family can be used to wrap a class instance within a new instance of an interface that is concretized by the class, or by one of its ancestor classes; this wrapped class instance can be used with methods of the interface.

NOTE Since all interfaces inherit from the `Abstract` type (either directly or indirectly), the concretizing class of any interface must have an ancestor class that concretizes a direct descendant of `Abstract` (it can be the same class itself). Therefore, the instance of any class can be wrapped as the `concrete` member of an `Abstract` instance.

Syntax

`abstract_` ([*interface-name*=`Abstract` ,] *expression*)
`abstract_1_` (*expression*)
`abstract_2_` (*interface-name* , *expression*)

Constraints

expression shall be pointer to an object type.

Semantics

`abstract_` invokes `abstract_`n`_` if the expanded argument sequence contains n arguments.

`abstract_1_(`*expression*`)` is equivalent to `abstract_2_(Abstract,` *expression*`)`.

`abstract_2_` creates an lvalue of type `struct` *interface-name*, whose `type` member is initialized with `type_(` *interface-name*`)`, and `concrete` member is initialized with *expression*. This lvalue has automatic storage duration, and its lifetime is limited to the nearest enclosing block where it is created.

NOTE If *expression* points to an instance of class *class-name*, then the wrapped instance can be correctly used with methods of *interface-name* only if `(`*expression*`)->type` points to `typex_(`*interface-name*`,` *class-name*`)`, which provides the necessary concrete procedures (as function pointers); otherwise the behavior is undefined.

8.5.4 Methods

For each class method that provides an implementation for an interface method, the concrete protocol can leverage the abstract protocol which already establishes pre-conditions and post-conditions. This promotes code reusability: if an interface is concretized by multiple classes, then each of their protocols need not re-implement the same logic.

In this subsection, we have only described the implementation of `append` method for the `Chain` class; rest of the concretizing methods are implemented in a similar way, and their source codes can be found in appendix A.

NOTE A concrete protocol can also specify its own set of pre-conditions and post-conditions specific to the class structure, in addition to those that are already imposed by the corresponding abstract protocol of the interface.

8.5.4.1 Protocol

`examples/include/class/Chain/append._`

```
#ifndef   CHAIN__APPEND__
#define   CHAIN__APPEND__

#include "Chain._"

protocol_( Bool_ , (Chain, append),
(let Ptr (Chain_), this),
(let Ptr ( Void ), data))
    pre_(validate_(Chain, this));
    Var   type = this->type;
```

```
    this->type = (Type)typex_(Iterable, Chain);
    Var success= verifier_(Iterable, append)
    (_site, abstract_(Iterable, this), data);
    this->type = type;
    post_((success implies  data == this->tail->data));
    return success;
end

#endif
```

Before calling the append protocol of Iterable interface, the concrete type is set to typex_(Iterable, Chain), which contains the function pointer for the append procedure of Chain class (and other concretizing procedures as well). The concrete instance is wrapped into an instance of Iterable using abstract_, which is passed to the append protocol of Iterable. The first argument is given as the parameter _site instead of SITE; the latter would give the source code coordinates of the current call site, whereas _site ensures that the original call site of the current invocation is used in diagnostic messages for any pre-condition violation. Calling the append protocol of Iterable ensures that the concrete instance is checked against its pre-conditions and post-conditions.

In addition to that, the concrete protocol also ensures that the following conditions are satisfied:

- *Pre-condition*: this must be a valid instance of Chain class, or one of its sub-classes (Liskov substitution).
- *Post-condition*: If data was successfully appended to this, then it must be at the tail node.

Before returning the success status, this->type is restored to its former value that was stored in a local variable.

8.5.4.2 Procedure

examples/compile/class/Chain/append.c_

```
#include "Chain/append._"

procedure_(Bool_ , (Chain, append),
(let Ptr (Chain_), this),
(let Ptr ( Void ), data))
    Var node = new__(Node);
    guard_(node, FALSE_())
    node->data = data;
    node->next = NULL;
    if  (this->length++) this->tail->next = node;
    else this->head = node;
    this->tail = node;
    return TRUE_();
end
```

If a new node cannot be allocated, then zero is returned. Otherwise the members data and next are initialized to the parameter data and null pointer; the latter is done because the new node will be set as the tail node. If the existing value of this->length prior to incrementing is non-zero, then the chain is non-empty, and the new node is linked to the next member of the current tail node; otherwise the chain is empty and the new node is set as the head node. In any case, the new node is set as the tail node before returning 1, which indicates success.

8.6 Dependency inversion

One of the blessings of inheritance in object-oriented programming is that it facilitates code reuse. We have already seen how the pre-conditions and post-conditions established by interface protocols can be utilized by concretizing classes without re-writing them. This workflow is illustrated below, where each arrow denotes a function call.

$\boxed{\text{Class protocol}} \longrightarrow \boxed{\text{Interface protocol}} \longrightarrow \boxed{\text{Interface procedure}} \longrightarrow \boxed{\text{Class procedure}}$

The above diagram leads to another interpretation of interfaces: an interface can act as an additional layer of abstraction between a class protocol and its corresponding procedure. If a class method provides the concrete implementation of an interface method, then the class protocol can indirectly invoke its procedure via an interface, which validates pre-conditions on arguments and verifies post-conditions on return value (and possible side effects).

So far we have seen that each interface requires some concretizing class, which can be summarily stated as "abstract depends on concrete": this is the conventional dependency. To further leverage the benefits of interfaces, we can also write class procedures that depend on an interface procedure, which inverts the dependency relationship.

For example, we have seen in §8.4.2 how the `compare` procedure of `Iterable` can be implemented by comparing the count of elements in each iterable, and if found to be equal, then checking if both of them contain precisely the same set of elements. Iteration and appending require knowledge of concrete representation, and must be facilitated by the class; once these are available, other operations such as `compare` and `copy` can be implemented using them.

This approach of comparing two iterables is abstract in nature, as it does not make any assumptions about how the data is actually stored. Abstract designs are highly reusable: each concretizing class need not implement its own custom logic for comparing instances, but can simply use the generic approach implemented by the interface. This is how the `compare` procedure of `Chain` class is implemented in `examples/compile/class/Chain/compare.c_`

```
#include "Chain._"

procedure_(LLong_, (Chain, compare),
(let Ptr (Chain) , this),
(let Ptr (Chain) , that))
    Chain_ _this[1] = {*this};
    Chain_ _that[1] = {*that};
    _this->type = (Type)typex_(Iterable, Chain);
    _that->type = (Type)typex_(Iterable, Chain);
    return solver_(Iterable, compare)
        (abstract_(Iterable,_this),
         abstract_(Iterable,_that));
end
```

The high-level workflow shown below is quite opposite to the conventional one we saw earlier.

$\boxed{\text{Class procedure}} \longrightarrow \boxed{\text{Interface procedure}}$

We refer to this as "dependency inversion", since the concretizing class depends on the interface for providing certain functionalities. The coordination and interdependence between an interface and its concretizing class is an important aspect in object-oriented design, and the technique of dependency inversion is also used to implement several other procedures associated with `Chain`, `List`, and `Vector` classes, for which the interface itself implements abstract algorithms not tied to any particular representation, by using the primitives of iteration and appending.

NOTE A basic tenet of abstract design is that instead of operating directly on data, we implement representation-agnostic higher-order functions that work with other lower-level functions which are "closer" to the concrete data.

8.7 Inheritance

Inheritance relationship between two interfaces can be established in the same way as done for classes; we have already seen an example of this in the `Iterable` interface, which extends the `Abstract` type. Now we shall discuss an example of multi-level inheritance with interfaces, by creating another interface `Collection` that extends `Iterable`.

8.7.1 Declaration

The following declaration is available in the header file `examples/include/interface/Collection._`

```
#ifndef COLLECTION__
#define COLLECTION__

#include "Iterable._"

#define Collection_EXTENDS Iterable,\
 validate, copy, read, write,\
 parse, text, decode, encode, add

Interface_(Collection)

prototype_(Bool_, (Collection, append),
(Collection *, this), (Void *, data))

prototype_(Type_, (Collection, species),
(Collection *, this), (Type  , species))

interface_(Collection, append, species)

#endif
```

`Collection` introduces an additional method `species`, whose purpose is to associate a `Type` with every instance of `Collection`. Each data pointer stored in a collection must refer to a valid instance of the `Type` associated with that collection; this is also a validation condition in the overriding `validate` procedure of `Collection`.

The benefit of associating a type with every instance of `Collection` is that it allows us to override most of the basic `Type` procedures inherited from the `Iterable` interface, which in turn inherits them from the `Abstract` type. For example, the `write` procedure iterates over a `Collection` instance, and since each element has its own `type` member, the `write` procedure of that `type` can be invoked to write the instance data on a given output stream. Similar abstract designs have been adopted for implementing most of the other `Type` methods, and concretizing classes can use the dependency inversion technique to utilize the interface procedures without any non-trivial logic.

8.7.2 Definition and concretization

Defining the `Collection` interface is quite trivial, as done in the file `examples/compile/interface/Collection.c_`

```
#include "Collection._"

 define_ (Collection)
```

In our examples, the `Collection` interface is concretized independently by `List` and `Vector` classes; the former provides its own definitions with the following code available in the source file `examples/compile/class/List.c_`

```
#include "List._"

 define_ (List)

 define_ (List implements Collection)
```

8.7.3 Methods

8.7.3.1 Type relaxation

To store instances of any object-oriented type, the argument for `species` can be specified as `type_(Object)`.
Protocol

`examples/include/interface/Collection/species._`

```
#ifndef   COLLECTION__SPECIES__
#define   COLLECTION__SPECIES__

#include "Collection._"

private
protocol_(Type_, (Collection, species),
(let Ptr (Collection), this),
(let Type, species))
    pre_(validate_(Collection, this));
    pre_((species implies is_type(species)));
    Var concrete = this->concrete;
    Var    typex = concrete_(this)->typex;
    Var   priori = typex->species(concrete, NULL);
    Var   update = solver_(Collection, species)(this, species);
    post_(is_type(update));
    post_(update == typex->species(concrete, NULL));
    post_((!species implies  update == priori));
    guard_( species, update)
    Var count = typex->base->count(concrete);
    post_((!count implies  update == species));
    post_ ((count implies  update == super(priori, species)));
    return update;
end

#endif
```

The pre-conditions ensure `this` must be a valid instance of `Collection`, and if `species` is not null, then it must be a valid `Type`. The post-conditions ensure that the updated type is also valid, and if the parameter `species` was null, then the collection's type must remain same as before. For an empty collection, a non-null `species` parameter must be directly used to update the collection's type. Otherwise the collection is non-empty and its updated type must be nearest common ancestor of the prior type and `species` parameter; rationale of using the super type is to avoid a non-empty collection from getting invalidated on account of some element not being an instance of `species`.

Procedure

examples/compile/interface/Collection/species.c_

```
#include "Collection/species._"

procedure_(Type_, (Collection, species),
(let Ptr (Collection), this),
(let Type, species))
    return concrete_(this)->typex->species(this->concrete, species);
end
```

8.7.3.2 Appending

A new data can be appended to a collection only if it is a valid instance of the collection's type (or its sub-type).

In order to impose this requirement as a pre-condition, `Collection` overrides `append` method of its base interface `Iterable`; implementation of the procedure is trivial and identical to the code for `append` procedure of `Iterable`.

Protocol

examples/include/interface/Collection/append._

```
#ifndef   COLLECTION__APPEND__
#define   COLLECTION__APPEND__

#include "Collection._"

private
protocol_(Bool_, (Collection, append),
(let Ptr (Collection), this),
(let Ptr (Void), data))
    pre_(validate_(Collection, this));
    Var  species = concrete_(this)->typex->species(this->concrete, NULL);
    pre_(validate (species, data));
    return verifier_(Iterable, append)(_site, this->base, data);
end

#endif
```

Procedure

examples/compile/interface/Collection/append.c_

```
#include "Collection/append._"

procedure_(Bool_, (Collection, append),
(let Ptr (Collection), this),
(let Ptr (Void), data))
    return concrete_(this)->typex->append(this->concrete, data);
end
```

8.7.4 Necessary condition

For Liskov substitution to work correctly with instances of interfaces, a fundamental necessity is that the concrete lineage must be a refinement of the abstract lineage. This requirement can be formalized as stated below:

If *interface-A* is extended by *interface-B*, and *interface-B* is concretized by *class-C*, then *interface-A* must also be concretized by *class-C* or one of its ancestors.

If an instance of *interface-B* is substituted as an instance of its base *interface-A*, the `concrete` member would be of *class-C* (or a derived class). However, any code operating with *interface-A* would expect `concrete` instance of a class that implements *interface-A*, which works fine if such a class happens to be an ancestor of *class-C*, or the same class itself. This is because in the concrete lineage, any instance of *class-C* (or sub-class) is also valid for all of its ancestors (due to `base` being the first member), and is thus suitable to be used with an instance of *interface-A*.

EXAMPLE `Collection` interface is concretized by `List` class, and its base interface `Iterable` is concretized by `Chain` (which is the base class of `List`). Since an instance of `List` can be used as an instance of `Chain`, an instance of `Collection` whose `concrete` member points to an instance of `List` can be used as an instance of `Iterable`.

8.8 More examples

As commented earlier in this chapter, an interface can be concretized by multiple classes, and a class can concretize multiple interfaces. In this section we present the `Vector` class, which concretizes both `Iterable` and `Collection`.

8.8.1 `Vector` class

The core concepts on interfaces have already been demonstrated with several examples, so for the sake of brevity, we shall discuss only the declaration and definition of the `Vector` class, along with its concretization of `Iterable` and `Collection` interface; interested readers can find the protocols and procedures collectively in appendix A.

8.8.1.1 Declaration

`examples/include/class/Vector._`

```
#ifndef VECTOR__
#define VECTOR__

#include "Collection._"

#define Vector_EXTENDS Object,\
 validate, init, free, compare, copy, read,\
 write, parse, text, decode, encode, add

#define Vector_IMPLEMENTS_Iterable SELF,\
 append, count, duplicate, get_next, has_next, (iterator, cursor)

#define Vector_IMPLEMENTS_Collection Vector,\
 append, species
```

```
class_ (Vector implements Iterable, Collection)
    Void  **array;
    Size_ capacity, count;
    Type_ species;
fin

typedef_(Cursor, struct Cursor
{   Vector *vector;
    Size_   index ;
})

prototype_(Bool_ , (Vector, append),
(Vector_ *, this), (Void *, data))

prototype_(Size_ , (Vector, count),
(Vector  *, this))

prototype_(Cursor_ *, (Vector, cursor),
(Vector  *, this))

prototype_(Cursor_ *, (Vector, duplicate),
(Typex (Iterable), typex), (Cursor  *, cursor))

prototype_(Void *, (Vector, get_next),
(Typex (Iterable), typex), (Cursor_ *, cursor))

prototype_(Bool_ , (Vector, has_next),
(Typex (Iterable), typex), (Cursor  *, cursor))

prototype_(Type_ , (Vector, species),
(Vector_ *, this), (Type  , species))

#endif
```

Unlike `Chain` and `List` that implement a linear linked list, `Vector` uses an array of pointers. The class structure contains four members: `array` points to the base element of an array of `Void` pointers; `capacity` stores the maximum number of elements that can be stored with an existing allocation of `array`, while `count` maintains the number of elements already stored in the array (from index zero); `species` is required for the `Collection` interface.

`Cursor` structure is used as an iterator, and the class method `cursor` implements the interface method `iterator`; the latter is established by the pair `(iterator, cursor)` in the replacement text of `Vector_IMPLEMENTS_Iterable`.

8.8.1.2 Definition and concretization

examples/compile/class/Vector.c_

```
#include "Vector._"

 define_ (Vector)

 define_ (Vector implements Iterable, Collection)
```

8.8.2 Re-concretization

We conclude our discussion on interfaces with a program for testing various classes and interfaces that are provided as examples in this documentation. To test the `Collection` interface, we instantiate different wrapper classes and append their instances to a collection concretized by the `List` class, which specifies the format of storage.

The abstract nature of the `copy` procedure of `Collection` naturally facilitates a format conversion from one concrete type to another, which we refer to as "re-concretization" of an existing instance (a shallow copy is created).

The following program is available in the source file `examples/compile/merry.c_`

```
#include     "List._"
#include "Rational._"
#include   "Signed._"
#include     "Text._"
#include "Unsigned._"
#include   "Vector._"

Int_ main()
begin
    Var  year = init__(type_(Unsigned), 2023ULL);
    guard_(year)
    Var month = init__(type_(  Signed), 12LL);
    guard_(month)
    Var   day = init__(type_(Rational), 24.F);
    guard_(day)
    Var  poem = init__(Text);
    guard_(poem)
    guard_(parse__(poem, &"'Twas the night before Christmas ..."))
    Var  list = (Collection_ *)
    init__(type_(Collection), typex_(Collection, List));
    guard_(list)
    guard_(call_((Collection, append), list, year))
    guard_(call_((Collection, append), list, month))
    guard_(call_((Collection, append), list, day))
    guard_(call_((Collection, append), list, poem))
    print_("Initial concrete type:", concrete_(list)->type->name);
    write_(list);
    Var vector= (Collection_ *)
    init__(type_(Collection), typex_(Collection, Vector), type_(Object));
    guard_(vector)
    guard_(copy_(vector, list) == vector)
    print_("After re-concretization:", concrete_(vector)->type->name);
    write_(vector);
end
```

The numerous guard clauses throughout this program ensure that the program terminates if any allocation fails; readers may ignore this cautious bit of defensive programming. We instantiate the wrapper classes `Unsigned`, `Signed`, and `Rational` by calling their respective constructors with an initializer data; for the wrapper class `Text`, we first create an empty instance, and then invoke the `parse` method (via `parse__`) to copy the given string to the instance (note that it expects pointer to an array, so the address-of operator `&` has been used on the string literal).

If all of the above operations are successful, we instantiate the `Collection` interface with `List` as the concretizing class; note that the second argument to `init__` is made into an lvalue, and a pointer to it becomes the first element of `tape` parameter for `init` method. Since a `species` has not been specified, the `List` constructor assigns `type_(Object)` as the default value, which allows the four instances of diverse wrapper classes to be stored in the same collection. If all the `append` operations are successful, we print the name of the concretizing class, followed by writing the collection data to `stdout` (the default output stream). The `write` procedure of `Collection` does not use the `species` type to print the elements: since each element is an instance, it has its own `type` member, whose `write` procedure is invoked for printing each particular instance; successive elements are separated by a newline.

Next we create another empty instance of `Collection`, but this time we use `Vector` as the concretizing class. Notice the additional argument `type_(Object)` which explicitly states `species` of the new collection; this is required because the `Vector` class implements both `Collection` as well as its base interface `Iterable` (which is untyped), so in the absence of an explicit argument for `species`, it would be set to null, making it an invalid instance of `Collection` (as per its `validate` procedure), thereby causing a post-condition violation for `init` protocol of `Type`.

Our task of re-concretizing the `List`-based collection to a `Vector`-based one is accomplished by simply calling the `copy` method, with `vector` as the destination and `list` as the source. To confirm that a successful re-concretization has indeed occurred, we print the concrete type name of the `Vector`-based instance, and then print its elements. The outcome would be precisely the same as before, because even though `Collection` itself does not impose any ordering, both `List` and `Vector` store their elements as per the order of insertion, so the sequence is preserved.

8.8.2.1 Compilation

Before running this program, we need to compile the source files of all classes and interfaces used in this code, along with their base types. To avoid recompiling all source files for any change we make in our program, it is suggested that the dependencies should be compiled only once, and their resulting object files should be stored in `object/` directory: this task can be automated with the shell script `examples/build.sh` that needs to be executed once to populate the `object/` directory. Once that is done, we can compile our program and link the object files as:

```
cc_ compile/merry.c_ -xnone object/lib.o object/*/*.o object/*/*/*.o
```

The option `-xnone` (contraction of `-x none`) is used before the object files to undo the effect of `-xc` option that is part of the `cc_` alias: `-x c` tells the compiler (`gcc` or `clang` in our tests) to consider subsequent files with any extension as `C` programs, which is not the case for object files (otherwise recognized by their filename extension `.o`).

If things go well with the compilation, executing the program (as `./a.out`) should print the following output.

```
Initial concrete type: List
2023
12
24
'Twas the night before Christmas ...
After re-concretization: Vector
2023
12
24
'Twas the night before Christmas ...
```

Chapter 9

Library

C_ extends the C standard library with type definitions and some new functions. Minor extensions include trivial wrappers over standard function-like macros, named as per the C_ convention of a trailing underscore, to differentiate them from actual functions. C23 specifies 31 standard headers whose names end with `.h`, and for each of them, C_ provides an extension header whose name ends with `._`: each C_ header includes its corresponding C header as a base, and provides C_ style type definitions along with wrapper macros for the components of the base header.

Sections of this chapter are organized as per standard headers: each section lists the extensions provided by a particular C_ header. Details about standard types have been omitted for brevity: only the names are mentioned, and as always, their modifiable twins are named with a trailing underscore. Wrapper macros retain the same meaning as their base macros from the standard library: their only purpose is to ensure a uniform naming scheme as per C_ conventions. For some headers, the reference implementation also tries to bridge the gap between C18 and C23, by implementing new features of the current C standard if they are not already provided by the compiler.

NOTE In few cases, an extension header is only a wrapper that includes a C header without any new content.

9.1 Diagnostics `<assert._>`

9.1.1 Types

`<assert._>` defines the types `Stream` and `Site`. `Stream_` is a synonym for `Ptr_(File_)`, and `Site_` is a synonym for `struct Site`, which has the following members: `func` and `file` of type `Ptr_(Char)`, and `line` of type `Int_`.

NOTE `File_` is defined in `<stdio._>` as a synonym for the object type `FILE` (from `<stdio.h>`).

Recommended practice

Portable code should not rely on the order of members in `struct Site`.

9.1.2 Macros

`<assert._>` provides the object-like macros `ASSERT__` and `SITE`. `ASSERT__` expands to `0` or `1`, indicating whether the macro `DEBUG` was defined when `<assert._>` was last included. `SITE` expands to a non-modifiable lvalue of type `Site`, whose members `func`, `file`, and `line` respectively contain the values obtained from `__func__`, `__FILE__`, and `__LINE__`; this lvalue has automatic storage duration, and its lifetime is limited to the nearest enclosing block.

9.1.2.1 `assert_`

The `assert_` family can be used to ensure that some condition is satisfied at runtime. Unlike the `pre_` and `post_` facilities, the `assert_` family can be customized during translation: when compiled in debugging mode, failure of a given condition causes a diagnostic message to be printed on the standard error stream and the process terminates with exit status as one; when compiled in production mode, the condition is evaluated but the result is ignored.

NOTE C_ `assert_` family is not a wrapper over the C `assert` macro defined in `<assert.h>`; as with other extension headers, `<assert.h>` is included by `<assert._>`, and the `assert` macro can be configured with `NDEBUG`.

Syntax

`# include <assert._>`

`assert_` (*condition* [, *text*="*condition*" [, *site*=`SITE` [, *sink*=`stderr` [, *DEBUG*=`ASSERT__`]]]])
`assert_1_` (*condition*)
`assert_2_` (*condition* , *text*)
`assert_3_` (*condition* , *text* , *site*)
`assert_4_` (*condition* , *text* , *site* , *sink*)
`assert_5_` (*condition* , *text* , *site* , *sink* , *DEBUG*)

Constraints

condition shall be a scalar expression. *text* shall be a string. *site* shall be of type `Site`. *sink* shall be of type `Stream`. *DEBUG* shall expand to either `0` or `1`.

Semantics

`assert_` invokes `assert_`*n*`_` if the expanded argument sequence contains *n* arguments. Outcome of the expression is of type `Bool_`: when compiled with *DEBUG* expanding to `0`, the outcome is `(Bool)(`*condition*`)`; otherwise *DEBUG* expands to `1` and the outcome is `(Bool) 1`.

When compiled with *DEBUG* expanding to `1`, if *condition* compares equal to zero, the process prints a diagnostic message of the following form, and then terminates by calling `exit(1)`.

`Assertion failed:` *condition*`, function` *function-identifier*`, file` *file-name*`, line` *line-number*`.`

The stringified form of *condition* is used as default, which can be customized with the argument *text*; an empty string is used if *text* is null. *function-identifier*, *file-name*, and *line-number* indicate the location where the assertion failed. The message is written to the standard error stream `stderr` by default, which can be changed with the *sink*. `stderr` is also the fallback if *sink* is null; the behavior of writing to *sink* is undefined if it is not an output stream.

All the arguments are always evaluated exactly once, regardless of whether *DEBUG* expands to `0` or `1`.

9.2 Complex arithmetic `<complex._>`

`<complex._>` is available but does not provide any includable content if the macro `__STDC_NO_COMPLEX__` is defined.

9.2.1 Macros

`<complex._>` additionally defines these function-like macros: `cmplxf_`, `cmplx_`, `cmplxl_`, `cpowf_`, `cpow_`, `cpowl_`.

`cmplxf_`, `cmplx_`, and `cmplxl_` are trivial wrappers over `CMPLXF`, `CMPLX`, and `CMPLXL` (respectively).

`cpowf_`, `cpow_`, and `cpowl_` generalize the functions `cpowf`, `cpow`, and `cpowl` using a right reduction. For example, `cpow_(2, 3, 4)` expands to `cpow(2, cpow(3, 4))`, and converting the result to an integer equals 512.

9.3 Character handling `<ctype._>`

`<ctype._>` only includes the C standard header `<ctype.h>` and does not provide any additional content.

9.4 Errors `<errno._>`

9.4.1 Types

If both the macros `__STDC_LIB_EXT1__` and `__STDC_WANT_LIB_EXT1__` remain defined before including `<errno._>`, then the type `Errno` is defined, with its twin `Errno_` being a synonym for `errno_t`.

9.4.2 Macros

`<errno._>` additionally defines the object-like macros `errnum` and `errnum_`: `errnum_` is same as `errno` and is modifiable, whereas `errnum` is its non-modifiable variant; in other words, `errnum_` is an lvalue, but `errnum` is not.

9.5 Floating-point environment `<fenv._>`

9.5.1 Types

`<fenv._>` additionally provides the following type synonyms: `Fenv`, `Fexcept`, `Femode` (introduced in C23).

NOTE `Femode_` is a synonym for `femode_t`, available only if `__STDC_VERSION_FENV_H__` is defined by `<fenv.h>`.

9.6 Characteristics of floating types `<float._>`

`<float._>` only includes the C standard header `<float.h>` and does not provide any additional content.

9.7 Format conversion of integer types `<inttypes._>`

`<inttypes._>` includes the extension header `<stdint._>`, as the C base header `<inttypes.h>` includes `<stdint.h>`.

9.7.1 Types

`<inttypes._>` additionally defines the type `Imaxdiv`: `Imaxdiv_` is a synonym for `imaxdiv_t` from `<inttypes.h>`.

9.7.2 Functions

`<inttypes._>` additionally provides inline definitions for the following functions:

```
UIntmax_ uimaxabs(Intmax);
UIntmax_ gcd (UIntmax, UIntmax);
UIntmax_ umax(UIntmax, UIntmax);
UIntmax_ umin(UIntmax, UIntmax);
 Intmax_ smax( Intmax,  Intmax);
 Intmax_ smin( Intmax,  Intmax);
```

`uimaxabs` is similar to `imaxabs`, except that the return type is `UIntmax_`: it returns the absolute value of an integer, and the behavior is well-defined for all values in range of the parameter type `Intmax`.

`gcd` returns the greatest common divisor of two non-negative integers; output is zero iff both inputs are zero.

`umax` and `umin` respectively return the maximum and minimum of two unsigned integers; `smax` and `smin` are the signed integer counterparts.

9.7.3 Macros

`<inttypes._>` additionally defines the following function-like macros: `uintmax_`, `gcd_`, `umax_`, `umin_`, `smax_`, `smin_`.

`uintmax_` exhibits well-defined behavior for large unsigned integers that cause signed overflow with `uintmax`.

`gcd_`, `umax_`, `umin_`, `smax_`, and `smin_` generalize the respective functions with fold/reduce operations; if a single argument is given, then that is the outcome after conversion to `UIntmax` or `Intmax` (as per the return type). The sequence of evaluation of arguments is unspecified.

NOTE The reference implementation provides generalization macros using `reduce__` from ellipsis framework.

9.8 Alternative spellings `<iso646._>`

`<iso646._>` only includes the C standard header `<iso646.h>` and does not provide any additional content.

9.9 Characteristics of integer types `<limits._>`

9.9.1 Macros

9.9.1.1 `bitmax_`

Syntax

`bitmax_ (` *width* `)`

Constraints

width shall have an integer type.

Semantics

If *width* is a non-negative integer not exceeding `width_(ULLong)`, then `bitmax_` gives the maximum value that is representable using *width* bits; otherwise the behavior is undefined. The result is an expression of type `ULLong_`, and it is an integer constant expression only if the argument *width* is also an integer constant expression.

9.9.1.2 `bitlen__`

Syntax

`bitlen__ (` *bitmax* `)`

Constraints

bitmax shall have an integer type.

Semantics

If *bitmax* is a non-negative integer that is one less than some power of 2, then `bitlen__` acts as the inverse of `bitmax_`, and computes the number of significant bits; otherwise the behavior is implementation-defined.

The result is an integer constant expression only if the argument *bitmax* is also an integer constant expression.

NOTE The reference implementation provides `bitlen__` with a formula suggested by Hallvard B. Furuseth; the basic technique can be found at `https://groups.google.com/g/comp.lang.c/c/NfedEFBFJ0k/m/9HAqis6IDqcJ`

9.9.1.3 Widths

<limits._> defines the following object-like macros introduced in C23, if they are not available in <limits.h>.

```
BOOL_WIDTH   UCHAR_WIDTH   USHRT_WIDTH   UINT_WIDTH   ULONG_WIDTH   ULLONG_WIDTH
CHAR_WIDTH    CHAR_WIDTH    SHRT_WIDTH    INT_WIDTH    LONG_WIDTH    LLONG_WIDTH
```

Additionally, the following object-like macros (left side) are alternative names for standard macros (right side).

<limits._>	<limits.h>	<limits._>	<limits.h>
UBYTE_WIDTH	UCHAR_WIDTH	USHORT_WIDTH	USHRT_WIDTH
UBYTE_MAX	UCHAR_MAX	USHORT_MAX	USHRT_MAX
BYTE_WIDTH	SCHAR_WIDTH	SHORT_WIDTH	SHRT_WIDTH
BYTE_MAX	SCHAR_MAX	SHORT_MAX	SHRT_MAX
BYTE_MIN	SCHAR_MIN	SHORT_MIN	SHRT_MIN

9.10 Localization <locale._>

9.10.1 Types

<locale._> additionally defines the type `Lconv`; `Lconv_` is a synonym for `struct lconv` from <locale.h>.

9.11 Mathematics <math._>

9.11.1 Types

<math._> provides the type synonyms `Float_t` and `Double_t`.

Additionally, if both the macros `__STDC_IEC_60559_DFP__` and `__STDC_WANT_IEC_60559_EXT__` remain defined before including <math._>, then the type synonyms `Decimal32_t` and `Decimal64_t` are also provided.

9.11.2 Macros

<math._> defines the following function-like wrapper macros as alternative names for standard macros from <math.h>.

```
fpclassify_       iscanonical_   isfinite_        issignaling_   isnan_
signbit_          isnormal_      isinf_           issubnormal_   iszero_

isgreater_        isless_        islessgreater_   iseqsig_
isgreaterequal_   islessequal_   isunordered_
```

The following function-like macros generalize standard functions from <math.h> using fold/reduce operations.

powf_	pownf_	powrf_	fadd_	fmul_
pow_	pown_	powr_	faddl_	fmull_
powl_	pownl_	powrl_	daddl_	dmull_

fmaxf_	fmaximumf_	fmaximum_magf_	fmaximum_numf_	fmaximum_mag_numf_
fmax_	fmaximum_	fmaximum_mag_	fmaximum_num_	fmaximum_mag_num_
fmaxl_	fmaximuml_	fmaximum_magl_	fmaximum_numl_	fmaximum_mag_numl_
fminf_	fminimumf_	fminimum_magf_	fminimum_numf_	fminimum_mag_numf_
fmin_	fminimum_	fminimum_mag_	fminimum_num_	fminimum_mag_num_
fminl_	fminimuml_	fminimum_magl_	fminimum_numl_	fminimum_mag_numl_

The following fold/reduce generalizations are provided only if __STDC_IEC_60559_DFP__ is defined.

powd32_	pownd32_	powrd32_	d32addd64_	d32muld64_
powd64_	pownd64_	powrd64_	d32addd128_	d32muld128_
powd128_	pownd128_	powrd128_	d64addd128_	d64muld128_

fmaxd32_	fmaximumd32_	fmaximum_magd32_	fmaximum_numd32_	fmaximum_mag_numd32_
fmaxd64_	fmaximumd64_	fmaximum_magd64_	fmaximum_numd64_	fmaximum_mag_numd64_
fmaxd128_	fmaximumd128_	fmaximum_magd128_	fmaximum_numd128_	fmaximum_mag_numd128_
fmind32_	fminimumd32_	fminimum_magd32_	fminimum_numd32_	fminimum_mag_numd32_
fmind64_	fminimumd64_	fminimum_magd64_	fminimum_numd64_	fminimum_mag_numd64_
fmind128_	fminimumd128_	fminimum_magd128_	fminimum_numd128_	fminimum_mag_numd128_

Macros in the pow family generalize the respective functions using a right-to-left reduction; for example, pow_(4, 3, 2) expands to pow(4, pow(3, 2)). The sequence of folding/reduction for rest of the generalization macros in <math._> is implementation-defined. In all cases, the order of evaluation of arguments is unspecified.

9.12 Non-local jumps <setjmp._>

9.12.1 Types

<setjmp._> additionally defines the type Jmp_buf, with Jmp_buf_ being a synonym for jmp_buf from <setjmp.h>.

9.12.2 Macros

<setjmp._> additionally defines the function-like macro setjmp_ as a trivial wrapper over setjmp from <setjmp.h>.

9.13 Signal handling <signal._>

9.13.1 Types

<signal._> additionally defines the type Sig_atomic, with Sig_atomic_ being a synonym for sig_atomic_t.

9.14 Alignment <stdalign._>

`<stdalign._>` only includes the C standard header `<stdalign.h>` and does not provide any additional content.

9.15 Variable arguments <stdarg._>

9.15.1 Types

`<stdarg._>` additionally defines the type `VA_list`, with `VA_list_` being a synonym for `va_list` from `<stdarg.h>`.

9.15.2 Macros

`va_copy_`, `va_start_`, `va_arg_`, and `va_end_` are trivial wrappers over `va_copy`, `va_start`, `va_arg`, and `va_end`.

9.16 Atomics <stdatomic._>

`<stdatomic._>` is available but does not provide includable content if the macro `__STDC_NO_ATOMICS__` is defined.

9.16.1 Types

`Memory_order` and `Atomic_flag` are synonyms for non-modifiable `memory_order` and `atomic_flag` (respectively).

Additionally, the following synonyms are defined for standard integer types: for a non-modifiable type named as `Atomic_`*Type*, its modifiable twin `Atomic_`*Type*`_` is a synonym for `atomic_`*type*. Similarly, for a non-modifiable unsigned type named as `Atomic_U`*Type*, its modifiable twin `Atomic_U`*Type*`_` is a synonym for `atomic_u`*type*.

Atomic_Bool	Atomic_UByte	Atomic_UShort	Atomic_UInt	Atomic_ULong	Atomic_ULLong
Atomic_Char	Atomic_Byte	Atomic_Short	Atomic_Int	Atomic_Long	Atomic_LLong

For the following types, the modifiable twin named as `Atomic_`*Type*`_` is a synonym for `atomic_`*type*`_t`.

For the unsigned types, the modifiable twin named as `Atomic_U`*Type*`_` is a synonym for `atomic_u`*type*`_t`.

Atomic_Char8	Atomic_Char16	Atomic_Char32	Atomic_WChar
Atomic_UInt_least8	Atomic_UInt_least16	Atomic_UInt_least32	Atomic_UInt_least64
Atomic_Int_least8	Atomic_Int_least16	Atomic_Int_least32	Atomic_Int_least64
Atomic_UInt_fast8	Atomic_UInt_fast16	Atomic_UInt_fast32	Atomic_UInt_fast64
Atomic_Int_fast8	Atomic_Int_fast16	Atomic_Int_fast32	Atomic_Int_fast64

Atomic_UIntptr	Atomic_Size	Atomic_UIntmax
Atomic_Intptr	Atomic_Ptrdiff	Atomic_Intmax

NOTE `Atomic_UByte_` and `Atomic_Byte_` are synonyms for `atomic_uchar` and `atomic_schar`. `Atomic_Char8_` is a synonym for `atomic_char8_t`, available only if `__STDC_VERSION_STDATOMIC_H__` is defined by `<stdatomic.h>`.

9.16.2 Macros

`<stdatomic._>` defines the function-like macro `kill_dependency_` as a trivial wrapper over `kill_dependency`.

9.17 Bit and byte utilities `<stdbit._>`

9.17.1 Types

`<stdbit._>` defines a pair of synonyms for the integer type `size_t`: `Size` is non-modifiable and `Size_` is modifiable.

If fixed-width integer types are supported by the implementation, then the following synonyms are defined.

`UInt8`	`UInt16`	`UInt32`	`UInt64`
`Int8`	`Int16`	`Int32`	`Int64`

The following synonyms are defined for minimum-width integer types.

`UInt_least8`	`UInt_least16`	`UInt_least32`	`UInt_least64`
`Int_least8`	`Int_least16`	`Int_least32`	`Int_least64`

9.17.2 Macros

Following function-like macros are trivial wrappers over their standard counterparts without a trailing underscore.

`stdc_leading_zeros_`	`stdc_first_leading_zero_`	`stdc_count_zeros_`	`stdc_bit_floor_`
`stdc_leading_ones_`	`stdc_first_leading_one_`	`stdc_count_ones_`	`stdc_bit_ceil_`
`stdc_trailing_zeros_`	`stdc_first_trailing_zero_`	`stdc_has_single_bit_`	
`stdc_trailing_ones_`	`stdc_first_trailing_one_`	`stdc_bit_width_`	

9.18 Boolean type and values `<stdbool._>`

`<stdbool._>` only includes the C standard header `<stdbool.h>` and does not provide any additional content.

9.19 Checked integer arithmetic `<stdckdint._>`

9.19.1 Macros

Function-like macros `ckd_add_`, `ckd_sub_`, and `ckd_mul_` are trivial wrappers over `ckd_add`, `ckd_sub`, and `ckd_mul`.

9.20 Common definitions `<stddef._>`

9.20.1 Types

Following type synonyms are defined, and in each case, the modifiable twin named *Type*`_` is a synonym for *type*`_t`.

`Size` `Ptrdiff` `Max_align` `WChar` `Nullptr`

Additionally, if both the macros `__STDC_LIB_EXT1__` and `__STDC_WANT_LIB_EXT1__` remain defined before including `<stddef._>`, then the type `Rsize` is also defined, with its twin `Rsize_` being a synonym for `rsize_t`.

NOTE The synonyms `Nullptr` and `Nullptr_` are available only if the implementation provides `nullptr_t`.

9.20.2 Macros

`offsetof_` is a trivial wrapper over `offsetof`, and the object-like macro `UNREACHABLE` expands to `unreachable()`.

9.21 Integer types <stdint._>

9.21.1 Types

9.21.1.1 Exact-width integer types

If exact-width types are supported by the implementation, then the following synonyms are provided by <stdint._>. In each case, the modifiable counterpart UInt$N$_ is a synonym for uint$N$_t, and Int$N$_ is a synonym for int$N$_t.

UInt8	UInt16	UInt32	UInt64
Int8	Int16	Int32	Int64

9.21.1.2 Minimum-width integer types

UInt_least$N$_ is a synonym for uint_least$N$_t, and Int_least$N$_ is a synonym for int_least$N$_t.

UInt_least8	UInt_least16	UInt_least32	UInt_least64
Int_least8	Int_least16	Int_least32	Int_least64

9.21.1.3 Fastest minimum-width integer types

UInt_fast$N$_ is a synonym for uint_fast$N$_t, and Int_fast$N$_ is a synonym for int_fast$N$_t.

UInt_fast8	UInt_fast16	UInt_fast32	UInt_fast64
Int_fast8	Int_fast16	Int_fast32	Int_fast64

9.21.1.4 Integer types capable of holding object pointers

If uintptr_t and intptr_t are available, then two pairs of synonyms UIntptr and Intptr are defined.

9.21.1.5 Greatest-width integer types

UIntmax_ and Intmax_ are synonyms for uintmax_t and intmax_t, with UIntmax and Intmax being non-modifiable.

9.21.2 Macros

9.21.2.1 Macros for minimum-width integer constants

Each UINTN_C_ is a trivial wrapper over UINTN_C, and each INTN_C_ is a trivial wrapper over INTN_C.

UINT8_C_	UINT16_C_	UINT32_C_	UINT64_C_
INT8_C_	INT16_C_	INT32_C_	INT64_C_

9.21.2.2 Macros for greatest-width integer constants

The function-like macros UINTMAX_C_ and INTMAX_C_ are trivial wrappers over UINTMAX_C and INTMAX_C.

9.22 Input/output <stdio._>

9.22.1 Types

File_ and Fpos_ are synonyms for FILE and fpos_t, with File and Fpos being the non-modifiable counterparts.

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <stdio._>, then the synonyms Errno and Rsize are also defined, corresponding to the types errno_t and rsize_t.

9.22.2 Macros

The object-like macro L_TMPNAM is an alternative name for L_tmpnam. The object-like macros STDIN, STDOUT, and STDERR expand to non-lvalue expressions corresponding to the standard I/O streams stdin, stdout, and stderr.

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <stdio._>, then the object-like macro L_TMPNAM_S is defined as an alternative name for L_tmpnam_s.

9.23 General utilities <stdlib._>

9.23.1 Types

The following type synonyms are defined, and in each case, the modifiable twin is named with a trailing underscore.

```
Div   LDiv   LLDiv   Once_flag   Size   WChar
```

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <stdlib._>, the following synonyms are defined with modifiable counterparts: Errno, Rsize, Constraint_handler.

9.23.2 Functions

9.23.2.1 Memory management functions

<stdlib._> provides inline definition for the following additional function that serves as a wrapper over malloc.

```
Void_ *malloc2(Size, Size);
```

The primary use of this function is to detect whether the mathematical value of the product of arguments can be represented in the type Size: if yes, then malloc is invoked with that product; otherwise a null pointer is returned.

9.23.2.2 Integer arithmetic functions

<stdlib._> provides inline definitions for the following additional functions to get the absolute value of an integer.

```
UInt_ uabs(Int);        ULong_ ulabs(Long);        ULLong_ ullabs(LLong);
```

These functions return unsigned types, so obtaining the absolute value of the most negative integer representable in the parameter type does not cause any signed overflow, and the behavior is well-defined for all signed values.

9.23.3 Macros

The function-like macros uabs_, ulabs_, and ullabs_ are non-trivial wrappers over the functions uabs, ulabs, and ullabs. These macros conditionally invoke their respective functions only if their argument has a signed type; otherwise the argument value is cast to the unsigned return type of the corresponding function, without calling it.

NOTE If the argument happens to be unsigned, calling the functions directly can cause signed overflow if the argument value cannot be represented in the signed parameter type. The wrapper macros avoid such predicaments.

9.24 _Noreturn <stdnoreturn._>

9.24.1 Macros

The object-like macro NORETURN is defined as an alternative name for noreturn, which itself expands to _Noreturn.

9.25 String handling <string._>

9.25.1 Types

Size is defined as a synonym for non-modifiable size_t, with Size_ being its modifiable counterpart.

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <string._>, then the synonyms Errno and Rsize are also defined, corresponding to types errno_t and rsize_t.

9.26 Type-generic math <tgmath._>

<tgmath._> includes <tgmath.h> along with <math._> and <complex._>; it does not provide any additional content.

9.27 Threads <threads._>

<threads._> is available but does not provide any includable content if the macro __STDC_NO_THREADS__ is defined. Otherwise <threads._> includes extension header <time._>, as the C base header <threads.h> includes <time.h>.

9.27.1 Types

Following type synonyms are defined, and in each case, the modifiable twin named *Type*_ is a synonym for *type*_t.

Cnd Thrd Tss Mtx Tss_dtor Thrd_start Once_flag

9.28 Date and time <time._>

9.28.1 Types

The following table lists the non-modifiable type synonyms along with the corresponding modifiable type.

<time._>	<time.h>	<time._>	<time.h>
Size	size_t	Timespec	struct timespec
Clock	clock_t	Tm	struct tm
Time	time_t		

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <time._>, then the synonyms Errno and Rsize are also defined, corresponding to the types errno_t and rsize_t.

9.29 Unicode utilities <uchar._>

9.29.1 Types

The following synonyms are defined, and in each case, the modifiable twin *Type_* is a synonym for *type*_t.

MBstate Size Char8 Char16 Char32

NOTE Char8_ is a synonym for char8_t, available only if __STDC_VERSION_UCHAR_H__ is defined by <uchar.h>.

9.30 Extended multibyte and wide character utilities <wchar._>

9.30.1 Types

The following synonyms are defined along with their modifiable counterparts (Tm_ is a synonym for struct tm).

WChar Size MBstate WInt Tm

Additionally, if both macros __STDC_LIB_EXT1__ and __STDC_WANT_LIB_EXT1__ remain defined before including <wchar._>, then the synonyms Errno and Rsize are also defined, corresponding to the types errno_t and rsize_t.

9.31 Wide character classification and mapping utilities <wctype._>

9.31.1 Types

The following type synonyms are defined, and in each case, the modifiable twin W*Type_* is a synonym for W*type*_t.

WInt WCtrans WCtype

9.32 Complete library <lib._>

The header <lib._> includes the following C_ extension headers for the C standard library.

<assert._>	<setjmp._>	<stdlib._>
<complex._>	<signal._>	<stdnoreturn._>
<ctype._>	<stdalign._>	<string._>
<errno._>	<stdarg._>	<tgmath._>
<fenv._>	<stdatomic._>	<threads._>
<float._>	<stdbit._>	<time._>
<inttypes._>	<stdbool._>	<uchar._>
<iso646._>	<stdckdint._>	<wchar._>
<limits._>	<stddef._>	<wctime._>
<locale._>	<stdint._>	
<math._>	<stdio._>	

<stdbit._> and <stdckdint._> are included only if __STDC_VERSION__ is not less than 202311 (C23 or later).

Appendix A

Examples

A.1 Unsigned

A.1.1 Declaration

examples/include/class/Unsigned._

```
#ifndef UNSIGNED__
#define UNSIGNED__

#include <c._>

#define UNSIGNED_MAX 18446744073709551615ULL

#define Unsigned_EXTENDS Object,\
 validate, init, compare, copy, read, write,\
 parse, text, decode, encode, add, sub, mul, div

class_ (Unsigned)
    ULLong_ value;
fin

#endif
```

A.1.2 Definition

examples/compile/class/Unsigned.c_

```
#include "Unsigned._"

 define_ (Unsigned)
```

A.1.3 validate

examples/include/class/Unsigned/validate._

```
#ifndef   UNSIGNED__VALIDATE__
#define   UNSIGNED__VALIDATE__

#include "Unsigned._"

private inline Bool_ validator(let ULLong value)
begin
    return value <= UNSIGNED_MAX;
end

#endif
```

examples/compile/class/Unsigned/validate.c_

```
#include "Unsigned/validate._"

procedure_(Bool_, (Unsigned, validate),
(let Ptr (Unsigned), this))
    return validator(this->value);
end
```

A.1.4 init

examples/compile/class/Unsigned/init.c_

```
#include  "Unsigned/validate._"

procedure_(Unsigned_ *, (Unsigned, init),
(let Type, type),
(let Tape, tape))
    Var value = *tape? *(ULLong *)*tape : 0;
    guard_(validator(value), NULL)
    Var object = new__(Unsigned);
    guard_(object, NULL)
    object->type  = type ;
    object->value = value;
    return object;
end
```

A.1.5 compare

examples/compile/class/Unsigned/compare.c_

```
#include "Unsigned._"

procedure_(LLong_  , (Unsigned, compare),
(let Ptr (Unsigned), this),
(let Ptr (Unsigned), that))
    return this->value < that->value
    ? -1 : this->value > that->value;
end
```

A.1.6 copy

examples/compile/class/Unsigned/copy.c_

```
#include  "Unsigned._"

procedure_(Unsigned_*, (Unsigned, copy),
(let  Ptr (Unsigned_), this),
(let  Ptr (Unsigned ), that))
    Var  copy = need__(this);
    guard_(copy, NULL)
    if (!this) copy->type = type_(Unsigned);
    copy->value = that->value;
    return copy;
end
```

A.1.7 read

examples/compile/class/Unsigned/read.c_

```
#include  "Unsigned/validate._"

procedure_(Unsigned_*, (Unsigned, read),
(let Ptr  (Unsigned_), this),
(let Stream, in))
    return input__(in, this->value) == 1
    && validator(this->value) ? this : NULL;
end
```

A.1.8 write

examples/compile/class/Unsigned/write.c_

```
#include "Unsigned._"

procedure_(LLong_, (Unsigned, write),
(let Ptr (Unsigned), this),
(let Stream, out))
    return output__(out, this->value);
end
```

A.1.9 parse

examples/compile/class/Unsigned/parse.c_

```
#include "Unsigned/validate._"

#include <errno._>

procedure_(Unsigned_*, (Unsigned, parse),
(let Ptr (Unsigned_), this),
(let Ptr (Size_), length),
(let Ptr (Void ), in))
    Void_ *memchr(Void *, Int, Size);
    guard_(memchr(in, '\0', *length), NULL)
    Auto_  endptr_ = (Char_ *)unqual__(in);
    errnum_ = 0;
    ULLong_ strtoull(String, Char_ **, Int);
    Var value = strtoull(in, &endptr_, 0);
    guard_((*length = endptr_-(Char *)in) && !errnum
    && validator(value), (errnum_ = 0, NULL))
    this->value = value;
    return this;
end
```

A.1.10 text

examples/compile/class/Unsigned/text.c_

```
#include "Unsigned._"

procedure_(Void_ *, (Unsigned, text),
(let Ptr (Unsigned), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    Var buflen = output__((String_){0}, this->value) + 1U;
    Var buf = (out && *length >= buflen)? (String_*)out : new__(Char_[buflen]);
    *length = buflen;
    guard_(buf, NULL)
    output__(*buf, this->value);
    return buf;
end
```

A.1.11 decode

examples/compile/class/Unsigned/decode.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, decode),
(let Ptr (Unsigned_), this),
(let Ptr (Size_), length),
(let Ptr (Encoding), in))
    guard_(8 <= *length, NULL)
    *length = 8;
    this->value =
     ((*in)[0] & 255)  +
    (((*in)[1] & 255U  ) << 8 ) +
    (((*in)[2] & 255UL ) << 16) +
    (((*in)[3] & 255UL ) << 24) +
    (((*in)[4] & 255ULL) << 32) +
    (((*in)[5] & 255ULL) << 40) +
    (((*in)[6] & 255ULL) << 48) +
    (((*in)[7] & 255ULL) << 56) ;
    return this;
end
```

A.1.12 encode

examples/compile/class/Unsigned/encode.c_

```
#include "Unsigned._"

procedure_(Encoding_*, (Unsigned, encode),
(let Ptr (Unsigned), this),
(let Ptr (Size_) , length),
(let Ptr (Encoding_), out))
    Var enc = (out &&  8 <= *length)? out : new__(UByte_ [8]);
    *length = 8;
    guard_(enc, NULL);
    Var value = this->value;
    (*enc)[0]  =  value     & 255;
    (*enc)[1]  =  value>>8  & 255;
    (*enc)[2]  =  value>>16 & 255;
    (*enc)[3]  =  value>>24 & 255;
    (*enc)[4]  =  value>>32 & 255;
    (*enc)[5]  =  value>>40 & 255;
    (*enc)[6]  =  value>>48 & 255;
    (*enc)[7]  =  value>>56 & 255;
    return enc;
end
```

A.1.13 add

examples/compile/class/Unsigned/add.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, add),
(let Ptr (Unsigned_), sum),
(let Ptr (Unsigned ), augend),
(let Ptr (Unsigned ), addend))
    Var result = need__(sum);
    guard_(result, NULL)
    if (!sum) result->type = type_(Unsigned);
    result->value = (augend->value + addend->value) & UNSIGNED_MAX;
    return result;
end
```

A.1.14 sub

examples/compile/class/Unsigned/sub.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, sub),
(let Ptr (Unsigned_), difference),
(let Ptr (Unsigned ), minuend),
(let Ptr (Unsigned ), subtrahend))
    Var result = need__(difference);
    guard_(result, NULL)
    if (!difference) result->type = type_(Unsigned);
    result->value = minuend->value - subtrahend->value;
    return result;
end
```

A.1.15 mul

examples/compile/class/Unsigned/mul.c_

```
#include "Unsigned._"

procedure_(Unsigned_*, (Unsigned, mul),
(let Ptr (Unsigned_), product),
(let Ptr (Unsigned ), multiplier),
(let Ptr (Unsigned ), multiplicand))
    Var result = need__(product);
    guard_(result, NULL)
    if (!product) result->type = type_(Unsigned);
    result->value = (multiplier->value * multiplicand->value) & UNSIGNED_MAX;
    return result;
end
```

A.1.16 div

examples/compile/class/Unsigned/div.c_

```
#include  "Unsigned._"

procedure_(Unsigned_*, (Unsigned, div),
(let  Ptr (Unsigned_), output),
(let  Ptr (Unsigned ), dividend),
(let  Ptr (Unsigned ), divisor))
    guard_(divisor->value, NULL)
    Var result = need__(output);
    guard_(result, NULL)
    if (!output) result->type = type_(Unsigned);
    result->value  =  dividend->value / divisor->value;
    return result;
end
```

A.2 Signed

A.2.1 Declaration

examples/include/class/Signed._

```
#ifndef SIGNED__
#define SIGNED__

#include <c._>

#define SIGNED_MAX 9223372036854775807LL
#define SIGNED_MIN -SIGNED_MAX

#define Signed_EXTENDS Object,\
 validate, init, compare, copy, read, write,\
 parse, text, decode, encode, add, sub, mul, div

class_ (Signed)
    LLong_ value;
fin

#endif
```

A.2.2 Definition

examples/compile/class/Signed.c_

```
#include "Signed._"

 define_ (Signed)
```

A.2.3 validate

examples/include/class/Signed/validate._

```
#ifndef   SIGNED__VALIDATE__
#define   SIGNED__VALIDATE__

#include "Signed._"

private inline Bool_ validator(let LLong value)
begin
    return value >= SIGNED_MIN
    &&     value <= SIGNED_MAX;
end

#endif
```

examples/compile/class/Signed/validate.c_

```
#include "Signed/validate._"

procedure_(Bool_ , (Signed, validate),
(let Ptr (Signed), this))
    return validator(this->value);
end
```

A.2.4 init

examples/compile/class/Signed/init.c_

```
#include  "Signed/validate._"

procedure_(Signed_ *, (Signed, init),
(let Type, type),
(let Tape, tape))
    Var value = *tape? *(LLong *)*tape : 0;
    guard_(validator(value), NULL)
    Var object = new__(Signed);
    guard_(object, NULL)
    object->type  = type ;
    object->value = value;
    return object;
end
```

A.2.5 compare

examples/compile/class/Signed/compare.c_

```
#include "Signed._"

procedure_(LLong_, (Signed, compare),
(let Ptr (Signed), this),
(let Ptr (Signed), that))
    return this->value < that->value
    ? -1 : this->value > that->value;
end
```

A.2.6 copy

examples/compile/class/Signed/copy.c_

```
#include  "Signed._"

procedure_(Signed_*, (Signed, copy),
(let  Ptr (Signed_), this),
(let  Ptr (Signed ), that))
    Var copy= need__(this);
    guard_(copy, NULL)
    if (!this) copy->type = type_(Signed);
    copy->value = that->value;
    return copy;
end
```

A.2.7 read

examples/compile/class/Signed/read.c_

```
#include  "Signed/validate._"

procedure_(Signed_*, (Signed, read),
(let Ptr  (Signed_), this),
(let Stream, in))
    return input__(in, this->value) == 1
    && validator(this->value) ? this : NULL;
end
```

A.2.8 write

examples/compile/class/Signed/write.c_

```
#include "Signed._"

procedure_(LLong_, (Signed, write),
(let Ptr (Signed), this),
(let Stream, out))
    return output__(out, this->value);
end
```

A.2.9 parse

examples/compile/class/Signed/parse.c_

```
#include "Signed/validate._"

#include <errno._>

procedure_(Signed_*, (Signed, parse),
(let Ptr (Signed_), this),
(let Ptr (Size_), length),
(let Ptr (Void ), in))
    Void_ *memchr(Void *, Int, Size);
    guard_(memchr(in, '\0', *length), NULL)
    Auto_ endptr_ = (Char_ *)unqual__(in);
    errnum_ = 0;
    LLong_ strtoll(String, Char_ **, Int);
    Var value = strtoll(in, &endptr_, 0);
    guard_((*length = endptr_-(Char *)in) && !errnum
    && validator(value), (errnum_ = 0, NULL))
    this->value = value;
    return this;
end
```

A.2.10 text

examples/compile/class/Signed/text.c_

```
#include "Signed._"

procedure_(Void_*, (Signed, text),
(let Ptr (Signed), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    Var buflen = output__((String_){0}, this->value) + 1U;
    Var buf = (out && *length >= buflen)? (String_*)out : new__(Char_[buflen]);
    *length = buflen;
    guard_(buf, NULL)
    output__(*buf, this->value);
    return buf;
end
```

A.2.11 decode

examples/compile/class/Signed/decode.c_

```
#include "Signed._"

procedure_(Signed_*, (Signed, decode),
(let Ptr (Signed_), this),
(let Ptr (Size_), length),
(let Ptr (Encoding), in))
    guard_(8 <= *length, NULL)
    *length = 8;
    this->value =
     ((*in)[0] & 255) +
    (((*in)[1] & 255U ) << 8 ) +
    (((*in)[2] & 255UL) << 16) +
    (((*in)[3] & 255UL) << 24) +
    (((*in)[4] & 255LL) << 32) +
    (((*in)[5] & 255LL) << 40) +
    (((*in)[6] & 255LL) << 48) +
    (((*in)[7] & 127LL) << 56) ;
    if ((*in)[7] & 128) this->value = -this->value;
    return this;
end
```

A.2.12 encode

examples/compile/class/Signed/encode.c_

```
#include "Signed._"

procedure_(Encoding_*, (Signed, encode),
(let Ptr (Signed), this),
(let Ptr (Size_ ), length),
(let Ptr (Encoding_), out))
    Var enc = (out && 8 <= *length)? out : new__(UByte_ [8]);
    *length = 8;
    guard_(enc, NULL);
    Var value = (ULLong)(this->value<0 ? -this->value : this->value);
    (*enc)[0] = value     & 255;
    (*enc)[1] = value>>8  & 255;
    (*enc)[2] = value>>16 & 255;
    (*enc)[3] = value>>24 & 255;
    (*enc)[4] = value>>32 & 255;
    (*enc)[5] = value>>40 & 255;
    (*enc)[6] = value>>48 & 255;
    (*enc)[7] = value>>56 & 127;
    if (this->value < 0) (*enc)[7] |= 128;
    return enc;
end
```

A.2.13 add

examples/compile/class/Signed/add.c_

```
#include "Signed._"

procedure_(Signed_*, (Signed, add),
(let Ptr (Signed_), sum),
(let Ptr (Signed ), augend),
(let Ptr (Signed ), addend))
    Var result = need__(sum);
    guard_(result, NULL)
    if (!sum) result->type = type_(Signed);
    Var_ l_ = augend->value;
    Var_ r_ = addend->value;
    Var sign  =  0>l_ &&  0>r_;
    if (sign) l_ = -l_, r_ = -r_;
    else guard_(0<l_ &&  0<r_, (result->value = l_+r_, result))
    result->value  =  ((ULLong)l_ + (ULLong)r_) & SIGNED_MAX;
    if (sign) result->value = -result->value;
    return result;
end
```

A.2.14 sub

examples/compile/class/Signed/sub.c_

```
#include "Signed._"

procedure_(Signed_*, (Signed, sub),
(let Ptr (Signed_), difference),
(let Ptr (Signed ), minuend),
(let Ptr (Signed ), subtrahend))
    Var result = need__(difference);
    guard_(result, NULL)
    if (!difference) result->type = type_(Signed);
    Var_ l_ = minuend->value;
    Var_ r_ = subtrahend->value;
    Var sign  =  0>l_ &&  0<r_;
    if (sign) l_ = -l_, r_ = -r_;
    else guard_(0<l_ &&  0>r_, (result->value = l_-r_, result))
    result->value  =  ((ULLong)l_ - (ULLong)r_) & SIGNED_MAX;
    if (sign) result->value = -result->value;
    return result;
end
```

A.2.15 mul

examples/compile/class/Signed/mul.c_

```
#include "Signed._"

procedure_(Signed_*, (Signed, mul),
(let Ptr (Signed_), product),
(let Ptr (Signed ), multiplier),
(let Ptr (Signed ), multiplicand))
    Var result = need__(product);
    guard_(result, NULL)
    if (!product) result->type = type_(Signed);
    Var l = multiplier->value;
    Var r = multiplicand->value;
    result->value = ((ULLong)(l<0 ? -l:l) * (ULLong)(r<0 ? -r:r)) & SIGNED_MAX;
    if (l && (0<l iff 0>r) && r) result->value = -result->value;
    return result;
end
```

A.2.16 div

examples/compile/class/Signed/div.c_

```
#include "Signed._"

procedure_(Signed_*, (Signed, div),
(let Ptr (Signed_), output),
(let Ptr (Signed ), dividend),
(let Ptr (Signed ), divisor))
    guard_(divisor->value, NULL)
    Var result = need__(output);
    guard_(result, NULL)
    if (!output) result->type = type_(Signed);
    result->value = dividend->value / divisor->value;
    return result;
end
```

A.3 Rational

A.3.1 Declaration

examples/include/class/Rational._

```
#ifndef RATIONAL__
#define RATIONAL__

#include <c._>
```

```
#define RATIONAL_MAX     (1e+37)
#define RATIONAL_MIN (-RATIONAL_MAX)
#define RATIONAL_MIN_ABS (1e-37)

#define Rational_EXTENDS Object,\
 validate, init, compare, copy, read, write,\
 parse, text, decode, encode, add, sub, mul, div

class_ (Rational)
    Float_ value;
fin

#endif
```

A.3.2 Definition

examples/compile/class/Rational.c_

```
#include "Rational._"

 define_ (Rational)
```

A.3.3 validate

examples/include/class/Rational/validate._

```
#ifndef   RATIONAL__VALIDATE__
#define   RATIONAL__VALIDATE__

#include "Rational._"

private inline Bool_ validator(let Float value)
begin
    return value >= RATIONAL_MIN  &&   value <= -RATIONAL_MIN_ABS
    ||     value <= RATIONAL_MAX  &&   value >=  RATIONAL_MIN_ABS;
end

#endif
```

examples/compile/class/Rational/validate.c_

```
#include "Rational/validate._"

procedure_(Bool_, (Rational, validate),
(let Ptr (Rational), this))
    return validator(this->value);
end
```

A.3.4 init

examples/compile/class/Rational/init.c_

```
#include  "Rational/validate._"

procedure_(Rational_ *, (Rational , init),
(let Type , type),
(let Tape , tape))
    Var value = *tape? *(Float *)*tape : 0;
    guard_(validator(value), NULL)
    Var object = new__(Rational);
    guard_(object , NULL)
    object->type  = type ;
    object->value = value;
    return object;
end
```

A.3.5 compare

examples/compile/class/Rational/compare.c_

```
#include "Rational._"

procedure_(LLong_ , (Rational , compare),
(let Ptr (Rational), this),
(let Ptr (Rational), that))
    return this->value < that->value
    ? -1 : this->value > that->value;
end
```

A.3.6 copy

examples/compile/class/Rational/copy.c_

```
#include  "Rational._"

procedure_(Rational_*, (Rational , copy),
(let  Ptr (Rational_), this),
(let  Ptr (Rational ), that))
    Var copy= need__(this);
    guard_(copy , NULL)
    if (!this) copy->type = type_(Rational);
    copy->value = that->value;
    return copy;
end
```

A.3.7 read

examples/compile/class/Rational/read.c_

```
#include "Rational/validate._"

procedure_(Rational_*, (Rational, read),
(let Ptr (Rational_), this),
(let Stream, in))
    return input__(in, this->value) == 1
    && validator(this->value) ? this : NULL;
end
```

A.3.8 write

examples/compile/class/Rational/write.c_

```
#include "Rational._"

procedure_(LLong_, (Rational, write),
(let Ptr (Rational), this),
(let Stream, out))
    return output__(out, this->value);
end
```

A.3.9 parse

examples/compile/class/Rational/parse.c_

```
#include "Rational/validate._"

#include <errno._>

procedure_(Rational_*, (Rational, parse),
(let  Ptr (Rational_), this),
(let  Ptr (Size_), length),
(let  Ptr (Void ), in))
    Void_ *memchr(Void *, Int, Size);
    guard_(memchr(in, '\0', *length), NULL)
    Auto_  endptr_ = (Char_ *)unqual__(in);
    errnum_ = 0;
    Float_ strtof(String, Char_ **);
    Var value = strtof(in, &endptr_);
    guard_((*length = endptr_-(Char *)in) &&  !errnum
    && validator(value), (errnum_ = 0, NULL))
    this->value = value;
    return this;
end
```

A.3.10 text

examples/compile/class/Rational/text.c_

```
#include "Rational._"

procedure_(Void_*, (Rational, text),
(let Ptr (Rational), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    Var buflen = output__((String_){0}, this->value) + 1U;
    Var buf = (out && *length >= buflen)? (String_*)out : new__(Char_[buflen]);
    *length = buflen;
    guard_(buf, NULL)
    output__(*buf, this->value);
    return buf;
end
```

A.3.11 decode

examples/compile/class/Rational/decode.c_

```
#include "Rational/validate._"

#include <errno._>

procedure_(Rational_*, (Rational, decode),
(let Ptr (Rational_), this),
(let Ptr (Size_), length),
(let Ptr (Encoding), in))
    Var dec = solver_(Rational, parse)(this, length, in);
    *length += ! (*in)[*length];
    return dec;
end
```

A.3.12 encode

examples/compile/class/Rational/encode.c_

```
#include "Rational._"

procedure_(Encoding_*, (Rational, encode),
(let Ptr (Rational), this),
(let Ptr (Size_ ), length),
(let Ptr (Encoding_), out))
    Var len = snprintf(NULL, 0, "%a", this->value) + 1U;
    Var enc = (out && len <= *length)? out : new__(UByte_ [len]);
    *length = len;
    guard_(enc, NULL)
    sprintf((Char_ *)*enc, "%a", this->value);
    return enc;
end
```

A.3.13 add

examples/compile/class/Rational/add.c_

```
#include "Rational/validate._"

procedure_(Rational_*, (Rational, add),
(let Ptr (Rational_), sum),
(let Ptr (Rational ), augend),
(let Ptr (Rational ), addend))
    Var value = augend->value + addend->value;
    guard_(validator(value), NULL)
    Var result = need__(sum);
    guard_(result, NULL)
    if (!sum) result->type = type_(Rational);
    result->value = value;
    return result;
end
```

A.3.14 sub

examples/compile/class/Rational/sub.c_

```
#include "Rational/validate._"

procedure_(Rational_*, (Rational, sub),
(let Ptr (Rational_), difference),
(let Ptr (Rational ), minuend),
(let Ptr (Rational ), subtrahend))
    Var value = minuend->value - subtrahend->value;
    guard_(validator(value), NULL)
    Var result = need__(difference);
    guard_(result, NULL)
    if (!difference) result->type = type_(Rational);
    result->value = value;
    return result;
end
```

A.3.15 mul

examples/compile/class/Rational/mul.c_

```
#include "Rational/validate._"

procedure_(Rational_*, (Rational, mul),
(let Ptr (Rational_), product),
(let Ptr (Rational ), multiplier),
(let Ptr (Rational ), multiplicand))
```

```
    Var value  =  multiplier->value * multiplicand->value;
    guard_(validator(value), NULL)
    Var result = need__(product);
    guard_(result, NULL)
    if (!product) result->type = type_(Rational);
    result->value = value;
    return result;
end
```

A.3.16 div

examples/compile/class/Rational/div.c_

```
#include  "Rational/validate._"

procedure_(Rational_*, (Rational, div),
(let  Ptr (Rational_), output),
(let  Ptr (Rational ), dividend),
(let  Ptr (Rational ), divisor))
    guard_(divisor->value , NULL)
    Var value  =  dividend->value / divisor->value;
    guard_(validator(value), NULL)
    Var result = need__(output);
    guard_(result, NULL)
    if (!output) result->type = type_(Rational);
    result->value = value;
    return result;
end
```

A.4 Text

A.4.1 Declaration

examples/include/class/Text._

```
#ifndef TEXT__
#define TEXT__

#include <c._>

#define Text_EXTENDS Object,\
 validate, init, free, compare, copy, read, write,\
 parse, text, decode, encode, add, sub, div

class_ (Text)
    Char_ *buffer;
    Size_  length;
fin

#endif
```

A.4.2 Definition

examples/compile/class/Text.c_

```
#include "Text._"

 define_ (Text)
```

A.4.3 validate

examples/compile/class/Text/validate.c_

```
#include "Text._"

procedure_(Bool_, (Text, validate),
(let Ptr (Text), this))
    Var buffer = this->buffer;
    Var length = this->length;
    guard_((buffer iff length), FALSE_())
    guard_( buffer, TRUE_())
    for (Var_ i_ = (Size)0; i_ < length; i_++)
        guard_((UByte) buffer[i_]  <=  255, FALSE_())
    return ! buffer[length - 1];
end
```

A.4.4 init

examples/compile/class/Text/init.c_

```
#include  "Text._"

procedure_(Text_ *, (Text, init),
(let Type, type),
(let Tape, tape))
    Var object = new__(Text);
    guard_(object, NULL)
    object->type = type;
    Var length = *tape? *(Size *)*tape : 0;
    guard_(object->length = length, (object->buffer = NULL, object))
    guard_(object->length + 1, (free(object), NULL))
    guard_(object->buffer = malloc(object->length + 1), (free(object), NULL))
    object->buffer[0] = '\0';
    object->buffer[length-1] = '\0';
    object->buffer[length  ] = '\0';
    return  object;
end
```

A.4.5 free

examples/compile/class/Text/free.c_

```
#include "Text._"

procedure_((Text, free),
(let Ptr (Text_), this))
    free(this->buffer);
    free(this);
end
```

A.4.6 compare

examples/compile/class/Text/compare.c_

```
#include "Text._"

procedure_(LLong_ , (Text, compare),
(let Ptr (Text), this),
(let Ptr (Text), that))
    Int_   strcmp(String, String);
    return strcmp(this->buffer, that->buffer);
end
```

A.4.7 copy

examples/compile/class/Text/copy.c_

```
#include  "Text._"

procedure_(Text_*, (Text, copy),
(let  Ptr (Text_), this),
(let  Ptr (Text ), that))
    Var copy = need__(this);
    guard_(copy, NULL)
    if (!this) *copy = (Text){.type = type_(Text)};
    if_(this->length < that->length)
        Var    buffer = realloc(copy->buffer , that->length + 1);
        guard_(buffer, (test_(!this, free(copy)), NULL))
        (this->buffer = buffer)[this->length = that->length] = '\0';
    end
    memcpy(this->buffer, that->buffer, that->length);
    return copy;
end
```

A.4.8 read

examples/compile/class/Text/read.c_

```
#include "Text._"

procedure_(Text_*, (Text, read),
(let Ptr (Text_), this),
(let Stream, in))
    let Ptr(Char_ [this->length]) buffer = (Void_ *) this->buffer;
    return input__(in, *buffer)? this : NULL;
end
```

A.4.9 write

examples/compile/class/Text/write.c_

```
#include "Text._"

procedure_(LLong_, (Text, write),
(let Ptr (Text), this),
(let Stream, out))
    return output__(out, this->buffer);
end
```

A.4.10 parse

examples/compile/class/Text/parse.c_

```
#include "Text._"

procedure_(Text_*, (Text, parse),
(let Ptr (Text_), this),
(let Ptr (Size_), length),
(let Ptr (Void ), in))
    Var_ i_ = (Size)0;
    Var text = (Char *)in;
    for (Var len = *length; i_ < len; i_++)
        guard_(text[i_] && (UByte) text[i_] <= 255)
    if_(this->length <= (*length = i_))
        Var buffer = realloc(this->buffer , i_+2);
        guard_(buffer, NULL)
        (this->buffer = buffer)[this->length = i_+1] = '\0';
    end
    Void_ *memmove(Void_ *, Void *, Size);
    memmove(this->buffer, in, i_);
    this->buffer[i_] = '\0';
    return this;
end
```

A.4.11 text

examples/compile/class/Text/text.c_

```
#include "Text._"

procedure_(Void_*, (Text, text),
(let Ptr (Text ), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    guard_(this->buffer, NULL)
    Size_ strlen(String);
    Var buflen = strlen(this->buffer) + 1;
    Var buf = (out && *length >= buflen)? (String_*)out : new__(Char_[buflen]);
    *length = buflen;
    guard_(buf, NULL)
    Void_ *memmove(Void_ *, Void *, Size);
    memmove(*buf, this->buffer, buflen);
    return buf;
end
```

A.4.12 decode

examples/compile/class/Text/decode.c_

```
#include "Text._"

procedure_(Text_*, (Text, decode),
(let Ptr (Text_), this),
(let Ptr (Size_), length),
(let Ptr (Encoding), in))
    Var_ i_ = (Size)0;
    while (i_<*length && (*in)[i_] <= 255) i_++;
    Var nul = i_<*length || (*in)[i_ - 1];
    if_(this->length < (*length = i_)+nul)
        Var    buffer = realloc(this->buffer , i_+nul + 1);
        guard_(buffer, NULL)
        (this->buffer = buffer)[this->length = i_+nul] = '\0';
    end
    memcpy(this->buffer, *in, i_);
    if (nul) this->buffer[i_] = '\0';
    return this;
end
```

A.4.13 encode

examples/compile/class/Text/encode.c_

```
#include "Text._"

procedure_(Encoding_*, (Text, encode),
(let Ptr (Text ), this),
(let Ptr (Size_), length),
(let Ptr (Encoding_), out))
    Var enc = (out&& this->length<=*length)? out : new__(UByte_ [this->length]);
    *length = this->length;
    guard_(enc, NULL)
    if ((Char*)*enc != this->buffer)
        memcpy(*enc , this->buffer, this->length);
    return enc;
end
```

A.4.14 add

examples/compile/class/Text/add.c_

```
#include "Text._"

procedure_(Text_*, (Text, add),
(let Ptr (Text_), sum),
(let Ptr (Text ), augend),
(let Ptr (Text ), addend))
    Var cat = need__(sum);
    guard_(cat, NULL)
    if (!sum) *cat = (Text){.type = type_(Text)};
    Size_ strlen(String);
    Var offset = strlen(augend->buffer);
    Var length = strlen(addend->buffer)+1 + offset;
    if_(cat->length < length)
        Var    buffer = realloc(cat->buffer , length + 1);
        guard_(buffer, (test_(!sum, free(cat)), NULL))
        ( cat->buffer = buffer)[cat->length = length] = '\0';
    end
    Void_ *memmove(Void_ *, Void *, Size);
    (cat != addend ? memcpy : memmove)
    (cat->buffer + offset, addend->buffer, length - offset);
    if (cat != augend) memcpy(cat->buffer, augend->buffer, offset);
    return cat;
end
```

A.4.15 sub

examples/compile/class/Text/sub.c_

```
#include "Text._"

procedure_(Text_*, (Text, sub),
(let Ptr (Text_), difference),
(let Ptr (Text ), minuend),
(let Ptr (Text ), subtrahend))
    Var dif = need__(difference);
    guard_(dif, NULL)
    if (!difference) *dif = (Text){.type = type_(Text)};
    Bool_ found_[256] = {FALSE_()};
    let Size_ i_, j_ = 0, count_ = 1;
    Var min =    minuend->buffer;
    Var sub = subtrahend->buffer;
    Size_ strlen(String);
    for (i_ = strlen(sub); i_--;)
          found_[(UByte) sub[i_]] = TRUE_();
    Var length = strlen(min);
    for (i_ = length; i_--;) count_ +=
        ! found_[(UByte) min[i_]];
    if_(dif->length < count_)
        Var    buffer = realloc(dif->buffer , count_ + 1);
        guard_(buffer, (test_(!difference, free(dif)), NULL))
        ( dif->buffer = buffer)[dif->length = count_] = '\0';
    end
    Var buffer = dif->buffer;
    for (i_ = 0; i_ <= length; i_++)
        if (! found_[(UByte) min[i_]]) buffer[j_++] = min[i_];
    return dif;
end
```

A.4.16 div

examples/compile/class/Text/div.c_

```
#include "Text._"

procedure_(Text_*, (Text, div),
(let Ptr (Text_), result),
(let Ptr (Text ), dividend),
(let Ptr (Text ), divisor))
    Var div = need__(result);
    guard_(div, NULL)
    if (!result) *div = (Text){.type = type_(Text)};
    Size_ strlen(String);
```

```
    Var length = strlen(dividend->buffer)+1;
    if_(div->length < length)
        Var    buffer = realloc(div->buffer , length + 1);
        guard_(buffer, (test_(!result, free(div)), NULL))
        ( div->buffer = buffer)[div->length = length] = '\0';
    end
    Var divider = divisor->buffer;
    Bool_ found_[256] = {FALSE_()};
    for (Var_ i_ = strlen(divider); i_--;)
        found_[(UByte) divider[i_]] = TRUE_();
    Var string = dividend->buffer;
    Var buffer = div->buffer;
    for (let Size_ i_ = 0, j_ = 0; i_ <= length; i_++)
        buffer[j_++] = found_[(UByte) string[i_]]? '\0' : string[i_];
    return div;
end
```

A.5 Iterable

A.5.1 Declaration

examples/include/interface/Iterable._

```
#ifndef ITERABLE__
#define ITERABLE__

#include <c._>

#define Iterable_EXTENDS Abstract,\
 validate, compare, copy, add

Interface_(Iterable)

typedef_  (Iterator, struct Iterator)

prototype_(Bool_, (Iterable, append),
(Iterable *, this), (Void *, data))

prototype_(Size_, (Iterable, count),
(Iterable *, this))

prototype_(Iterator_ *, (Iterable, duplicate),
(Typex (Iterable), typex), (Iterator *, iterator))

prototype_(Void *, (Iterable, get_next),
(Typex (Iterable), typex), (Iterator_*, iterator))
```

```
prototype_(Bool_ , (Iterable, has_next),
(Typex (Iterable), typex), (Iterator *, iterator))

prototype_(Iterator_ *, (Iterable, iterator),
(Iterable *, this))

interface_(Iterable, append, count, duplicate, get_next, has_next, iterator)

#endif
```

A.5.2 Definition

examples/compile/interface/Iterable.c_

```
#include "Iterable._"

 define_ (Iterable)
```

A.5.3 validate

examples/compile/interface/Iterable/validate.c_

```
#include "Iterable._"

procedure_(Bool_, (Iterable, validate),
(let Ptr (Iterable), this))
    Var    typex = concrete_(this)->typex;
    guard_(typex->append
        && typex->count
        && typex->duplicate
        && typex->get_next
        && typex->has_next
        && typex->iterator, FALSE_())
    Var concrete = this->concrete;
    Var_  count_ = typex->count(concrete);
    Var_   more_ = TRUE_();
{   Var iterator = typex->iterator(concrete);
    post_(iterator != NULL);
    Var has_next = typex->has_next;
    Var get_next = typex->get_next;
    for (; (more_ = has_next(typex, iterator)) &&  count_; count_--)
        get_next(typex, iterator);
    free(iterator);
}   return !(more_ ||  count_);
end
```

A.5.4 compare

examples/compile/interface/Iterable/compare.c_

```
#include "Iterable._"

procedure_(LLong_, (Iterable, compare),
(let Ptr (Iterable), this),
(let Ptr (Iterable), that))
    Var this_typex = concrete_(this)->typex;
    Var that_typex = concrete_(that)->typex;
    Var this_count = this_typex->count(this->concrete);
    Var that_count = that_typex->count(that->concrete);
    guard_(this_count == that_count, this_count - that_count)
    guard_(this_count, 0)
    Var this_array = new__(Void *[this_count]);
    post_(this_array != NULL);
{   Var   iterator = this_typex->iterator(this->concrete);
    post_(iterator != NULL);
    Var get_next = this_typex->get_next;
    for (Var_ i_ = this_count; i_--;)
        (*this_array)[i_] = get_next(this_typex, iterator);
    free (iterator);
}{  Var   iterator = that_typex->iterator(that->concrete);
    post_(iterator != NULL);
    Var get_next = that_typex->get_next;
    for_(Var_ count_ = that_count; count_--;)
        Var next = get_next(that_typex, iterator);
        Var_ i_ = count_;
        do  if (next == (*this_array)[i_])
        {   (*this_array)[i_] = (*this_array)[count_];
            break;
        }   while (i_--);
        guard_(i_ + 1, 1)
    end
    free (iterator);
}   return 0;
end
```

A.5.5 copy

examples/compile/interface/Iterable/copy.c_

```
#include  "Iterable._"

procedure_(Iterable_*, (Iterable, copy),
(let  Ptr (Iterable_), this),
(let  Ptr (Iterable ), that))
```

```
    Var typex = concrete_(that)->typex;
    Var iterator = typex->iterator(that->concrete);
    guard_(iterator, NULL)
    Var copy = validate_(this) &&  ! concrete_(this)->typex->count(this->concrete)
    ? this : (Iterable_ *) init__(type_(Iterable), typex);
    guard_(copy, (free(iterator), NULL))
    Var   append = concrete_(copy)->typex->append;
    Var concrete = copy->concrete_;
    Var get_next = typex->get_next;
    for_(Var_ count_ = typex->count(that->concrete); count_--;)
        continue_(append(concrete, get_next(typex, iterator)))
        free_(concrete);
        if (copy == this) copy->concrete = NULL;
        else free(copy);
        free (iterator);
        return NULL;
    end
    free(iterator);
    return copy;
end
```

A.5.6 add

examples/compile/interface/Iterable/add.c_

```
#include  "Iterable._"

procedure_(Iterable_*, (Iterable, add),
(let  Ptr (Iterable_), sum),
(let  Ptr (Iterable ), augend),
(let  Ptr (Iterable ), addend))
    Var aug = concrete_(augend)->typex->iterator(augend->concrete);
    guard_(aug, NULL)
    Var add = concrete_(addend)->typex->iterator(addend->concrete);
    guard_(add, (free(aug), NULL))
    Var cat = validate_(sum)
    &&  sum->concrete != augend->concrete
    &&  sum->concrete != addend->concrete
    ?   sum : (Iterable_ *) init__(type_(Iterable), concrete_(augend)->type);
    guard_(cat, (free(aug), free(add), NULL))
    Var   append = concrete_(cat)->typex->append;
    Var concrete = cat->concrete_;
{   Var    typex = concrete_(augend)->typex;
    Var get_next = typex->get_next;
    for_(Var_ count_ = typex->count(augend->concrete); count_--;)
        continue_(append(concrete, get_next(typex, aug)))
        free_(concrete);
```

```
        if (cat == sum) cat->concrete = NULL;
        else free(cat);
       free(aug);
        free(add);
        return NULL;
    end
}   free(aug);
{   Var typex  = concrete_(addend)->typex;
    Var get_next = typex->get_next;
    for_(Var_ count_ = typex->count(addend->concrete); count_--;)
        continue_(append(concrete, get_next(typex, add)))
        free_(concrete);
        if (cat == sum) cat->concrete = NULL;
        else free(cat);
        free(add);
        return NULL;
    end
}   free(add);
    return cat;
end
```

A.5.7 append

A.5.7.1 Protocol

examples/include/interface/Iterable/append._

```
#ifndef   ITERABLE__APPEND__
#define   ITERABLE__APPEND__

#include "Iterable._"

private
protocol_(Bool_, (Iterable, append),
(let Ptr (Iterable), this),
(let Ptr (Void), data))
    pre_ (validate_(Iterable, this));
    Var count   = concrete_(this)->typex->count;
    Var priori  = count(this->concrete);
    Var success = solver_(Iterable, append)(this, data);
    post_(validate_(Iterable, this));
    post_((!success implies  count(this->concrete) == priori));
    post_ ((success implies  count(this->concrete) == priori+1));
    return  success;
end

#endif
```

A.5.7.2 Procedure

examples/compile/interface/Iterable/append.c_

```
#include "Iterable/append._"

procedure_(Bool_, (Iterable, append),
(let Ptr (Iterable), this),
(let Ptr (Void), data))
    return concrete_(this)->typex->append(this->concrete, data);
end
```

A.5.8 count

A.5.8.1 Protocol

examples/include/interface/Iterable/count._

```
#ifndef   ITERABLE__COUNT__
#define   ITERABLE__COUNT__

#include "Iterable._"

private
protocol_(Size_, (Iterable, count),
(let Ptr (Iterable), this))
    pre_ (validate_(Iterable, this));
    return solver_(Iterable, count)(this);
end

#endif
```

A.5.8.2 Procedure

examples/compile/interface/Iterable/count.c_

```
#include "Iterable/count._"

procedure_(Size_, (Iterable, count),
(let Ptr (Iterable), this))
    return concrete_(this)->typex->count(this->concrete);
end
```

A.5.9 duplicate

A.5.9.1 Protocol

examples/include/interface/Iterable/duplicate._

```
#ifndef   ITERABLE__DUPLICATE__
#define   ITERABLE__DUPLICATE__

#include "Iterable._"

private
protocol_(Iterator_ *, (Iterable, duplicate),
(let Typex (Iterable), typex),
(let Ptr (Iterator), iterator))
    pre_(is_typex(typex));
    pre_(typex->duplicate != NULL);
    pre_(iterator != NULL);
    return solver_(Iterable, duplicate)(typex, iterator);
end

#endif
```

A.5.9.2 Procedure

examples/compile/interface/Iterable/duplicate.c_

```
#include  "Iterable/duplicate._"

procedure_(Iterator_ *, (Iterable, duplicate),
(let Typex(Iterable), typex),
(let Ptr  (Iterator), iterator))
    return typex->duplicate(typex, iterator);
end
```

A.5.10 get_next

A.5.10.1 Protocol

examples/include/interface/Iterable/get_next._

```
#ifndef   ITERABLE__GET_NEXT__
#define   ITERABLE__GET_NEXT__

#include "Iterable._"

private
protocol_(Void *, (Iterable, get_next),
(let Typex(Iterable), typex),
(let Ptr (Iterator_), iterator))
    pre_(is_typex(typex));
    pre_(typex->has_next != NULL);
    pre_(typex->get_next != NULL);
    pre_(iterator != NULL);
```

```
    Var hasnext = typex->has_next(typex, iterator);
    Var    next = solver_(Iterable, get_next)(typex, iterator);
    post_((next implies  hasnext));
    return next;
end

#endif
```

A.5.10.2 Procedure

examples/compile/interface/Iterable/get_next.c_

```
#include  "Iterable/get_next._"

procedure_(Void *, (Iterable, get_next),
(let Typex(Iterable ), typex),
(let Ptr  (Iterator_), iterator))
    return typex->get_next(typex, iterator);
end
```

A.5.11 has_next

A.5.11.1 Protocol

examples/include/interface/Iterable/has_next._

```
#ifndef   ITERABLE__HAS_NEXT__
#define   ITERABLE__HAS_NEXT__

#include "Iterable._"

private
protocol_(Bool_, (Iterable, has_next),
(let Typex (Iterable), typex),
(let Ptr (Iterator), iterator))
    pre_(is_typex(typex));
    pre_(typex->has_next != NULL);
    pre_(iterator != NULL);
    return solver_(Iterable, has_next)(typex, iterator);
end

#endif
```

A.5.11.2 Procedure

examples/compile/interface/Iterable/has_next.c_

```
#include  "Iterable/has_next._"

procedure_(Bool_, (Iterable, has_next),
(let Typex(Iterable), typex),
(let Ptr  (Iterator), iterator))
    return typex->has_next(typex, iterator);
end
```

A.5.12 iterator

A.5.12.1 Protocol

examples/include/interface/Iterable/iterator._

```
#ifndef   ITERABLE__ITERATOR__
#define   ITERABLE__ITERATOR__

#include "Iterable._"

private
protocol_(Iterator_ *, (Iterable, iterator),
(let Ptr (Iterable), this))
    pre_(validate_(Iterable, this));
    return solver_(Iterable, iterator)(this);
end

#endif
```

A.5.12.2 Procedure

examples/compile/interface/Iterable/iterator.c_

```
#include  "Iterable/iterator._"

procedure_(Iterator_ *, (Iterable, iterator),
(let  Ptr (Iterable), this))
    return concrete_(this)->typex->iterator(this->concrete);
end
```

A.6 Collection

A.6.1 Declaration

examples/include/interface/Collection._

```
#ifndef COLLECTION__
#define COLLECTION__
```

```
#include "Iterable._"

#define Collection_EXTENDS Iterable,\
 validate, copy, read, write,\
 parse, text, decode, encode, add

Interface_(Collection)

prototype_(Bool_, (Collection, append),
(Collection *, this), (Void *, data))

prototype_(Type_, (Collection, species),
(Collection *, this), (Type  , species))

interface_(Collection, append, species)

#endif
```

A.6.2 Definition

examples/compile/interface/Collection.c_

```
#include "Collection._"

 define_ (Collection)
```

A.6.3 validate

examples/compile/interface/Collection/validate.c_

```
#include "Collection._"

procedure_(Bool_, (Collection, validate),
(let Ptr (Collection), this))
    Var    typex = concrete_(this)->typex;
    guard_(typex->append &&  typex->species, FALSE_())
    Var concrete = this->concrete;
    Var  species = typex->species(concrete, NULL);
    guard_(is_type(species), FALSE_())
    Var_ valid_ = TRUE_();
{   Var   iterator  = typex->base->iterator(concrete);
    post_(iterator != NULL);
    Var get_next = typex->base->get_next;
    for (Var_ count_ = typex->base->count(concrete); count_--
    &&  (valid_ = validate(species, get_next(typex->base, iterator)));)
    free (iterator);
}   return valid_;
end
```

A.6.4 copy

examples/compile/interface/Collection/copy.c_

```
#include "Collection._"

procedure_(Collection_*, (Collection, copy),
(let  Ptr (Collection_), this),
(let  Ptr (Collection ), that))
    Var copy =
    validate_(this) &&  ! concrete_(this)->typex->base->count(this->concrete)
    ? this : (Collection_ *) init__(type_(Collection), concrete_(that)->type);
    guard_(copy, NULL)
    concrete_(copy)->typex->species(copy->concrete,
    concrete_(that)->typex->species(that->concrete, NULL));
    Var cpy = solver_(Iterable, copy)(copy->base, that->base);
    guard_(cpy, (test_(copy != this, free(copy)), NULL))
    post_ (cpy == copy->base);
    return copy;
end
```

A.6.5 read

examples/compile/interface/Collection/read.c_

```
#include "Collection._"

procedure_(Collection_*, (Collection, read),
(let Ptr  (Collection_), this),
(let Stream, in))
    guard_(validate_(Collection, this), NULL)
    Var concrete = this->concrete_;
    Var    typex = concrete_(this)->typex;
    Var   append = typex->append;
    Var  species = typex->species(concrete, NULL);
    Var     init = species->init;
    Var     read = species->read;
    Var     free = species->free;
{   Var_ object_ = NULL;
    begin
        guard_(object_ ||  (object_ = init(species, (Tape){NULL})))
    {   Var data = read(object_, in);
        guard_(data)
        if (data == object_) object_ = NULL;
        if (append(concrete, data)) continue_(fgetc(in) == '\n')
        else free(data);
    }   break;
    again
    if (object_) free(object_);
}   return this;
end
```

A.6.6 write

examples/compile/interface/Collection/write.c_

```
#include "Collection._"

procedure_(LLong_, (Collection, write),
(let Ptr (Collection), this),
(let Stream, out))
    Var  concrete = this->concrete;
    Var     typex = concrete_(this)->typex->base;
    Var_ written_ = 0LL;
{   Var  iterator = typex->iterator(concrete);
    guard_(iterator, -1)
    Var get_next = typex->get_next;
    for (Var_  count_ = typex->count(concrete); count_--; written_++)
        guard_(write_(get_next(typex, iterator), out) >= 0
        &&  fputc('\n', out) > 0)
    free(iterator);
}   return written_;
end
```

A.6.7 parse

examples/compile/interface/Collection/parse.c_

```
#include  "Collection._"

procedure_(Collection_*, (Collection, parse),
(let  Ptr (Collection_), this),
(let  Ptr (Size_), length),
(let  Ptr (Void), in))
    guard_(validate_(Collection, this), NULL)
    Var concrete = this->concrete_;
    Var    typex = concrete_(this)->typex;
    Var   append = typex->append;
    Var  species = typex->species(concrete, NULL);
    Var     init = species->init;
    Var    parse = species->parse;
    Var     free = species->free;
    Var_  count_ = (Size)0;
{   Var_ object_ = NULL;
    Var text = (Char *)in;
    for_(Var len = *length; text[count_];)
        guard_(object_ ||  (object_ = init(species, (Tape){NULL})))
        Size_ length_  =  len - count_;
    {   Var data = parse(object_, &length_, text + count_);
        Var next  =  (count_ += length_)<len &&  text[count_] == '\n';
```

```
        count_ += next;
        guard_(data)
      if (data == object_) object_ = NULL;
        if (append(concrete, data)) continue_(next)
        else free(data);
    }   break;
    end
    if (object_) free(object_);
}   *length = count_;
    return this;
end
```

A.6.8 text

examples/compile/interface/Collection/text.c_

```
#include "Collection._"

procedure_(Void_*, (Collection, text),
(let  Ptr (Collection), this),
(let  Ptr (Size_), length),
(let  Ptr (Void_), out))
    Var concrete = this->concrete;
    Var    typex = concrete_(this)->typex->base;
    Var    count = typex->count(concrete);
    if_(!count)
        Var text = out? out : malloc(1);
        if (text) *(Char_ *)text = '\0';
        return text;
    end
    Var parts = new__(Char_ * [count]);
    guard_(parts, NULL)
    Var_ len_ = (Size)1;
{   Var iterator = typex->iterator(concrete);
    guard_(iterator, (free(parts), NULL))
    Var get_next = typex->get_next;
    for_(Var_ i_ = (Size)0; i_ < count; i_++)
        Size_ length_ = 1;
        if_(!((*parts)[i_] = text__(get_next(typex, iterator), &length_, NULL)))
            while (i_--) free((*parts)[i_]);
            free(iterator);
            free(parts);
            return NULL;
        end
        len_ += length_;
    end
    free(iterator);
}   Var text = (out &&  *length >= len_)? (String_ *)out : new__(Char_ [len_]);
```

```
    *length = len_;
    if_(text)
        Var_ text_ = *text;
        for_(Var_ i_ = (Size)0; i_ < count; i_++)
            Var_ part_ = (*parts)[i_];
            while ((*text_ = part_++[0])) text_++;
            text_++[0] = '\n';
        end
        *text_ = '\0';
    end
    for (Var_ i_ = count; i_--;) free((*parts)[i_]);
    free(parts);
    return text;
end
```

A.6.9 decode

examples/compile/interface/Collection/decode.c_

```
#include "Collection._"

procedure_(Collection_*, (Collection, decode),
(let  Ptr (Collection_), this),
(let  Ptr (Size_), length),
(let  Ptr (Encoding), in))
    guard_(validate_(Collection, this), NULL)
    Var concrete = this->concrete_;
    Var    typex = concrete_(this)->typex;
    Var   append = typex->append;
    Var  species = typex->species(concrete, NULL);
    Var     init = species->init;
    Var   decode = species->decode;
    Var     free = species->free;
    Var_  count_ = (Size)0;
{   Var_ object_ = NULL;
    for_(Var len = *length - 1; count_ < len;)
        guard_(object_ ||  (object_ = init(species, (Tape){NULL})))
        Size_ length_  =  *length - count_;
    {   Var data = decode(object_, &length_, (Void *)& (*in)[count_]);
        count_ += length_;
        guard_(data)
        if (data == object_) object_ = NULL;
        continue_(append(concrete, data))
        free(data);
    }   break;
    end
    if (object_) free(object_);
}   *length  =  count_ + (count_+1 == *length);
    return this;
end
```

A.6.10 encode

examples/compile/interface/Collection/encode.c_

```
#include "Collection._"

procedure_(Encoding_ *, (Collection , encode),
(let  Ptr (Collection), this),
(let  Ptr (Size_), length),
(let  Ptr (Encoding_) , out))
    Var concrete = this->concrete;
    Var    typex = concrete_(this)->typex->base;
    Var    count = typex->count(concrete);
    if_(!count)
        Var enc = out? out : malloc(1);
        if (enc) *(UByte_ *)enc = '\0';
        return enc;
    end
    Var parts = new__(Encoding_ * [count]);
    guard_(parts, NULL)
    Var sizes = new__(Size_ [count]);
    guard_(sizes, (free(parts), NULL))
    Var_ len_ = (Size)1;
{   Var iterator = typex->iterator(concrete);
    guard_(iterator, (free(parts), free(sizes), NULL))
    Var get_next = typex->get_next;
    for_(Var_ i_ = (Size)0; i_ < count; i_++)
        Size_ length_ = 1;
        if_(!((*parts)[i_] = encode__(get_next(typex, iterator),&length_,NULL)))
            while (i_--) free((*parts)[i_]);
            free(iterator);
            free(sizes);
            free(parts);
            return NULL;
        end
        len_ += ((*sizes)[i_] = length_);
    end
    free(iterator);
}   Var enc = (out &&  *length >= len_)? out : new__(UByte_ [len_]);
    *length = len_;
    if_(enc)
        Var_ enc_ = *enc;
        for_(Var_ i_ = (Size)0; i_ < count; i_++)
            Var size  =  (*sizes)[i_];
            memcpy(enc_, (*parts)[i_], size);
            enc_ += size;
        end
        *enc_ = '\0';
    end
```

```
    for (Var_ i_ = count; i_--;) free((*parts)[i_]);
    free(sizes);
    free(parts);
    return enc ;
end
```

A.6.11 add

examples/compile/interface/Collection/add.c_

```
#include  "Collection._"

procedure_(Collection_*, (Collection , add),
(let  Ptr (Collection_), sum),
(let  Ptr (Collection ), augend),
(let  Ptr (Collection ), addend))
    Var result = validate_(sum)
    &&  sum->concrete != augend->concrete
    &&  sum->concrete != addend->concrete
    ?   sum : (Collection_*) init__(type_(Collection), concrete_(augend)->type);
    guard_(result , NULL)
    concrete_(result)->typex->species(result->concrete , super(
    concrete_(augend)->typex->species(augend->concrete , NULL),
    concrete_(addend)->typex->species(addend->concrete , NULL)));
    Var join = solver_(Iterable , add)(result->base , augend->base , addend->base);
    guard_(join , (test_(result != sum , free(result)), NULL))
    post_ (join == result->base);
    return result;
end
```

A.6.12 append

A.6.12.1 Protocol

examples/include/interface/Collection/append._

```
#ifndef   COLLECTION__APPEND__
#define   COLLECTION__APPEND__

#include "Collection._"

private
protocol_(Bool_ , (Collection , append),
(let Ptr (Collection), this),
(let Ptr (Void), data))
    pre_(validate_(Collection , this));
    Var  species = concrete_(this)->typex->species(this->concrete , NULL);
    pre_(validate (species , data));
    return verifier_(Iterable , append)(_site , this->base , data);
end

#endif
```

A.6.12.2 Procedure

examples/compile/interface/Collection/append.c_

```
#include "Collection/append._"

procedure_(Bool_, (Collection, append),
(let Ptr (Collection), this),
(let Ptr (Void), data))
    return concrete_(this)->typex->append(this->concrete, data);
end
```

A.6.13 species

A.6.13.1 Protocol

examples/include/interface/Collection/species._

```
#ifndef   COLLECTION__SPECIES__
#define   COLLECTION__SPECIES__

#include "Collection._"

private
protocol_(Type_, (Collection, species),
(let Ptr (Collection), this),
(let Type, species))
    pre_(validate_(Collection, this));
    pre_((species implies is_type(species)));
    Var concrete = this->concrete;
    Var    typex = concrete_(this)->typex;
    Var   priori = typex->species(concrete, NULL);
    Var   update = solver_(Collection, species)(this, species);
    post_(is_type(update));
    post_(update == typex->species(concrete, NULL));
    post_((!species implies  update == priori));
    guard_( species, update)
    Var count = typex->base->count(concrete);
    post_((!count implies  update == species));
    post_ ((count implies  update == super(priori, species)));
    return update;
end

#endif
```

A.6.13.2 Procedure

examples/compile/interface/Collection/species.c_

```
#include "Collection/species._"

procedure_(Type_, (Collection, species),
(let Ptr (Collection), this),
(let Type, species))
    return concrete_(this)->typex->species(this->concrete, species);
end
```

A.7 Chain

A.7.1 Declaration

examples/include/class/Chain._

```
#ifndef CHAIN__
#define CHAIN__

#include "Iterable._"

#define Chain_EXTENDS Object,\
 validate, init, free, compare, copy, add

#define Chain_IMPLEMENTS_Iterable SELF,\
 append, count, duplicate, get_next, has_next, iterator

typedef_(Node, struct Node
{   Void  *data;
    struct Node *next;
})

class_ (Chain implements Iterable)
    Node_ *head;
    Size_  length;
    Node_ *tail;
fin

prototype_(Bool_ , (Chain, append),
(Chain_ *, this) , (Void *, data))

prototype_(Size_ , (Chain, count),
(Chain  *, this))

prototype_(Node_*, (Chain, iterator),
(Chain  *, this))

prototype_(Node_*, (Chain, duplicate),
(Typex (Iterable), typex), (Node  *, node))
```

```
prototype_(Void *, (Chain, get_next),
(Typex (Iterable), typex), (Node_ *, node))

prototype_(Bool_ , (Chain, has_next),
(Typex (Iterable), typex), (Node  *, node))

#endif
```

A.7.2 Definition

examples/compile/class/Chain.c_

```
#include "Chain._"

 define_ (Chain)

 define_ (Chain implements Iterable)
```

A.7.3 validate

examples/compile/class/Chain/validate.c_

```
#include "Chain._"

procedure_(Bool_, (Chain, validate),
(let Ptr (Chain), this))
    guard_(this->type->self == this->type, TRUE_())
    Chain_ chain[1] = {*this};
    chain->type = (Type)typex_(Iterable, Chain);
    return solver_(Iterable, validate)(abstract_(Iterable, chain));
end
```

A.7.4 init

examples/compile/class/Chain/init.c_

```
#include  "Chain._"

procedure_(Chain_ *, (Chain, init),
(let Type, type),
(let Tape, tape))
    (Void) tape;
    Var chain = new__(Chain);
    if (chain) *chain = (Chain){.type = type};
    return chain;
end
```

A.7.5 free

examples/compile/class/Chain/free.c_

```
#include "Chain._"

procedure_((Chain, free),
(let Ptr (Chain_), this))
    if (!validate_(this)) this->head = NULL;
    for (let Ptr_(Node_) node_,
    next_ = this->head; (node_ = next_); free(node_))
        next_ = node_->next;
    free(this);
end
```

A.7.6 compare

examples/compile/class/Chain/compare.c_

```
#include "Chain._"

procedure_(LLong_, (Chain, compare),
(let Ptr (Chain) , this),
(let Ptr (Chain) , that))
    Chain_ _this[1] = {*this};
    Chain_ _that[1] = {*that};
    _this->type = (Type)typex_(Iterable, Chain);
    _that->type = (Type)typex_(Iterable, Chain);
    return solver_(Iterable, compare)
        (abstract_(Iterable,_this),
         abstract_(Iterable,_that));
end
```

A.7.7 copy

examples/compile/class/Chain/copy.c_

```
#include  "Chain._"

procedure_(Chain_*, (Chain, copy),
(let  Ptr (Chain_), this),
(let  Ptr (Chain ), that))
    Var copy = (validate_(this) &&  ! this->length)? this : new__(*this);
    guard_(copy, NULL)
    if (copy != this) *copy = (Chain){.type = type_(Chain)};
    Chain_ _src[1] = {*that};
    _src->type = (Type)typex_(Iterable, Chain);
    Var   type = copy->type;
```

```
    copy->type = (Type)typex_(Iterable, Chain);
    Var it
    =  solver_(Iterable, copy)
   (abstract_(Iterable, copy),
     abstract_(Iterable, _src));
    copy->type = type;
    guard_(it, (test_(copy != this, free(copy)), NULL))
    post_ (it->concrete == copy);
    return copy;
end
```

A.7.8 add

examples/compile/class/Chain/add.c_

```
#include  "Chain._"

procedure_(Chain_*, (Chain, add),
(let  Ptr (Chain_), sum),
(let  Ptr (Chain ), augend),
(let  Ptr (Chain ), addend))
    Var join = validate_(sum)? sum : new__(*sum);
    guard_(join, NULL)
    if (join != sum) *join = (Chain){.type = type_(Chain)};
    Chain_ _augend[1] = {*augend};
    Chain_ _addend[1] = {*addend};
    _augend->type = (Type)typex_(Iterable, Chain);
    _addend->type = (Type)typex_(Iterable, Chain);
    Var   type = join->type;
    join->type = (Type)typex_(Iterable, Chain);
    Var it
    =  solver_(Iterable, add)
    (abstract_(Iterable, join),
     abstract_(Iterable,_augend),
     abstract_(Iterable,_addend));
    join->type = type;
    guard_(it, (test_(join != sum, free(join)), NULL))
    post_ (it->concrete == join);
    return join;
end
```

A.7.9 append

A.7.9.1 Protocol

examples/include/class/Chain/append._

```
#ifndef   CHAIN__APPEND__
#define   CHAIN__APPEND__

#include "Chain._"

protocol_( Bool_ , (Chain, append),
(let Ptr (Chain_), this),
(let Ptr ( Void ), data))
    pre_(validate_(Chain, this));
    Var   type = this->type;
    this->type = (Type)typex_(Iterable, Chain);
    Var success= verifier_(Iterable, append)
    (_site, abstract_(Iterable, this), data);
    this->type = type;
    post_((success implies  data == this->tail->data));
    return success;
end

#endif
```

A.7.9.2 Procedure

examples/compile/class/Chain/append.c_

```
#include "Chain/append._"

procedure_(Bool_ , (Chain, append),
(let Ptr (Chain_), this),
(let Ptr ( Void ), data))
    Var node = new__(Node);
    guard_(node, FALSE_())
    node->data = data;
    node->next = NULL;
    if  (this->length++) this->tail->next = node;
    else this->head = node;
    this->tail = node;
    return TRUE_();
end
```

A.7.10 count

A.7.10.1 Protocol

examples/include/class/Chain/count._

```
#ifndef   CHAIN__COUNT__
#define   CHAIN__COUNT__
```

```
#include "Chain._"

protocol_(Size_ , (Chain, count),
(let Ptr (Chain), this))
    pre_(validate_(Chain, this));
    Var count = solver_(Chain, count)(this);
    post_ (count == this->length);
    return count;
end

#endif
```

A.7.10.2 Procedure

examples/compile/class/Chain/count.c_

```
#include "Chain/count._"

procedure_(Size_, (Chain, count),
(let Ptr (Chain), this))
    return this->length;
end
```

A.7.11 duplicate

A.7.11.1 Protocol

examples/include/class/Chain/duplicate._

```
#ifndef   CHAIN__DUPLICATE__
#define   CHAIN__DUPLICATE__

#include "Chain._"

protocol_(Node_*, (Chain, duplicate),
(let Typex (Iterable), typex),
(let Ptr (Node), node))
    pre_ (node != NULL);
    return solver_(Chain, duplicate)(typex, node);
end

#endif
```

A.7.11.2 Procedure

examples/compile/class/Chain/duplicate.c_

```
#include "Chain/duplicate._"

procedure_(Node_ *, (Chain, duplicate),
(let Typex (Iterable), typex),
(let Ptr  (Node), node))
    (Void) typex;
    Var dup = new__(Node);
    guard_(dup, NULL)
    dup->data =  node->data;
    dup->next = (node->next != node)? node->next : dup;
    return dup;
end
```

A.7.12 get_next

A.7.12.1 Protocol

examples/include/class/Chain/get_next._

```
#ifndef   CHAIN__GET_NEXT__
#define   CHAIN__GET_NEXT__

#include "Chain._"

protocol_(Void *, (Chain, get_next),
(let Typex (Iterable), typex),
(let Ptr (Node_), node))
    pre_ (node != NULL);
    return solver_(Chain, get_next)(typex, node);
end

#endif
```

A.7.12.2 Procedure

examples/compile/class/Chain/get_next.c_

```
#include "Chain/get_next._"

procedure_(Void *, (Chain, get_next),
(let Typex (Iterable), typex),
(let Ptr (Node_), node))
    (Void) typex;
    Var data = node->data;
    *node = node->next? * node->next : (Node){.next = node};
    return data;
end
```

A.7.13 has_next

A.7.13.1 Protocol

examples/include/class/Chain/has_next._

```
#ifndef   CHAIN__HAS_NEXT__
#define   CHAIN__HAS_NEXT__

#include "Chain._"

protocol_(Bool_, (Chain, has_next),
(let Typex (Iterable), typex),
(let Ptr (Node), node))
    pre_ (node != NULL);
    return solver_(Chain, has_next)(typex, node);
end

#endif
```

A.7.13.2 Procedure

examples/compile/class/Chain/has_next.c_

```
#include "Chain/has_next._"

procedure_(Bool_, (Chain, has_next),
(let Typex (Iterable), typex),
(let Ptr (Node), node))
    (Void) typex;
    return node->next != node;
end
```

A.7.14 iterator

A.7.14.1 Protocol

examples/include/class/Chain/iterator._

```
#ifndef   CHAIN__ITERATOR__
#define   CHAIN__ITERATOR__

#include "Chain._"

protocol_(Node_*, (Chain, iterator),
(let Ptr (Chain), this))
    pre_(validate_(Chain, this));
    Var node = solver_(Chain, iterator)(this);
    guard_(node, NULL)
```

```
    Var head = this->head;
    post_ (node != head);
    post_((!head implies  node->data == NULL));
    post_((!head implies  node->next == node));
    post_(( head implies  node->data == head->data));
    post_(( head implies  node->next == head->next));
    return node;
end

#endif
```

A.7.14.2 Procedure

examples/compile/class/Chain/iterator.c_

```
#include "Chain/iterator._"

procedure_(Node_ *, (Chain, iterator),
(let Ptr (Chain), this))
    Var node = new__(Node);
    guard_(node, NULL)
    *node = this->length? * this->head : (Node){.next = node};
    return node;
end
```

A.8 List

A.8.1 Declaration

examples/include/class/List._

```
#ifndef LIST__
#define LIST__

#include "Chain._"
#include "Collection._"

#define List_EXTENDS Chain,\
 validate, init, copy, read, write,\
 parse, text, decode, encode, add

#define List_IMPLEMENTS_Collection Chain,\
 append, species

class_ (List implements Collection)
    Type_ species;
fin
```

```
prototype_(Bool_, (List, append),
(List_ *, this), (Void *, data))

prototype_(Type_, (List, species),
(List_ *, this), (Type, species))

#endif
```

A.8.2 Definition

examples/compile/class/List.c_

```
#include "List._"

 define_ (List)

 define_ (List implements Collection)
```

A.8.3 validate

examples/compile/class/List/validate.c_

```
#include "List._"

procedure_(Bool_, (List, validate),
(let Ptr (List), this))
    guard_(this->type->self == this->type, TRUE_())
    List_ list[1] = {*this};
    list->type = (Type)typex_(Collection, List);
    return solver_(Collection, validate)(abstract_(Collection, list));
end
```

A.8.4 init

examples/compile/class/List/init.c_

```
#include  "List._"

procedure_(List_ *, (List, init),
(let Type, type),
(let Tape, tape))
    Var species = (Type *)*tape;
    guard_((species implies  is_type(*species)), NULL)
    Var list  =  new__(List);
    if (list) *list = (List)
    {   .type    = type,
        .species = species? *species : type_(Object),
    };
    return list;
end
```

A.8.5 copy

examples/compile/class/List/copy.c_

```
#include "List._"

procedure_(List_*, (List, copy),
(let Ptr (List_), this),
(let Ptr (List ), that))
    Var copy = (validate_(this) && ! this->base->length)? this : new__(*this);
    guard_(copy, NULL)
    if (copy != this) *copy = (List){.type = type_(List)};
    Var cpy = solver_(Chain, copy)(copy->base, that->base);
    guard_(cpy, (test_(copy != this, free(copy)), NULL))
    post_ (cpy == copy->base);
    copy->species = that->species;
    return copy;
end
```

A.8.6 read

examples/compile/class/List/read.c_

```
#include "List._"

procedure_(List_*, (List, read),
(let Ptr (List_), this),
(let Stream, in))
    Var   type = this->type;
    this->type = (Type)typex_(Collection, List);
    Var    col = solver_(Collection, read)(abstract_(Collection, this), in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.8.7 write

examples/compile/class/List/write.c_

```
#include "List._"

procedure_(LLong_, (List, write),
(let Ptr (List), this),
(let Stream, out))
    List_ list[1] = {*this};
    list->type = (Type)typex_(Collection, List);
    return solver_(Collection, write)(abstract_(Collection, list), out);
end
```

A.8.8 parse

examples/compile/class/List/parse.c_

```
#include "List._"

procedure_(List_*, (List, parse),
(let Ptr (List_), this),
(let Ptr (Size_), len),
(let Ptr (Void ), in))
    Var  type = this->type;
    this->type = (Type)typex_(Collection, List);
    Var col = solver_(Collection, parse)(abstract_(Collection, this), len, in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.8.9 text

examples/compile/class/List/text.c_

```
#include "List._"

procedure_(Void_*, (List, text),
(let Ptr (List ), this),
(let Ptr (Size_), length),
(let Ptr (Void_), out))
    List_ list[1] = {*this};
    list->type = (Type)typex_(Collection, List);
    return solver_(Collection, text)(abstract_(Collection, list), length, out);
end
```

A.8.10 decode

examples/compile/class/List/decode.c_

```
#include "List._"

procedure_(List_*, (List, decode),
(let Ptr (List_), this),
(let Ptr (Size_), len),
(let Ptr (Encoding), in))
    Var  type = this->type;
    this->type = (Type)typex_(Collection, List);
    Var col = solver_(Collection, decode)(abstract_(Collection, this), len, in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.8.11 encode

examples/compile/class/List/encode.c_

```
#include  "List._"

procedure_(Encoding_*, (List, encode),
(let  Ptr (List ), this),
(let  Ptr (Size_), length),
(let  Ptr (Encoding_), out))
    List_ list[1] = {*this};
    list->type = (Type)typex_(Collection, List);
    return solver_(Collection,encode)(abstract_(Collection, list), length, out);
end
```

A.8.12 add

examples/compile/class/List/add.c_

```
#include  "List._"

procedure_(List_*, (List, add),
(let  Ptr (List_), sum),
(let  Ptr (List ), augend),
(let  Ptr (List ), addend))
    Var join = validate_(sum)? sum : new__(*sum);
    guard_(join, NULL)
    if (join != sum) *join = (List){.type = type_(List)};
    Var cat = solver_(Chain, add)(join->base, augend->base, addend->base);
    guard_(cat, (test_(join != sum, free(join)), NULL))
    post_ (cat == join->base);
    Var   species = super(augend->species, addend->species);
    join->species = join->species? super(join->species, species) : species;
    return join;
end
```

A.8.13 append

A.8.13.1 Protocol

examples/include/class/List/append._

```
#ifndef   LIST__APPEND__
#define   LIST__APPEND__

#include "List._"

protocol_(Bool_ , (List, append),
(let Ptr (List_), this),
(let Ptr (Void ), data))
```

```
    pre_ (validate_(List, this));
    Var   type = this->type;
    this->type = (Type)typex_(Collection, List);
    Var success= verifier_(Collection, append)
    (_site, abstract_(Collection, this), data);
    this->type = type;
    post_((success implies  data == this->base->tail->data));
    return success;
end

#endif
```

A.8.13.2 Procedure

examples/compile/class/List/append.c_

```
#include "List/append._"

procedure_(Bool_, (List, append),
(let Ptr (List_), this),
(let Ptr (Void ), data))
    Bool_  solver_(Chain, append)(Chain_ *, Void *);
    return solver_(Chain, append)(this->base, data);
end
```

A.8.14 species

A.8.14.1 Protocol

examples/include/class/List/species._

```
#ifndef   LIST__SPECIES__
#define   LIST__SPECIES__

#include "List._"

protocol_(Type_ , (List, species),
(let Ptr (List_), this),
(let Type, species))
    pre_(validate_(List, this));
    Var   type = this->type;
    this->type = (Type)typex_(Collection, List);
    Var update = verifier_(Collection, species)
    (_site, abstract_(Collection, this), species);
    this->type = type;
    post_ (update == this->species);
    return update;
end

#endif
```

A.8.14.2 Procedure

examples/compile/class/List/species.c_

```
#include "List._"

procedure_(Type_, (List, species),
(let Ptr (List_), this),
(let Type, species))
    guard_(species, this->species)
    return this->species = this->base->length?
     super(this->species, species) : species;
end
```

A.9 Vector

A.9.1 Declaration

examples/include/class/Vector._

```
#ifndef VECTOR__
#define VECTOR__

#include "Collection._"

#define Vector_EXTENDS Object,\
 validate, init, free, compare, copy, read,\
 write, parse, text, decode, encode, add

#define Vector_IMPLEMENTS_Iterable SELF,\
 append, count, duplicate, get_next, has_next, (iterator, cursor)

#define Vector_IMPLEMENTS_Collection Vector,\
 append, species

class_ (Vector implements Iterable, Collection)
    Void  **array;
    Size_ capacity, count;
    Type_ species;
fin

typedef_(Cursor, struct Cursor
{   Vector *vector;
    Size_   index ;
})
```

```
prototype_(Bool_ , (Vector, append),
(Vector_ *, this), (Void *, data))

prototype_(Size_ , (Vector, count),
(Vector  *, this))

prototype_(Cursor_ *, (Vector, cursor),
(Vector  *, this))

prototype_(Cursor_ *, (Vector, duplicate),
(Typex (Iterable), typex), (Cursor  *, cursor))

prototype_(Void *, (Vector, get_next),
(Typex (Iterable), typex), (Cursor_ *, cursor))

prototype_(Bool_ , (Vector, has_next),
(Typex (Iterable), typex), (Cursor  *, cursor))

prototype_(Type_ , (Vector, species),
(Vector_ *, this), (Type  , species))

#endif
```

A.9.2 Definition

examples/compile/class/Vector.c_

```
#include "Vector._"

 define_ (Vector)

 define_ (Vector implements Iterable, Collection)
```

A.9.3 validate

examples/compile/class/Vector/validate.c_

```
#include "Vector._"

procedure_(Bool_, (Vector, validate),
(let Ptr (Vector), this))
    guard_(this->array , FALSE_())
    guard_(this->count <= this->capacity, FALSE_())
    guard_(this->type->self == this->type, TRUE_())
    guard_(this->species, TRUE_())
    Vector_ vector[1] = {*this};
    vector->type = (Type)typex_(Collection, Vector);
    return solver_(Collection, validate)(abstract_(Collection, vector));
end
```

A.9.4 init

examples/compile/class/Vector/init.c_

```
#include  "Vector._"

procedure_(Vector_ *, (Vector, init),
(let Type, type),
(let Tape, tape))
    Var species = (Type *)*tape;
    guard_(((species &&  *species) implies  is_type(*species)), NULL)
    Var capacity = (species &&  tape[1])? *(Size *) tape[1] : BUFSIZ;
    guard_(capacity >= 0, NULL)
    Var vector = new__(Vector);
    guard_(vector, NULL)
    guard_(vector->array = *new__(Void * [capacity]), (free(vector), NULL))
    vector->type = type;
    vector->capacity = capacity;
    vector->count = 0;
    vector->species = species? *species : NULL;
    return vector;
end
```

A.9.5 free

examples/compile/class/Vector/free.c_

```
#include "Vector._"

procedure_((Vector, free),
(let Ptr (Vector_), this))
    free(this->array);
    free(this);
end
```

A.9.6 compare

examples/compile/class/Vector/compare.c_

```
#include "Vector._"

procedure_(LLong_, (Vector, compare),
(let Ptr (Vector), this),
(let Ptr (Vector), that))
    Vector_ _this[1] = {*this};
    Vector_ _that[1] = {*that};
    _this->type = (Type)typex_(Iterable, Vector);
    _that->type = (Type)typex_(Iterable, Vector);
```

```
    return solver_(Iterable, compare)
        (abstract_(Iterable,_this),
         abstract_(Iterable,_that));
end
```

A.9.7 copy

examples/compile/class/Vector/copy.c_

```
#include  "Vector._"

procedure_(Vector_*, (Vector, copy),
(let  Ptr (Vector_), this),
(let  Ptr (Vector ), that))
    Var copy = validate_(this)? this : solver_(Vector, init)
    (type_(Vector), (Tape){&(Type){NULL}, &this->count});
    guard_(copy, NULL)
    copy->count = 0;
    copy->species = that->species;
    Vector_ _src[1] = {*that};
    _src->type = (Type)typex_(Iterable, Vector);
    Var   type = copy->type;
    copy->type = (Type)typex_(Iterable, Vector);
    Var it
    =  solver_(Iterable, copy)
    (abstract_(Iterable, copy),
     abstract_(Iterable, _src));
    copy->type = type;
    guard_(it, (test_(copy != this, free(copy)), NULL))
    post_ (it->concrete == copy);
    return copy;
end
```

A.9.8 read

examples/compile/class/Vector/read.c_

```
#include  "Vector._"

procedure_(Vector_*, (Vector, read),
(let Ptr  (Vector_), this),
(let Stream, in))
    Var   type = this->type;
    this->type = (Type)typex_(Collection, Vector);
    Var    col = solver_(Collection, read)(abstract_(Collection, this), in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.9.9 write

examples/compile/class/Vector/write.c_

```
#include "Vector._"

procedure_(LLong_, (Vector, write),
(let Ptr (Vector), this),
(let Stream, out))
    guard_(this->species, -1)
    Vector_ list[1] = {*this};
    list->type = (Type)typex_(Collection, Vector);
    return solver_(Collection, write)(abstract_(Collection, list), out);
end
```

A.9.10 parse

examples/compile/class/Vector/parse.c_

```
#include  "Vector._"

procedure_(Vector_*, (Vector, parse),
(let  Ptr (Vector_), this),
(let  Ptr (Size_), len),
(let  Ptr (Void ), in))
    Var   type = this->type;
    this->type = (Type)typex_(Collection, Vector);
    Var col = solver_(Collection, parse)(abstract_(Collection, this), len, in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.9.11 text

examples/compile/class/Vector/text.c_

```
#include  "Vector._"

procedure_(Void_*, (Vector, text),
(let  Ptr (Vector), this),
(let  Ptr (Size_), length),
(let  Ptr (Void_), out))
    guard_(this->species, NULL)
    Vector_ list[1] = {*this};
    list->type = (Type)typex_(Collection, Vector);
    return solver_(Collection, text)(abstract_(Collection, list), length, out);
end
```

A.9.12 decode

examples/compile/class/Vector/decode.c_

```
#include "Vector._"

procedure_(Vector_*, (Vector, decode),
(let Ptr (Vector_), this),
(let Ptr (Size_), len),
(let Ptr (Encoding), in))
    Var  type = this->type;
    this->type = (Type)typex_(Collection, Vector);
    Var col = solver_(Collection, decode)(abstract_(Collection, this), len, in);
    this->type = type;
    return col? col->concrete_ : NULL;
end
```

A.9.13 encode

examples/compile/class/Vector/encode.c_

```
#include "Vector._"

procedure_(Encoding_*, (Vector, encode),
(let Ptr (Vector), this),
(let Ptr (Size_ ), length),
(let Ptr (Encoding_), out))
    guard_(this->species, NULL)
    Vector_ list[1] = {*this};
    list->type = (Type)typex_(Collection, Vector);
    return solver_(Collection,encode)(abstract_(Collection, list), length, out);
end
```

A.9.14 add

examples/compile/class/Vector/add.c_

```
#include "Vector._"

procedure_(Vector_*, (Vector, add),
(let Ptr (Vector_), sum),
(let Ptr (Vector ), augend),
(let Ptr (Vector ), addend))
    guard_(augend->count  <=  SIZE_MAX - addend->count, NULL)
    Var join = validate_(sum)? sum : solver_(Vector, init)(type_(Vector),
    (Tape){ &(Type){NULL}, &(Size){augend->count + addend->count} });
    guard_(join, NULL)
    Var  species = super(augend->species, addend->species);
```

```
    join->species = join->species? super(join->species, species) : species;
    Vector_ _augend[1] = {*augend};
    Vector_ _addend[1] = {*addend};
    _augend->type = (Type)typex_(Iterable, Vector);
    _addend->type = (Type)typex_(Iterable, Vector);
    Var   type = join->type;
    join->type = (Type)typex_(Iterable, Vector);
    Var it
    =  solver_(Iterable, add)
    (abstract_(Iterable, join),
     abstract_(Iterable,_augend),
     abstract_(Iterable,_addend));
    join->type = type;
    guard_(it, (test_(join != sum, free(join)), NULL))
    post_ (it->concrete == join);
    return join;
end
```

A.9.15 append

A.9.15.1 Protocol

examples/include/class/Vector/append._

```
#ifndef   VECTOR__APPEND__
#define   VECTOR__APPEND__

#include "Vector._"

protocol_(Bool_, (Vector, append),
(let Ptr (Vector_), this),
(let Ptr (Void), data))
    pre_(validate_(Vector, this));
    pre_((this->species implies  validate(this->species, data)));
    Var   type = this->type;
    this->type = (Type)typex_(Iterable, Vector);
    Var success= verifier_(Iterable, append)
    (_site, abstract_(Iterable, this), data);
    this->type = type;
    post_((success implies  data == this->array[this->count - 1]));
    return success;
end

#endif
```

A.9.15.2 Procedure

examples/compile/class/Vector/append.c_

```
#include "Vector/append._"

procedure_(Bool_, (Vector, append),
(let Ptr (Vector_), this),
(let Ptr (Void), data))
    Var arr = this->array;
    Var cap = this->capacity;
    stop_(this->count < cap, (arr[this->count++] = data, TRUE_()))
    Var_ arr_ = NULLPTR;
    Var_ ext_ = (cap <= SIZE_MAX-cap)?  cap : SIZE_MAX-cap;
    while (ext_ &&  !(arr_ = realloc(arr, cap + ext_))) ext_ >>= 1;
    guard_(ext_, FALSE_())
    this->capacity += ext_;
    this->count++[this->array = arr_] = data;
    return TRUE_();
end
```

A.9.16 count

A.9.16.1 Protocol

examples/include/class/Vector/count._

```
#ifndef   VECTOR__COUNT__
#define   VECTOR__COUNT__

#include "Vector._"

protocol_(Size_, (Vector, count),
(let Ptr (Vector), this))
    pre_(validate_(Vector, this));
    Var count = solver_(Vector, count)(this);
    post_ (count == this->count);
    return count;
end

#endif
```

A.9.16.2 Procedure

examples/compile/class/Vector/count.c_

```
#include "Vector/count._"

procedure_(Size_ , (Vector, count),
(let Ptr (Vector), this))
    return this->count;
end
```

A.9.17 cursor

A.9.17.1 Protocol

examples/include/class/Vector/cursor._

```
#ifndef   VECTOR__CURSOR__
#define   VECTOR__CURSOR__

#include "Vector._"

protocol_(Cursor_ *, (Vector, cursor),
(let Ptr (Vector), this))
    pre_(validate_(Vector, this));
    Var    cursor = solver_(Vector, cursor)(this);
    guard_(cursor, NULL)
    post_ (cursor->vector == this);
    post_ (cursor->index  == 0);
    return cursor;
end

#endif
```

A.9.17.2 Procedure

examples/compile/class/Vector/cursor.c_

```
#include "Vector/cursor._"

procedure_(Cursor_ *, (Vector, cursor),
(let Ptr (Vector), this))
    Var cursor = new__(Cursor);
    if (cursor) *cursor = (Cursor){.vector = this};
    return cursor;
end
```

A.9.18 duplicate

A.9.18.1 Protocol

examples/include/class/Vector/duplicate._

```
#ifndef   VECTOR__DUPLICATE__
#define   VECTOR__DUPLICATE__
```

```
#include "Vector._"

protocol_(Cursor_ *, (Vector, duplicate),
(let Typex (Iterable), typex),
(let Ptr (Cursor), cursor))
    pre_ (cursor != NULL);
    return solver_(Vector, duplicate)(typex, cursor);
end

#endif
```

A.9.18.2 Procedure

examples/compile/class/Vector/duplicate.c_

```
#include "Vector/duplicate._"

procedure_(Cursor_ *, (Vector, duplicate),
(let Typex(Iterable), typex),
(let Ptr  (Cursor), cursor))
    (Void) typex;
    Var dup = new__ (Cursor);
    if (dup) *dup = *cursor ;
    return dup;
end
```

A.9.19 get_next

A.9.19.1 Protocol

examples/include/class/Vector/get_next._

```
#ifndef   VECTOR__GET_NEXT__
#define   VECTOR__GET_NEXT__

#include "Vector._"

protocol_(Void *, (Vector, get_next),
(let Typex (Iterable), typex),
(let Ptr (Cursor_), cursor))
    pre_ (cursor != NULL);
    return solver_(Vector, get_next)(typex, cursor);
end

#endif
```

A.9.19.2 Procedure

examples/compile/class/Vector/get_next.c_

```
#include "Vector/get_next._"

procedure_(Void *, (Vector, get_next),
(let Typex (Iterable), typex),
(let Ptr (Cursor_), cursor))
    (Void) typex;
    Var vector = cursor->vector;
    return cursor->index < vector->count ?
    vector->array[cursor->index++] : NULL;
end
```

A.9.20 has_next

A.9.20.1 Protocol

examples/include/class/Vector/has_next._

```
#ifndef   VECTOR__HAS_NEXT__
#define   VECTOR__HAS_NEXT__

#include "Vector._"

protocol_(Bool_, (Vector, has_next),
(let Typex (Iterable), typex),
(let Ptr (Cursor), cursor))
    pre_ (cursor != NULL);
    return solver_(Vector, has_next)(typex, cursor);
end

#endif
```

A.9.20.2 Procedure

examples/compile/class/Vector/has_next.c_

```
#include "Vector/has_next._"

procedure_(Bool_, (Vector, has_next),
(let Typex (Iterable), typex),
(let Ptr (Cursor), cursor))
    (Void) typex;
    return cursor->index < cursor->vector->count;
end
```

A.9.21 species

A.9.21.1 Protocol

examples/include/class/Vector/species._

```
#ifndef   VECTOR__SPECIES__
#define   VECTOR__SPECIES__

#include "Vector._"

protocol_(Type_, (Vector, species),
(let Ptr (Vector_), this),
(let Type, species))
    pre_(validate_(Vector, this));
    pre_(this->species != NULL);
    Var   type = this->type;
    this->type = (Type)typex_(Collection, Vector);
    Var update = verifier_(Collection, species)
    (_site, abstract_(Collection, this), species);
    this->type = type;
    post_ (update == this->species);
    return update;
end

#endif
```

A.9.21.2 Procedure

examples/compile/class/Vector/species.c_

```
#include "Vector._"

procedure_(Type_, (Vector, species),
(let Ptr (Vector_), this),
(let Type, species))
    guard_(species, this->species)
    return this->species = this->count?
     super(this->species, species) : species;
end
```

Appendix B

Naming

The reference implementation uses name mangling to generate various identifiers associated with methods and object-oriented types. It is also important to discuss how the name mangling scheme works, for two primary reasons: firstly, the mangled names are used as identifiers with external linkage, so other implementations need to follow the same scheme for portability; secondly, some of the mangled names are used as function identifiers, and consequently, the predefined identifier `__func__` stores the mangled name. `__func__` is used as an initializer by `SITE`, which is in turn used by several diagnostic features, including the `pre_` and `post_` families. When a pre-condition or post-condition is violated, it is the mangled function name that gets printed in the error message. Hence it is important to know the name mangling scheme for identifying which function a mangled name refers to.

Name mangling is nothing but pasting together two identifiers with a "gluing text" inserted between them. This technique is well-known and employed in many other programming languages. For example, Python uses name mangling for class members that are named with multiple leading underscores and not ending with multiple trailing underscores, such as `__dob`; if the member `__dob` is an attribute of class `Person`, then the actual identifier used is `_Person__dob`. C_ is influenced by this scheme, though it uses name mangling for a different purpose: emulating the functionality of namespaces. However, name mangling by itself cannot capture all the traits of a proper namespace, and therefore we refer to it as pseudo-namespace. For instance, multiple classes and interfaces can have methods with the same name, since the actual identifier being used is formed by prefixing the class or interface name.

For this scheme to work as expected, it is necessary that two distinct pairs of `(`$prefix$`,` *name*`)` should not generate the same identifier; in other words, the mapping must be one-to-one. The so-called "gluing text" inserted between $prefix$ and *name* is intended to prevent name collision, and one of the naming restrictions discussed in the introductory chapter requires that programmers should not declare identifiers with multiple underscores. Also, $prefix$ and *name* should not have leading or trailing underscores (recall that leading underscores have restricted usage in C as well). For example, if the consider the naming scheme tabulated below, then `property_(Prefix_, name)` and `property_(Prefix, _name)` both generate the same identifier `Prefix___name`, leading to name collision.

Assuming that naming restrictions on the use of underscores are respected by the programmer, the following name mangling scheme should work as expected in creating pseudo-namespaces (mostly for object-oriented types).

	Source text	Mangled name
`method_`	($prefix$, $name$)	$prefix$`__2`$name$
`Method_`	($prefix$, $name$)	$prefix$`__3`$name$
`proxy_`	($prefix$, $name$)	$prefix$`__4`$name$
`solver_`	($prefix$, $name$)	$prefix$`__5`$name$
`verifier_`	($prefix$, $name$)	$prefix$`__6`$name$
`property_`	($prefix$, $name$)	$prefix$`__`$name$
`Typex`	($interface$)	$interface$`__0`
`Typex_`	($interface$)	$interface$`__0_`

Recommended practice

For maximum portability, the number of combined characters in $prefix$ and $name$ should be less than 29 for method names, and should not exceed 29 for property names. This is because the mangled names are used as identifiers with external linkage for which implementations can consider only the first 31 characters; in other words, if two distinct identifiers are identical in their initial 31 characters, then they may be considered as same. Hence it is recommended that the full length of mangled name (with the gluing underscores) should not exceed 31.

Appendix C

Limits

The ellipsis framework provides function-like macros for performing various arithmetic, logical, and relational operations on non-negative integers. These features are modeled after machine instructions in a physical processing unit, and like any other translator, the ellipsis framework can operate only on a limited set of values. This limit is determined by the macro `PP_MAX`, which the reference implementation defines as 127; this value was chosen because C compilers can limit the maximum number of arguments permitted in a function-like macro invocation, and this upper limit must be at least 127 (for all conforming compilers). The C standard does not encourage the use of fixed translation limits, and most preprocessors have a reasonably larger limit on the number of macro arguments (it should be acknowledged that an upper limit is inevitable due to the finiteness of memory address space).

`PP_MAX` is required to be at least 127, and for maximum portability across implementations, its value should not be changed. However, C (and thus C_) programs can be non-portable, and this upper limit can easily be raised: the only pre-requisite is that the underlying C processor must support at least `PP_MAX` arguments in a macro invocation.

The reference implementation of the ellipsis framework follows a modular design, and it is quite trivial to support operations on larger integers simply by updating a few macro constants and extending the o*n*__ family in an inductive manner (defined in the header <meta._>); all in all, the entire process is fairly mechanical.

The following changes need to be made in the header <count._>, located in `.include/ellipsis/` directory:

- Define a value greater than 127 in the replacement text of `PP_MAX`.
- Change the replacement text of `PP_MAW` to one less than the value defined for `PP_MAX`.
- Update the replacement text of `PP_LOG2` with $\lfloor \log_2(\texttt{PP\_MAX}) \rfloor$ (truncating any fractional part).
- Compute the value of $\lfloor \sqrt{\texttt{PP\_MAX}} \rfloor$ (truncating any fractional part) and update `PP_SQRT` with this result.
- Update `PP_INT` as the sequence of integers from `PP_MAW` - 1 through 1 (in reverse or decreasing order).

The following changes is required in the header <utilities._>, also located in `.include/ellipsis/` directory:

- Update `PP_RANGE` as reverse of `PP_INT`, *i.e.* the increasing sequence of integers from 1 through `PP_MAW` - 1.

Finally, for each integer i greater than 127 and up to the updated value of `PP_MAX`, define a macro o*i*__ as:

```
#define oi__(F, f) F(f)oi−1__(F, f)
```

In other words, each o*i*__ macro invokes the one immediately before that. The default o*n*__ family up to `o127__` is defined in the header <meta._>, and additional macros beyond `o127__` should be defined in the same header.

Appendix D

Benchmarking

The noticeable slowness in the compilation of C_ programs is primarily due to the large scale preprocessing overhead incurred by the reference implementation, which is by virtue of fundamental limitations in its design itself. This appendix chapter explores an approach to quantify performance in terms of number of preprocessing operations.

As discussed in the main chapters, the design of the ellipsis framework is influenced by microprogramming architecture used in physical processors: the function-like macros for arithmetic, logical, and relational operations are implemented with the help of primitive macros that perform preprocessing micro-operations on argument lists. The total number of macro invocations required for a given task can be a benchmark parameter, for which a lower bound can be estimated by counting how many times the primitive macros `cat_`, `echo_`, `pop_`, and `top_` are invoked.

A convenient way to do this is with the help of `__COUNTER__` macro, which is predefined as a GNU C extension (also supported by other compilers). The unique property of `__COUNTER__` is that it is incremented every time it is used, with zero being the initial value. We can modify the primitive macros to use `__COUNTER__` in such a way that their replacement text is not altered, so that `__COUNTER__` gets incremented each time a primitive macro is invoked. Here we present one possible way of updating the primitive macros without changing their functional behavior.

```
#define beanie_c_(...)

#define bean_c_(counter) beanie_c_(counter)

#define cat_( l , r) bean_c_(__COUNTER__) l ## r

#define echo_(  ...) bean_c_(__COUNTER__) __VA_ARGS__

#define pop_(t, ...) bean_c_(__COUNTER__) __VA_ARGS__

#define top_(t, ...) bean_c_(__COUNTER__) t
```

A technical subtlety in the incrementation of `__COUNTER__` is that it is considered to be "used" only when it is scanned and expanded. For this reason, we have introduced two additional macros: `bean_c_` is invoked by the primitive macros, but as discussed in chapter 4, macro arguments are not expanded at the call site, so `bean_c_` calls another macro `beanie_c_`, and it is during this invocation that `__COUNTER__` is actually expanded and considered to be "used", thereby causing its value to be incremented by one. On the other hand, `__COUNTER__` would not get updated if the primitive macros directly invoke `beanie_c_`, as it does not expand its arguments at all.

With this small modification in the header <primitives._>, we can get a conservative estimate of macro invocations required for high-level operations done with the preprocessor. For example, if we run the preprocessor as `cc_ -E` on the text `sort_(echo_, RANGE_(125, 1, 2), RANGE_(126, 2, 2))`, followed by `__COUNTER__`, the preprocessed text is the sorted list of positive integers up to 126, followed by 12157714 as the value of `__COUNTER__`, which indicates the number of times the primitive macros were invoked by `sort_` and its helper macros.

It is worth mentioning that this underestimated count is much lower than the total number of macro invocations; if we consider that each invocation of a primitive macro is initiated by some other macro (such as the on__ family), then the total count would be more than twice the value reported by `__COUNTER__`, going well above 24 million.

Such an astonishing scale of macro invocations remains a major bottleneck in the transpilation of `C_` programs to `C`. While it is hoped that future improvements on the reference implementation can lower these numbers, it should be accepted that a "giant leap" in performance can only be possible with the help of a dedicated compiler frontend for the `C_` dialect. Optimistically speaking, this can also open more opportunities for a wider adoption of C_ in real-world projects, beyond its humble beginnings as a small recreational exercise in metaprogramming.

Appendix E

Build

The build script mentioned in the introductory chapter can be used to generate object files by compiling the source files in `compile/` directory, and placing them in the `object/` directory, having an identical subdirectory structure.

In this appendix, we shall describe the contents of the shell script used to automate the build process on Unix-based environments, along with a brief overview of the command-line options used for `gcc` and `clang` compilers.

E.1 Shell script

The following script code is available in the source file `examples/build.sh`

```
#!/bin/bash

set -e
C_=$(dirname "$(realpath "$0")")

#<<'#'
CC_="gcc   -c -xc -std=c2x -O3 -ftrack-macro-expansion=0 -Werror -iprefix
'$C_'/.include -iwithprefix/ellipsis  -iwithprefix/dialect -iwithprefix/library
-iprefix '$C_'/include -iwithprefix/. -iwithprefix/class -iwithprefix/interface
-Wall -Wextra -Wpedantic -Wcast-align -Wcast-qual -Wswitch-enum -Wwrite-strings
-Wduplicated-branches    -Winit-self  -Wshift-overflow=2
-Wduplicated-cond  -Wnull-dereference -Wstrict-overflow=2
-Wno-override-init -Wno-missing-field-initializers
-Wno-parentheses   -Wno-tautological-compare -Wno-type-limits"
#

<<'#'
CC_="clang -c -xc -std=c2x -O3 -fmacro-backtrace-limit=1 -Werror -iprefix
'$C_'/.include -iwithprefix/ellipsis  -iwithprefix/dialect -iwithprefix/library
-iprefix '$C_'/include -iwithprefix/. -iwithprefix/class -iwithprefix/interface
-Wall -Wextra -Wpedantic -Wcast-align -Wcast-qual -Wswitch-enum -Wwrite-strings
-Wassign-enum -Wshift-sign-overflow   -Wunreachable-code-aggressive
-Wno-override-init -Wno-missing-field-initializers -Wno-pointer-arith"
#
```

```
cd  "$C_"/compile
eval $CC_ lib.c_
strip --strip-unneeded lib.o
cd ..
mkdir -p object
mv compile/lib.o object
cd object

mkdir -p class
cd class
eval $CC_ "'$C_'"/compile/class/*.c_
strip --strip-unneeded *.o
for d in $(cd ../../compile/class     &&  ls -d */)
do
    mkdir -p $d
    cd $d
    eval $CC_ "'$C_'/compile/class/$d"*.c_
    strip --strip-unneeded *.o
    cd ..
done

cd ..

mkdir -p interface
cd interface
eval $CC_ "'$C_'"/compile/interface/*.c_
strip --strip-unneeded *.o
for d in $(cd ../../compile/interface &&  ls -d */)
do
    mkdir -p $d
    cd $d
    eval $CC_ "'$C_'/compile/interface/$d"*.c_
    strip --strip-unneeded *.o
    cd ..
done
```

When the script is executed directly (such as `./build.sh`), the first line starting with the "shebang" `#!` tells that the program `/bin/bash` is to be used for running this script (it is also a comment line due to the preceding `#`).

The `set -e` line is to stop the script immediately as soon as a command terminates with a non-zero exit status, skipping rest of the subsequent commands (recall that a non-zero exit code indicates failure, whereas zero indicates success). Exit status is nothing but the return value of `main` or argument to `exit`/`_Exit`/`quick_exit` calls.

`$0` gives the first command-line argument, which is the name by which the script was invoked: passing it to `realpath` gives the full pathname where the file is located (ending with the filename), and `dirname` obtains the directory name only (without the filename). Enclosing the command within parentheses makes it execute within a subshell, and its outcome is stored in the variable `C_`, which will be used as a path prefix in subsequent commands.

The heredoc starting with `<<` is used to emulate a multi-line comment: the `'#'` after that marks the delimiter whose subsequent occurrence terminates the heredoc text. In the first case, the heredoc itself is commented out by a preceding `'#'`, and the `#` in a subsequent line that would have otherwise delimited the heredoc text becomes an ordinary comment line. Both blocks of heredoc texts are for initializing a variable `CC_` with an invocation of `gcc` or `clang`, with its associated command-line flags that includes several diagnostic options. When the `<<` line is commented out, the following text comes into effect, which is used to set compilation options. For example, to use `clang` instead of `gcc`, uncomment the heredoc before `gcc` options can comment out the one before `clang` options.

The file `build.sh` is located directly within `examples/` directory, whose absolute path is stored in the variable `C_`. After initializing `CC_` with the compiler invocation string, we change the current directory to `C_/compile/`. Once inside the `compile/` directory, we execute the compiler command on the source file `lib.c_`, which generates an object file named `lib.o` containing various external variables and function definitions (for the dialect as well as standard library extensions). The `strip` command is used to reduce size of the object file by removing non-essential symbols that are not required for code relocation and linking purposes (such as debugging symbols). We then create a subdirectory named `object/` within `examples/` directory, and move the file `lib.o` to that directory (the `-p` option to `mkdir` avoids an error if the directory is already present). After that we change our current directory to `object/`.

Once inside the `object/` directory, we create a subdirectory named `class/` and change to that directory. The source files directly within `compile/class/` contain class definitions, which are compiled into object files and placed within `object/class/`; the `strip` command is invoked on each object file to reduce file size. The loop iterates over each directory name in `compile/class/`: for this we open a subshell, change the current directory to `compile/class/`, and the output of `ls -d */` gives all directory names without path prefix and with a trailing `'/'`. For each directory in `compile/class/`, we create a corresponding directory with the same name in `object/class/`, which is used for storing stripped object files of each source file containing the methods associated with that class.

Once the loop terminates, we go back to the `object/` directory, and create a subdirectory named `interface/`. As done for each class, similar steps are repeated to create object files for each interface and its associated methods.

NOTE To run the script as `./build.sh`, make it executable using `chmod +x build.sh` (if not done already).

E.2 Compiler flags

The following compilation options are common for both `gcc` and `clang`.

- `-c` means compile only; in other words, generate relocatable object files without linking them to an executable.
- `-xc` means consider the filename extension as `.c` for subsequent input files and invoke the C language compiler.
- `-std=c2x` sets the language dialect to C23 (also called C2x); this option also affects the diagnosis of `-Wpedantic`.
- `-O3` enables several optimizations at level 3, mostly focused at improving runtime efficiency (though often at the expense of increased code due to space-time tradeoff). Another benefit of using `-O3` is that some warnings options are activated only when certain optimizations are enabled that perform a more rigorous static analysis.
- `-Werror` turns diagnostic or warning messages into hard errors that cause a compilation failure.
- `-iprefix` sets a path prefix for subsequent use of `-iwithprefix` option, until the next occurrence of `-iprefix`.
- Each usage of `-iwithprefix` adds the subsequent name to the list of search directories for `#include` directives: directory name is prefixed with the path specified by the preceding `-iprefix`. More precisely, the following directories (relative to `examples/`) are added to the path for locating header files, searched in the given order:

1. `.include/ellipsis/`	2. `.include/dialect/`	3. `.include/library/`
4. `include/`	5. `include/class/`	6. `include/interface/`

- `-Wall` and `-Wextra` enable warnings that can help diagnose potential sources of bugs and undefined behavior; some of the warnings vary between `gcc` and `clang`, and the precise lists can be found in their documentations.
- `-Wpedantic` ensures strict conformance with the language dialect specified by `-std` option, by flagging the use of non-portable extensions and certain kinds of code whose behavior is not well-defined by the C standard. For example, the features that are marked with an asterisk (*) in this documentation are mostly provided by the reference implementation using statement expressions, a GNU C feature that is supported by several C compilers alongside `gcc`, but as of this writing, the syntax is not permitted by the rules of ISO C grammar.
- `-Wcast-align` warns on casting a pointer type to another pointer type whose dereferenced type has stricter alignment (higher power of two). For instance, a `ULLong` is wider than `UByte` on practically all existing environments, so `-Wcast-align` would generate a warning if a pointer to `UByte` is cast as pointer to `ULLong`.
- `-Wcast-qual` warns on removal of qualifiers from the target type of a pointer. For example, non-modifiable types named without a trailing underscore are implemented using `const` qualifier, so warning would be generated on converting a `Char *` to `Char_ *` (such as by assignment or type cast). It is worth mentioning that there are known workarounds to circumvent this artificial limitation; for instance, the reference implementation provides `unqual__`/`unqual_`* in the header `<pointer._>` by removing qualifiers via type punning using `union`.
- `-Wswitch-enum`, as the name suggests, applies when the controlling expression of a `switch` statement has an enumeration type. Warnings are generated if any constant of that enumeration type has been omitted as a case label, or if a case label is not a constant of that enumeration. A `default` case does not affect this option.
- `-Wwrite-strings` changes the type of string literals to array of `Char` instead of `Char_`, and consequently, warnings are generated when string literals are converted as pointer to `Char_`. In a sense, this option changes the rules of C language to a small extent, and can generate warnings even for strictly conforming programs: this is because the C standard specifies the type of string literals as array of `Char_`, even though updating that array causes undefined behavior. Another point of concern is that `-Wwrite-strings` can silently change the behavior of type-sensitive code; for example, consider a `_Generic` expression whose controlling expression is a string literal, and there are two selection expressions: one associated with `Char *` and another with `Char_ *`.

  NOTE The minor inconsistency between the type of string literals and their non-modifiability dates back before the `const` qualifier was standardized by the ANSI committee in C89/C90; "fixing" this rule in the language standard at a late stage can be counterproductive and cause constraint violations for legacy codebases.

The options starting with `-Wno-` are used to disable specific warnings, most of which are considered harmless.

- `-Wno-override-init` disables warnings when designated initializers use multiple expressions to initialize a structure or union member. Disabling the warning is necessary because the reference implementation uses designated initializers to provide the feature of named arguments for method invocations using `call_` family.
- `-Wno-missing-field-initializers` is used to disable warnings about the absence of an explicit initialization of structure members, which by default are initialized as if with the integer constant `0` (null pointer for members with pointer type). This warning has been disabled to permit default arguments for method invocations when named arguments are not used; as mentioned before, the latter is implemented using designated initializers.

E.2.1 gcc options

The following additional options are used for invoking `gcc` in our build script.

- `-ftrack-macro-expansion` is an option for the preprocessor `cpp` that controls location tracking of preprocessing tokens that undergo nested macro invocations, and this information is shown in diagnostic messages when a macro invocation causes an error. As the foundation of the C_ reference implementation is built upon the preprocessor, even a minor typographic mistake in code can trigger an avalanche of preprocessing errors. A detailed stack trace report of macro expansions can be beneficial for finding bugs in the reference implementation itself; however, they are of limited interest to most programmers, and excessive verbosity can overwhelm beginners about the precise cause of an error, which can be as minor as an extra comma in a macro invocation. Setting this option to zero disables it, thereby limiting the depth of preprocessing error messages.
- `-Wduplicated-branches` warns when two blocks of a conditional expression or statement have identical code.
- `-Wduplicated-cond` warns when two mutually exclusive branches, such as `if` and `elif` (short for `else if`), have conditions whose values can be statically determined to be identical. This makes the code guarded by the `elif` unreachable: when the condition is satisfied for `if` branch, then `elif` branch will not execute due to mutual exclusion, and when the condition fails for `if` branch, then it will also fail for `elif` branch as well.

  This programming fallacy be demonstrated with a concrete example, as shown in this contrived code snippet.

  ```
  #include <c._>

  Int_ fun(Int n)
  begin
      if (!!n) return 0;
      elif (n) return 1;
      return 2;
  end
  ```

  The above function cannot execute `return 1` due to the guarding `elif` having the same condition as its preceding `if`. However, declaring the parameter as `volatile Int n` suppresses the warning, as it tells the compiler that the value of `n` can possibly change from zero to non-zero between the branches, due to external sources of mutation (such as another concurrent thread of execution updating this value via a pointer to it).
- `-Winit-self` warns about initializing an uninitialized variable with its indeterminate value, such as `Int n = n;`
- `-Wnull-dereference` warns about execution paths that can lead to dereferencing a possibly null pointer. This is an example of a warning that is enabled by optimizations: in this case, `-Wnull-dereference` comes into effect when `-fdelete-null-pointer-checks` is active, which is enabled by `-O2` and higher optimizations.
- `-Wshift-overflow=2` warns about integer overflow for bitwise left shift operations; in particular, setting it to `2` enables warning about shifting a `1` to sign bit position, when the promoted left operand has a signed type.
- `-Wstrict-overflow=2` warns about signed overflows in cases where `gcc` assumes that overflow will not occur.

The following options starting with `-Wno-` are used to disable certain warnings turned on by `-Wall` or `-Wextra`.

- `-Wno-parentheses` is used to forgo redundant parentheses in some expressions, notably with `iff` and `implies`.
- `-Wno-tautological-compare` is used to disable few false positive warnings for the reference implementation.
- `-Wno-type-limits` disables warnings on range checking conditions that are always true on most environments.

E.2.2 `clang` options

The following additional options are used for invoking `clang` in our build script.

- `-fmacro-backtrace-limit` determines the maximum backtrace depth for the preprocessing call stack that is implicitly updated for macro invocations. Setting it to `1` reduces the verbosity of preprocessing error messages, which is analogous (though not entirely equivalent) to the option `-ftrack-macro-expansion=0` for `gcc`.
- `-Wassign-enum` when an enumeration type lvalue is assigned a value other than its enumeration constants.
- `-Wshift-sign-overflow` warns when a left shift operation with a signed left operand (after type promotion) might shift a `1` to the sign bit position; this option is analogous to the setting `-Wshift-overflow=2` in `gcc`.
- `-Wunreachable-code-aggressive` activates several diagnostic options for detecting code points that cannot be reached during execution. Removing such code reduces object file size without changing functional behavior.
- `-Wno-pointer-arith` is used to disable some false positive warnings for address arithmetic on pointer to VLA.

Appendix F

References

0. *ISO/IEC 9899:2023 — N3096 Draft*, April 1, 2023
   `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf`
1. *ISO/IEC 9899:201x — N1570 Draft*, April 12, 2011
   `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
2. *ISO/IEC 9899:TC3 — N1256 Draft*, September 7, 2007
   `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
3. *Rationale for International Standard — Programming Languages — C*, April 2003
   `https://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf`
4. *The GNU C Reference Manual*
   `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`
5. *Warning Options Using the GNU Compiler Collection (GCC)*
   `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`
6. *Optimize Options Using the GNU Compiler Collection (GCC)*
   `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
7. *Preprocessor Options Using the GNU Compiler Collection (GCC)*
   `https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html`
8. *Clang Compiler User's Manual*
   `https://clang.llvm.org/docs/UsersManual.html`
9. *Diagnostic Flags in Clang*
   `https://clang.llvm.org/docs/DiagnosticsReference.html`
10. *LLVM's Analysis and Transform Passes*
    `https://llvm.org/docs/Passes.html`
11. *Clang Command Line Argument Reference*
    `https://clang.llvm.org/docs/ClangCommandLineReference.html`

12. Steve Summit, *comp.lang.c Frequently Asked Questions*
    `https://c-faq.com`
13. David Keaton, *Programming Language C — C23 Charter*, November 9, 2020
    `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2611.htm`
14. Martin Sebor, *Interpreting the C23 Charter*, May 10, 2022
    `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2986.htm`
15. Dennis M. Ritchie, *The Development of the C Language*, April 1993
    `https://www.nokia.com/bell-labs/about/dennis-m-ritchie/chist.html`

Index

by cHaR

<https://github.com/cHaR-shinigami/c_>