Phelma SEOC 2A APMAN Course handout

François Cayre

September 22, 2025

Abstract

This course introduces some useful programming techniques in C that are applied to build a library of reusable data structures.

Code testing and benchmarking are expected to be two important parts of your work.

This document has been designed to fill the gap between C you wrote when you were a child (last week), and C as it is written by the adults in the room.

Note on LLMs: Use them to understand code, not to write it for you. Use them also to help you when reading the references in the bibliography—in any case, these references and the material in this document set the bar of the expected technical depth of your report. The deepest your technical understanding of what you are doing, the best you will be able to make out of LLMs: toddlers and grown-ups rarely ask the same questions and at best they only obtain answers they may understand.

Contents

List of Figures

List of Codes

Introduction

This is a crash course on Advanced C Programming with an emphasis on:

1. Code reusability: you should end up with software that you can actually reuse for other projects;
2. Testing: your code shall *prove* that it actually does what it should;
3. Benchmarking: your code should get the most out of the machine it runs onto;
4. Reporting: you should be able to write clearly because you do not work alone and at least your future self should respect you.

This is for real

Not only are you going to write code, but also code about your code—for testing, benchmarking and reporting purposes. All of these areas of computer engineering are hard, and even harder to get to work in C.[1]

Hence, we have no other choice but to provide you with a bootstrap that already implements the basic requirements and mechanisms we shall need. This bootstrap is entirely open-source so you can study it in every possible detail in case you need to. It only uses professional, standard tools lovingly tweaked for a seemless and comfortable programming experience. You are expected to spend some time understanding how it works.

This bootstrap consists of several parts:

- A `Makefile` that controls everything;
- An environment for managing debug/release code;
- An interface for testing;
- A very basic benchmark example, complete with integration into:
- A skeleton for your report.

You are expected to find and read additional material by yourself. To get you started, the bibliography contains many references that are just one click away. Those that are not are books that are widely available. List the additional references you used in `report/biblio.bib`.

How to ask a question

Let us assume the following hypotheses:

- Nobody will do your work for free;
- It is perfectly normal to get help from others;
- In less than 2 years, you'll be on your own.

It follows that the only acceptable way to ask for help is detailed on StackOverflow:

[1]Interpreted or more recent languages than C like Python or Rust have builtin features in the language to ease these steps considerably.

1. Explain what you want to achieve;
2. Explain what you tried already.

Example: *"I do not know how to write a `Makefile` or LaTeX code, will you give me a course?"* No way.

Example: *"I have read a few tutorials on writing a `Gnuplot` script, I tried to change feature X in that of the bootstrap to generate that particular plot I want but I observe the following issue..."* Sure I'll help!

So. . . What's the plan?

The plan is to practice, to try, to fail, to try harder and to succeed. You only learn when you struggle. This document and the bootstrap are just here to provide you with guidelines.

Very often at the start of a project in C, one misses data structures that are not built in the language—C only has arrays, other languages also have lists, maps and a few more. It turns out that coding data structures in C not only is pretty useful, but it also considerably improves your understanding of these beautiful mathematical objects and how they behave on the metal. Obviously, this understanding has to be substantiated with tests and benchmarks.

Your goal in this course is to build as much efficient data structures as you can, with your report serving as a reference for your code. Each data structure should be introduced by a description followed by its interface and the main decisions for its (various) implementation(s).

There are many great books online on data structures, see [1] for instance. You will learn the most when you *compare* what (at least!) two different authors say on the same data structure—or how they introduce their distinctive features. In many ways, your report shall resemble these books. Cite your sources of inspiration!

There is a huge number of data structures out there, some of them may be niche or tailored for domain-specific applications (think of spatial data structures, that are used in cartography). We shall target data structures that are expected to find general use. At the very least, you want dynamic arrays, lists, stacks, queues, sets and maps. Each of which may be subject to a particular implementation chosen at compile-time by the user. Of course, you are free to add other useful data structures like heaps, priority queues, graphs or whatever you think is also useful.

Besides the joy of documenting a great experience, writing a good report is important because our focus is on *implementing* data structures in C, in such a way that you may eventually *forget* how you wrote them:

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."—Mark Weiser.

It all starts with interfaces

An interface tells the user of some library [2] (external) code how to use it. For example, the `double cos( double );` prototype is part of the interface for trigonometry in the mathematical library: it tells us that if we want our code to use the code for computing a cosine, then we must pass a `double` as its only parameter.

In many ways, an interface is akin to a contract between the user and the people who have written the library. User code should not change upon the invention of some hypothetical faster way of computing cosines that will be available in the next release of the mathematical library. Hence, the separation that is implemented by an interface allows library code, which is the actual *implementation* of the interface, to evolve on its own.

Regarding data structures, interfaces should be reasonably general and coherent. For example, an interface for lists may or may not have a function to reverse them, or it may have it in a future release. But an interface for stacks that cannot pop an element is simply not acceptable.

Coherence means that the same coding rules are enforced uniformly across the library code. As an example, the coding rules of the C Standard Library have that:

- A function that fails returns either a negative value (in case the function returns an integer) or `NULL` (in case it returns a pointer);
- Functions that print something return the number of characters they actually wrote,
- *etc.*

These are pretty sane rules you are expected to also follow.

As for us, we already state our first additional coding rule:

"Functions that write to some data structure take its address as their first parameter." Many such functions will not need actually change the address of the data structure, but those who have to may do so, and the user does not have to bother about that—with the additional benefit that having to explicitly write an ampersand (`&`) upon calling the function will act as a reminder that the address *may* have changed.

Our second coding rule builds on the previous one to state that:

"Deleted data structure variables are reset to `NULL`." Be aware that it may give a false sense of code safety, however it blends nicely with functions preconditions and still provides for safer C code on average.

Course evaluation

The person in charge of evaluating your work will only do the following things:

1. Clone your repository;
2. Issue a `make` command to compile (code, tests, benchs, report) and run everything (tests and benchmark);
3. Read and evaluate your report, code and tests.

The bootstrap that is provided ensures the above will run smoothly.

Make real sure this is the case with your final repository.

No debug will be attempted on our side.

The deadline will be set in class but in any case it will be before starting your next big project.

Data structures (DS)

Data structures are the basic building blocks to implement abstract data types.

Each data structure you write shall feature:

- Its description (features, theoretical complexity of primitives, *etc.*), interface and implementation notes (genericity, optimizations, *etc.*) in the report;
- Good code coverage by tests (serving as in-depth reference).

Here are the tentative number of points you will be rewarded for each data structure:

- Arrays (1), sorted arrays (1),
- Deques (2),
- Hash-based structures (2/variant),
- Tree-based structures (2/variant).

Abstract data types (ADT)

These are what the user of your library code will use, hence it is the actual code to be called for benchmarking.

An abstract data type may be parameterized to change the default data structure (implementation, variant) that it uses under the hood. Sound defaults must be provided.

Each abstract data type you write shall feature:

- Its description (interface, available implementations);
- Functional tests (serving as documentation).

Ideally, functional tests are code-templated so all possible implementations (and their variants) are easily demonstrated to work as expected.

Your code is expected to expose the following minimal set of abstract data types:

- Arrays (1), sorted arrays (1);
- Lists/stacks/queues (2) from arrays or deques;
- Sets/maps (2) from (un/sorted) arrays or tree-/hash-based structures;

Benchmarks

The only mandatory benchmark in your report is that of the map ADT, that compares the available underlying data structures.

In many ways, it should be the culmination of your work. A benchmark is supposed to be backed by:

- Overall design rationale and implementation;
- Comments (discrepancies with respect to expected theoretical behaviour, tentative explanations, parameter tuning, *etc.*)

Of course, you are free to construct as much additional benchmarks as needed to back your claims.

Benchmarking maps is the hardest (but it's really not unfeasible!): if you end up treating the part on sorting routines, their benchmark will be quite easy to design.

Now for some flash-forward. As you already know, closed-addressing hashtables use an array of lists. Open-addressing hashtables only use arrays. So the DS/ADT part should start with array- and then deque-related stuff. Since the interface for sets/maps is pretty straightforward stuff, the design and implementation of the benchmark is rather an independent effort (see how nice it is to have ADTs?) Full-featured hashtable variants are expected to account for an important part of your effort.

Extra credits

After showing the effect of memory prefetching and limited cache size on the performance of open-addressing maps in your banchmark (now *that* was a hint!), you are rightfully overwhelmed by the void of idleness.

Why not have a look by yourself at the following suggestions to push your library to the next level? But this needs to be carefully discussed beforehand so please contact a teacher first!

The number of stars is the perceived difficulty level if full-featured Robin-Hood hashtables are considered (***):

- Custom memory allocators and arena-based allocation (***);
- Catenable strings, ropes (**);
- Thread pool for asynchronous parallel computations (*****);
- Efficient linear algebra routines (****).

Course material

The next chapter lists some useful common idioms in C: a few of them are used extensively in the following chapters that describes four strategies for code reusability in C and how iterators may be leveraged to write much cleaner code. The last chapter is a presentation of the code bootstrap.

Chapter 1

Common C idioms

We list in this chapter common idioms in C. These idioms are intended to make you focus on real features of your code, not spending your life chasing obvious segfaults.

When we write `function(number)`, it means that if you want to know what we are talking about, you should get help from the manual in your terminal:

```
man number function
```

Nobody will answer questions like "What does `function`‘ do?" – except maybe with "RTFM!"

The `number` is usually `3` or `2` in this course (these are the relevant sections of the manual for the standard library and system interface, respectively).

1.1 Shortcuts of logical operators

One easy way to avoid a few segfaults is to make sure you are not accessing some `struct` field behind a `NULL` pointer. For instance, let:

```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
  char    *name;
  double  height;
  size_t  age;
} *my_struct_t;

int main () {
  my_struct_t data = NULL;

  assert( data && data->name );
  printf( "Name = %s\n", data->name );

  exit( EXIT_SUCCESS );
}
```

Observe how the `&&` operator will shortcut in the `assert(3)`, so a wrongful access to `data->name` in the assertion is not even attempted.

Many variations are used throughout this course, that also include the `||` operator (which does not attempt to evaluate its second operand if the first has already been evaluated to be true). This is especially useful when writing macros.

1.2 More useful assertions

Use `assert(3)` to ensure pre/post-conditions are met upon entry/exit of a function. Errors should be caught as soon as possible, before they propagate. Sometimes the condition to be asserted is not obvious, so you can use a direct application of the above to write more informative assertions like:

```c
int main () {
  my_struct_t data = NULL;

  assert( data && data->name && "Missing data or name" );
  printf( "Name = %s\n", data->name );

  exit( EXIT_SUCCESS );
}
```

The `&&` operator will shortcut before attempting to evaluate the string `"Missing data or name"` (which is evaluated as the non-`NULL` address of a constant string, hence the string is true). But since `assert(3)` will print the whole assertion, including the string because it is part of it, you get more informative messages on failed assertions for free.

1.3 Ternary? operator: explained

A compact `if( [TEST] ) [CONSEQUENCE] else [ALTERNATIVE]` construct is sometimes written as a one-liner with the ternary operator `[TEST] ? [CONSEQUENCE] : [ALTERNATIVE]`.

The above could be rewritten so as to not abort like:

```c
int main () {
  my_struct_t data = NULL;

  printf( "Name = %s\n", (data && data->name) ? data->name : "**Error** Missing data or name"
    );

  exit( EXIT_SUCCESS );
}
```

1.4 Comma

The comma operator in C (`,`) is used to execute statements in sequence. It evaluates to the result of the last expression in the sequence. For example:

```c
  int a = ( printf( "Guess who's back?\n" ) , 2+5 );
```

will store 7 in variable `a` *after* the `printf(3)` has been executed. Quite useful for debugging macros. Or even for writing them.

1.5 Macros... or not?

Macros are introduced with the `#define` preprocessor directive. The are expanded *before* the compiler is actually called. They may be used to define constants or to locally tailor source code for various use cases.

The common wisdom is that macros that define constants should be written with capital letters like:

```c
#define MY_CONSTANT 42
```

Macros for tailoring code may be written in lowercase (think of the `assert(3)` *macro*!), and uppercase should be used in local code (lowercase should be reserved for such macros exported in header files).

The following is an attempt for a reasonable use of macros. Because one of their main danger lies in using them all over the place.

1.5.1 You have been warned

Their other main danger is that they may introduce unwanted side effects. Imagine we have the following macro to compute the absolute value:

```c
#define ABS(x) ( (x) < 0 ? -(x) : (x) )
```

Now let `int v = -2 , w = ABS( ++v );`. The expected value of `w` would be `1` (absolute value of `-1`). But the macro will be expanded to:

```c
int v = -2 , w = ( (++v) < 0 ? -(++v) : (++v) );
```

The test of the ternary operator would increment the value of `v`, but `v` would *also* be incremented in the `-(++v)` part of the ternary operator, leading to `w` having value of `0`.

Never use a macro with side effects.

Also, there is the possibility that the user will feed your macro with fancy expressions as parameters. As a good default rule, surround your macro parameters with parentheses in the expansion expression.

1.5.2 Constant macros and the compiler

Macros may or may not be defined, and they may or may not have an associated value. Imagine our `code.c` uses some block size value `BLK_SZ`. We may provide some safe fallback inside `code.c` in case it is not defined:

```c
#if !defined BLK_SZ
#define BLK_SZ 1024
#endif
```

If the default value is fine, then we may simply compile like:

```
cc code.c -c
```

If, however, we want a custom value, we can pass it to the compiler like:

```
cc -DBLK_SZ=2048 code.c -c
```

1.5.3 Constant macros *vs.* `enum`'s, passing option flags

Macros defining integer constants should be reserved for the use cases above. For constants that are truly local to the code, prefer using an `enum`. Imagine we want to implement read/write/execute permissions on our block. We could use the following:

```c
enum { BLK_READ = 0x1, BLK_WRITE = 0x2, BLK_EXECUTE = 0x4 };
```

Observe that we can set arbitrary integer values to `enum` members. We used hexadecimal because it's then easier to describe option flags that we may want to combine by passing them in one integer parameter:

```c
int block_set_perms( blk_t *blk, int perms ) {
    blk->perms = perms;
}

int block_fprint_perms( FILE *fp, blk_t *blk ) {
    int nchars = 0;
    nchars += fprintf( fp, "%c", blk->perms & BLK_READ    ? 'r' : '-' );
    nchars += fprintf( fp, "%c", blk->perms & BLK_WRITE   ? 'w' : '-' );
    nchars += fprintf( fp, "%c", blk->perms & BLK_EXECUTE ? 'x' : '-' );
    return nchars;
}
```

To set read and write permissions, we would then write:

```
block_set_perms( &blk, BLK_READ | BLK_WRITE );
```

1.5.4 Introducing local scopes

In case you need a local scope in which you could declare variables, the following idiom is a classic:

```
do { /* variables decl. and code */ } while ( 0 );
```

Variants follow, that are more fit to implement visitor macros (because the trailing `while ( 0 )` gets in the way of clearer code).

Now suppose your macro needs some local variable(s). Use a `for` construct that is guaranteed to loop only once. Let's start with an integer:

```
for ( int my_local_integer , once = 1 ; once ; once-- )
```

The same may be done with pointers (see below for another variant with pointers):

```
for ( int *my_local_int_pointer , **once = &my_local_int_pointer ; once ; once = NULL )
```

Since each `for` construct introduces a new local lexical scope, the identifier `once` may be used everytime. Each `for` construct may introduce several local variables of the same base type. Use as many `for` constructs as different types of local variables are needed. This is most useful for `foreach`-like macros that would iterate over some data structure.

A macro may be defined on several lines, that must then end with a \. We may therefore write:

```
#define MY_MACRO( foo ) \
  for ( int my_integer , once = 1 ; once ; once = 0 )
```

Useful constructs can be achieved by leveraging the builtin shortcuts of logical operators:

```
#define MY_CUSTOM_FOR( integer_ptr , min , max , curr ) \
  for ( int *elem_ptr , *once = 1+(int*)NULL ; once ; once = NULL ) \
    for ( elem_ptr = integer_ptr + min ; integer_ptr && min < max && elem_ptr < integer_ptr +
    max && ( curr = *elem_ptr || 1 ) ; elem_ptr++ )
```

The second `for` loop above is the one that we are really interested in. Observe how we ensure its body will *not* be entered if `integer_ptr` is `NULL` or if `min >= max`. The value of `current` is only set after these conditions are ensured. In order to keep in the loop even if it is zero, we use a logical or (`||`) that will always evaluate to true. We may then call it like:

```
int array[ 42 ];
int current;

MY_CUSTOM_FOR( array , 10 , 22 , current ) {
    printf( "Current array value: %d\n", current );
}
```

1.5.5 X-Macros

The socalled X-macros are macros that expect external symbols to be defined before they are loaded. One application is to build quick and uniform serialization routines [3, X-macros]. We shall use extensively this idea when we introduce code templating in Sec. 2.2.

1.6 Common integer manipulations

Much more recipes like the two below are described elsewhere [4] [5].

1.6.1 Checking that an integer is a power of two

Let `int i = /* some value */;`. To check whether `i` is a (strictly positive) power of two, use the following macro:

```c
#define INT_IS_POWER_OF_TWO( value ) ( (value) > 0 && ( (value) & ( (value) - 1 ) ) )
```

1.6.2 Number of blocks for data of given size

Oftentimes, one needs to know how many blocks of size `BLK_SZ` bytes are needed to accomodate `len` bytes of data. A macro using integer arithmetics for that is:

```c
#define NBLOCKS( len, BLK_SZ ) ( ( (len) + (BLK_SZ) - 1 ) / (BLK_SZ) )
```

1.7 Doing math on address values

When you want to convert pointer values to an integer type so you can do math on it, convert your pointer to `uintptr_t`. Then convert back to your pointer type of choice once your computation is done. Now for a totally useless example, let:

```c
#include <stdint.h>
#include <stdlib.h> /* exit(3), EXIT_SUCCESS */

int main () {
  char          *str = "This␣is␣a␣constant␣string";
  uintptr_t ptr_value = (uintptr_t)str;

  ptr_value /= 2;

  str = (char*)ptr_value;

  /* Segfault is looming if trying to print 'str' now... */

  exit( EXIT_SUCCESS );
}
```

These types are available upon a `#include <stdint.h>` and this header also gives you access to useful integers of specified length like `uint8_t` up to `uint64_t`. Signed versions are also available as `int8_t` and so on.

Note that `stdint.h` will include `stddef.h`, which defines `NULL`, `size_t` and a bunch of other useful stuff (so we can avoid including the bigger `stdlib.h` more often).

1.8 From pointer tagging to NaN-boxing

It is not uncommon to have a few bits of meta-information to be added to some pointer value.[1] By default, `malloc(3)` and friends will return an address that is suitably aligned for any C builtin type. The largest one

[1]Imagine we have three bits of meta-information. Among many other examples, we could imagine to mark a pointer as being the address of one of 2^3 possible data types (for more zero bits and types, use `posix_memalign(3)` with a bigger alignment), we could mark a pointer as "in-use" (typically when implementing a garbage collector), Red-Black trees need two bits to encode their "color", *etc.*

being `double` (8 bytes), it follows that every valid address ever used by `malloc()` will have its three lowest bits at zero.

Pointer tagging is a direct application of the above where these zero `malloc()` address bits are set to any more useful value, and obviously zeroed upon accessing the actual address. It may be implemented in a few macros like:

```
#define PTR_VALUE( ptr )      ( (uintptr_t)(ptr) & 0xfffffffffffffff8 )
/* Interface: Read (generic) address or tag value from pointer */
#define PTR_ADDR( ptr )       ( (void*)PTR_VALUE( ptr ) )
#define PTR_TAG( ptr )        ( (uintptr_t)(ptr) & 0x0000000000000007 )
/* Interface: Set pointer tag */
#define PTR_MARK( ptr, tag )  PTR_ADDR( PTR_VALUE( ptr ) | PTR_TAG(tag) )
```

For instance, one would then have to write `free( PTR_ADDR( ptr ) )` so the lowest bits of the pointer address are correctly reset before the call to `free()`.

Of course this introduces a small constant overhead, but the only alternative is to use a structure with explicit metadata—which is heavy and will use at least four additional bytes for C structure fields alignment reasons.

Note that similar hacking of builtin types internal representation includes the infamous NaN-boxing trick [6][7, Optimization]. This time, the socalled "quiet Not-a-Number" condition of a `double` is leveraged to *also* allow the representation of pointers and medium-sized integers in a `double`. This may come in handy if you have some small interpreted language to throw off: basic dynamic typing (that links type and value information in a variable) is only a few macros away.

1.9 A better type for sizes and offsets

This one may be a bit controversial so take it with a pinch of salt. Surprisingly enough, it turns out we're better off with *signed* sizes and offsets. So a good replacement for `size_t` is `ptrdiff_t`.

1.10 Advanced topics

Here, we list several other general techniques for the sake of exhaustivity. We shall not need them however. The first two could make valuable improvements to your own code, the others begin to be somewhat more arcane. The list is ranked by decreasing level of general usefulness.

1.10.1 Arena-based memory allocation

In real-life projects, you may quickly encounter ugly pointer soups: data structures that reference each other, shared pointers between several data structures and many other tricky issues that make proper memory deallocation a nightmare or even impossible due to pointer loops. One possibility is to use a garbage collector but most of the time it is quite overkill. Another possibility is to use a custom, arena-based memory allocator.

An arena is basically a growing set of memory blocks of variable sizes one can allocate from. When your code enters a task that will cause a pointer soup, a new arena is created and memory blocks needed for the task are all allocated from there.

Now deallocation is trivial, no matter the ugliness of the pointer soup: it suffices to deallocate the set of memory blocks in the arena. Arena-based memory allocation make coding in C almost as pleasant as writing code for a garbage-collected language, yet with more control.

The main drawback is that all functions that need to allocate memory have to know about the specific arena to use. Concealing its address in a data structure header does help, but a few functions will still need the arena as an additional parameter.

As the complexity of a project grows, which may happen fairly quickly, one has to implement their own memory allocator(s). That may be for performance reasons (that particular data structure may use a simpler and faster allocator than `malloc(3)`) or, as we have advocated, for writing clear and efficient code for arbitrarily complex memory patterns any application may use. Arena-based memory allocation is discussed everywhere online [8] and a somewhat more portable and less tricky implementation than [9] is extensively covered in [10].

1.10.2 Exceptions

Exceptions are designed to handle irrecoverable errors that are expected to happen only rarely. The typical examples are failed memory allocation or division by zero. See [10] for a comprehensive implementation of simple exceptions in C.

1.10.3 Variadic macros

Variadic macros use the C ellipsis (`...`) to provide a variable number of arguments to a macro. Then the issue becomes that of selecting a different, specialized macro depending on the number of arguments passed to the variadic macro.

The mechanism is quite tricky however (and equally fun), but we shall not need variadic macros in this course. The curious reader will find a simple example in [9] to construct a polymorphic memory allocator (the `new` macro).

1.10.4 Stack-less coroutines

Integer constant labels of `switch` branches may be automatically constructed from the `__LINE__` predefined preprocessor symbol, the value of which is the running line number. This technique builds on the socalled Duff's device and allows to execute different code section depending on where the macro is called from.

One application is to build pieces of code that only remotely resemble coroutines (autonomous pieces of code that interact with each other). These pieces of code alas share the same stack as the executable and they are therefore stack-less [11] themselves because none of them may use that of the executable for its own purposes. Dennis Ritchie himself once flagged Duff's device as one of the brightest abuse of the C preprocessor. Definitely not for the faint-hearted. Having true coroutines in C (with their own stack) is possible but requires hackish integration with the operating system.

Chapter 2

Four tales on genericity

Genericity in the idea that the definition of data structures should be independent from the actual type of items they carry.

Consider the following sample of an example definition of a venerable singly-linked list:

```c
typedef struct link_int_t {
    struct link_int_t *next;
    int               datum;
} *list_int_t;

list_int_t list_int_prepend( int value, list_int_t list ) {
    list_int_t new_head_link = malloc( sizeof *new_link );
    assert( new_head_link && "Allocation␣failed!" );

    new_head_link->datum = value;
    new_head_link->next  = list;

    return new_head_link;
}
```

This definition is *not* generic: the `int` type of the datum must be explicitly specified all over the place, and if we were to *also* maintain lists of `double`'s, we would have to rewrite the entire interface and code to provide this functionality. At the very least, that would bloat source code with redundancy (and duplicate bug occurrences!) More importantly:

The less code you write, the less errors you'll encounter.

There are a few ways to achieve some form of genericity in C, and not all would equally successfully apply to all data structures. It just turns out lists are the easiest data structure to compare them in depth.

2.1 Genericity with `void*` pointers

The *generic* pointer in C has type `void*`: any address may be safely converted to and from this type. This is the simplest strategy for genericity, as we could just store the addresses of datum items as `void*` variables like in:

```c
typedef struct link_t *list_t;
```

`course/demo/generic/pointer/include/list.h`

```c
struct link_t {
  struct link_t *next;
  void          *datum;
};
```

`course/demo/generic/pointer/list.c`

One consequence is that items must be stored elsewhere so they actually have an address. We could store them in an array.[1] But more often than not, one has to resort to a *constructor* like:

```c
void *int_new( int value ) {
  int *int_on_heap = malloc( sizeof *int_on_heap );
  assert( int_on_heap && "Allocation␣failed!" );

  *int_on_heap = value;

  return int_on_heap;
}
```

course/demo/generic/pointer/main.c

So in practice, we would write `list_t list = list_prepend( new_int( 42 ), list_new() );` to create a list with a single integer. But this amounts to *two* calls to `malloc()` to create one such link, which is expected to be pretty slow.

This strategy also suffers from the following drawbacks:

- The blocks for the link and the datum may reside sufficiently far apart in memory, so much so that numerous cache misses would occur (poor memory locality);
- Any access to the actual datum value implies pointer dereferencing, which is slow too (yet another jump to some random address in memory);
- The immense benefits of type-safety checks performed by the compiler are pretty much completely lost.

The main advantage of the `void*` strategy is that it is arguably the easiest to implement.

Our finest point is not to discard any strategy *per se*, but to highlight their internals and relative merits. A good interface will almost always blend a few aspects of each of them. There is no silver bullet.

2.2 Genericity with macros (code templating)

The second strategy would let the *preprocessor* write code for us as soon as we need lists for a specific type of datum. This is often called *code templating*.

Suppose we have the following `CAT` macro that concatenates two syntactic tokens to form a third one:

```c
#ifndef _MACROS_H_
#define _MACROS_H_

#define CONCAT( foo, bar ) foo ## bar
#define CAT( foo, bar ) CONCAT( foo, bar )

#endif
```

course/demo/generic/template/include/macros.h

Now let:

```c
#ifndef T
#error "Undefined␣data␣structure!"
#endif

#ifndef datum_t
#error "Undefined␣datum␣type!"
#endif

#include "macros.h"

#define TYPENAME( T, datum_t ) CAT( T, CAT( _, datum_t ) )
#define TYPE( T, datum_t )      CAT( TYPENAME( T, datum_t ), _t )
#define METHOD( name )          CAT( TYPENAME( T, datum_t ), CAT( _, name ) )
```

course/demo/generic/template/include/generics.h

[1]But this is a static structure! Even when we have dynamic arrays, there is a risk that their addresses change over time. So really, we need some global address for our datum.

These two small files are the core of our templating strategy. They are used to write generic module templates for data structures.

Observe that the symbols `T` (the data structure type name) and `datum_t` are expected to be defined *before* the file is included: this is enforced up to line 7, otherwise compilation is aborted. Similarly, a trailer would automatically `#undef` them so a new cycle of file inclusions may take place.

Our generic module for lists would now contain two template files:

- One for the header (`template/list-export-defs.h`), and
- Another for the actual implementation (`template/list-implementation.h`).

Let's start with the header template `template/list-export-defs.h`:

```c
#ifndef datum_t
#error "Undefined␣datum␣type!"
#endif

#define T list
#include "generics.h"

typedef struct TYPE( link, datum_t ) *TYPE( T, datum_t );

TYPE( T, datum_t ) METHOD( new ) ( void );
int                METHOD( is_empty ) ( TYPE( T, datum_t ) list );
datum_t            METHOD( first ) ( TYPE( T, datum_t ) list );
TYPE( T, datum_t ) METHOD( next ) ( TYPE( T, datum_t ) list );
TYPE( T, datum_t ) METHOD( prepend ) ( datum_t value, TYPE( T, datum_t ) list );
TYPE( T, datum_t ) METHOD( delete ) ( TYPE( T, datum_t ) list, void (*destructor)( datum_t ) ) ;

#undef T
#undef datum_t
#undef TYPENAME
#undef TYPE
#undef METHOD
```

course/demo/generic/template/include/template/list-export-defs.h

Observe the trailer with the `#undef`'s.

Our `template/list-implementation.h` template would then contain how to write all lists implementations for every datum types we may need in our lists like:

```c
#ifndef datum_t
#error "Undefined␣datum␣type!"
#endif

#define T list
#include "generics.h"

#include <stdlib.h> /* malloc(3), NULL */
#include <assert.h>

struct TYPE( link, datum_t ) {
  struct TYPE( link, datum_t ) *next;
  datum_t                       datum;
};

#define NIL NULL
TYPE( T, datum_t ) METHOD( new ) ( void ) { return NIL; }
#undef NIL

int                METHOD( is_empty ) ( TYPE( T, datum_t ) list ) {
  return METHOD( new ) () == list;
}

datum_t            METHOD( first ) ( TYPE( T, datum_t ) list ) {
  assert( !METHOD( is_empty ) ( list ) && "List␣is␣empty!" );

  return list->datum;
}

TYPE( T, datum_t ) METHOD( next ) ( TYPE( T, datum_t ) list ) {
  assert( !METHOD( is_empty ) ( list ) && "List␣is␣empty!" );

  return list->next;
}

TYPE( T, datum_t ) METHOD( prepend ) ( datum_t value, TYPE( T, datum_t ) list ) {
  TYPE( T, datum_t ) new_head = malloc( sizeof *new_head );
  assert( new_head && "Allocation␣failed!" );

  new_head->datum = value;
  new_head->next  = list;

  return new_head;
}

TYPE( T, datum_t ) METHOD( delete ) ( TYPE( T, datum_t ) list, void (*destructor)( datum_t ) )
     {
  while ( !METHOD( is_empty ) ( list ) ) {
    TYPE( T, datum_t ) next = METHOD( next ) ( list );
    if ( destructor ) destructor( METHOD( first ) ( list ) );
    free( list );
    list = next;
  }

  return list;
}

#undef T
#undef datum_t
#undef TYPENAME
#undef TYPE
#undef METHOD
```

`course/demo/generic/template/include/template/list-implementation.h`

And we would end up instantiating the template header to define lists of `int`'s in `type/list-int.h` like:

```c
#ifndef _LIST_INT_H_
#define _LIST_INT_H_

#define datum_t int
#include "template/list-export-defs.h"

#endif
```

`course/demo/generic/template/include/type/list-int.h`

And the actual implementation would end up in `list-int.c` like:

```c
#include "type/list-int.h"

#define datum_t int
#include "template/list-implementation.h"
```

`course/demo/generic/template/list-int.c`

Observe that this strategy encompasses the `void*` strategy upon a simple `typedef void *generic_ptr_t;` so `void*` now has a one-symbol synonym `generic_ptr_t`, that may eventually be used for `datum_t`.

Let us stress again that we implement automatic `#undef`'s so we may reuse templates on demand. Make sure you understand their interplay. Typically, you want your code to be tested against structures of *two* different datum types to validate your whole code. Preprocessor symbols do have a lifecycle too!

In the general case, we should expect better performance: there is only one call to `malloc()` per link, so we also have better locality of memory (the datum is in the same block as the link), and we need not constantly dereference pointers. Also, the compiler may still perform relevant type-checking.

The main drawbacks are not that important:

- Machine code keeps inflating as we have more and more data structures of different datum types (hence it is genericity only at the *syntactic* level);
- Legibility may feel below average (although macro-templating in C is often *much* worse than what we do);

- Debugging may be tedious at the beginning since everything happens in the preprocessor (use `cc list-int.c -Iinclude -E` to skim through the preprocessor output).

The last step, which is not documented here, is to merge the interface and the implementation into a single file. This is rather trivial once an additional preprocessor symbol is used to control macro expansion either of the interface or of the implementation.

Successful uses of code templating include [12]-[13]. The parts of your code that use code templating are likely to resemble these projects (except for the byzantine parts around the various C standards, which you may safely ignore).

2.3 Genericity through intrusive data structures

The third strategy is quite surprising at first glance. We previously embedded the datum into our data structure definition. Now we shall do the opposite and *embed a reference to a data structure into our datum.* Our example list of integers would now look like:

```c
typedef struct {
    link_t   list;
    int      value;
} datum_t;
```

While this may resemble the above, the semantics is vastly different: this should be read as "`int value` is part of a list called `list`". The two other strategies above should have read "this is how we define a list of `int datum`".

An appealing consequence is that we are able to make a datum part of *several* (possibly of different types) data structures:

```c
typedef struct {
  link_t      list;
  link_t      even;
  int         value;
} datum_t;
```

`course/demo/generic/intrusive/two-lists.c`

To achieve this, here is all that is needed for a singly-linked list:

```c
typedef struct link_t {
  struct link_t *next;
} link_t;
```

`course/demo/generic/intrusive/include/list.h`

This strategy is used pervasively in the Linux kernel source code [14], and it relies on custom features of `gcc` that are extensions to the C standard. Instead, we strive to comply with the standard and we need another helper structure:

```c
typedef struct {
  link_t      head;
  ptrdiff_t length;
  ptrdiff_t offset;
} list_t;
```

`course/demo/generic/intrusive/include/list.h`

Observe that `link_t` and `list_t` are no longer pointer types.

The `head` field obviously keeps track of the head of the list, and the `length` field is updated upon datum insertion into / deletion from the list (so getting the length of the list is in $O(1)$ and we need not traverse it).

The `offset` field keeps track of the number of bytes from the start of the datum to a given `link_t` field it contains. It would be `0` for the `list` field of `datum_t` above (which is at the start of the datum), and it would be `sizeof list_t` for the `even` field (because it is after the `list` field).

Let us start by a demo of how this is supposed to be used in practice:

```c
#include <stdlib.h> /*  exit(3), EXIT_SUCCESS */
#include <stdio.h>

#include "list.h"

typedef struct {
  link_t      list;
  link_t      even;
  int        value;
} datum_t;

int      datum_print( datum_t *datum )  { return printf( "%d␣", datum->value ); }
void     datum_delete( datum_t *datum ) { free( datum ); }

datum_t *datum_new( int value ) {
  datum_t *datum = malloc( sizeof *datum );
  assert( datum && "Allocation␣failed!" );

  link_init( &datum->list );
  link_init( &datum->even );
  datum->value = value;

  return datum;
}

int main () {
  list_t  all_ints;
  list_t even_ints;

  list_init(  &all_ints, datum_t, list );
  list_init( &even_ints, datum_t, even );

  for ( int i = 0 ; i < 6 ; i++ ) {
    datum_t *ith_integer = datum_new( i );

    list_push( &all_ints, ith_integer );

    if ( 0 == ith_integer->value % 2 )
      list_push( &even_ints, ith_integer );
  }

  printf( "␣ALL␣ints:␣" );
  list_foreach( &all_ints, integer )  { datum_print( integer ); }
  printf( "\n" );

  printf( "EVEN␣ints:␣" );
  list_foreach( &even_ints, integer ) { datum_print( integer ); }
  printf( "\n" );

  list_destroy( &all_ints, datum_delete );
  list_reset( &even_ints );

  exit( EXIT_SUCCESS );
}
```

course/demo/generic/intrusive/two-lists.c

How this magic is implemented resorts to a fine combination of functions and function-like macros. Let us start with the functions we need in `list.c`:

```c
#include <assert.h>

#include "list.h"

void      list_reset( list_t *list ) {
  assert( list && "List␣head␣is␣NULL!" );

  list->head.next  = NULL;
  list->length     = 0;
  list->offset     = 0;

  return;
}

void      link_init( link_t *link ) {
  assert( link && "Link␣is␣NULL!" );

  link->next = NULL;

  return;
}

void      list_prepend( list_t *list, link_t *link ) {
  assert( list && "List␣is␣NULL!" );
  assert( link && "Link␣is␣NULL!");

  link->next       = list->head.next;
  list->head.next = link;
  list->length++;

  return;
}
```

course/demo/generic/intrusive/list.c

Now `list.h` is only slightly more involved:

```c
#ifndef _LIST_H_
#define _LIST_H_

#include <stddef.h> /* ptrdiff_t */
#include <assert.h>

typedef struct link_t {
  struct link_t *next;
} link_t;

typedef struct {
  link_t      head;
  ptrdiff_t length;
  ptrdiff_t offset;
} list_t;

void      list_reset( list_t *list_head );
void      link_init( link_t *list );
void      list_prepend( list_t *list_head, link_t *link );

#define   list_push( list_head_ptr, object )    \
  list_prepend( list_head_ptr, (link_t*)( (char*)(object) + (list_head_ptr)->offset ) )

#define   list_init( list_head_ptr, type, list_field ) do {          \
    link_init( &(list_head_ptr)->head );                             \
    (list_head_ptr)->offset = offsetof( type, list_field );          \
  } while ( 0 )

#define   list_foreach( list_head_ptr, element )                          \
  for ( void *element, *once = &once ; once ; once = NULL )              \
    for ( link_t *link = (list_head_ptr)->head.next, *next ; link && ( ( next = link->next )
    || 1 ) && ( element = (char*)link - (list_head_ptr)->offset ) ; link = next )

#define   list_destroy( list_head_ptr, destructor ) do {             \
    list_foreach( list_head_ptr, element ) destructor( element );    \
    list_reset( list_head_ptr );                                     \
  } while ( 0 )

#endif
```

course/demo/generic/intrusive/include/list.h

Using the `offsetof` operator is mandatory to get the byte offset from the start of a structure to any of its field (in `list_head_init`). Also observe that we reuse part of the `void*` strategy to produce the generic visitor `list_foreach`. Make sure you understand how `list_foreach` handles the `NULL` end-of-list marker and saving the address of the `next` link upon iterations, in case the current one should be deleted (like in `list_delete`).

The main advantages are as follows:

- An object may easily be part of several data structures (this is close to impossible with code templating);
- Locality of memory is also top notch (datum and link information belong to the same block);
- The compiler may still do most of its type-checking job.

2.4 Genericity through hidden metadata

Metadata is data about the data. In the case of lists, metadata would be the address of the next link. Hiding the metadata means we shall fiddle with pointers so the user is given an address for the datum value, and the metadata shall be stored *before* it, in a separate structure called a *header*:

```
alloc_ptr
   v
   +---------+---------------+
   |  header |  data slots   |
   +---------+---------------+
             ^
         user_ptr
```

We need a `LIST_HEADER` macro to access the start address of the header, as illustrated below:

```c
#include <stdlib.h> /* malloc(3), NULL */
#include <stddef.h> /* ptrdiff_t */
#include <assert.h>

typedef struct list_header_t {
  struct list_header_t *next;
} list_header_t;

#define LIST_HEADER( list ) ((list_header_t*)(list) - 1)

#define NIL NULL
void *list_new( void ) { return NIL; }
#undef  NIL

int   list_is_empty( void *list ) { return list_new() == list; }

void  list_prepend_link( ptrdiff_t value_size, void **list ) {
  list_header_t *header;

  assert( value_size > 0 && "Value␣size␣is␣negative!" );

  header = malloc( sizeof *header + value_size );
  assert( header && "Allocation␣failed!" );

  header->next = LIST_HEADER( *list );

  *list = header+1;

  return;
}

void *list_next( void *list ) {
  list_header_t *next;

  assert( !list_is_empty( list ) && "List␣is␣empty!" );

  next = LIST_HEADER( list )->next;

  return next + ( list_is_empty( next ) ? 0 : 1 );
}

void *list_delete( void *list, void (*destructor)( void* ) ) {
  while ( !list_is_empty( list ) ) {
    void *next = list_next( list );
    if ( destructor ) destructor( list );
    free( LIST_HEADER( list ) );
    list = next;
  }

  return list;
}

#undef LIST_HEADER
```

`course/demo/generic/metadata/list.c`

The interface relies on the `list_prepend_link` function and the comma operator to implement `list_prepend` as a macro:

```c
#ifndef _LIST_H_
#define _LIST_H_

#include <stddef.h>

void   *list_new( void );
int     list_is_empty( void *list );
void   *list_prepend_link( ptrdiff_t value_size, void **list );
#define list_prepend( value, list )                                \
  ( list_prepend_link( sizeof(value), (void**)&(list) ) ,          \
    ( *(list) = (value) ), (list) )
void   *list_next( void *list );
void   *list_delete( void *list, void (*destructor)( void* ) );

#endif
```

course/demo/generic/metadata/include/list.h

The main advantages are as follows:

- Locality of memory is also top notch (datum and link information belong to the same block);
- The compiler may still do most of its type-checking job.

2.5 Summary

Example code for all four strategies is available in the `course/demo/generic` directory. Play with it!

The first two strategies are those that are commonly presented when it comes to genericity in C. The `void*` strategy has many defects that are not compensated by its simplicity. Code templating is mostly OK except it keeps bloating machine code as more different datum types are used in an application. Also, we showed that among the two, only code templating is worth implementing, as it encompasses the `void*` strategy.

In practice, recursive data structures like lists and trees are often better accomodated by the intrusive strategy. The hidden metadata strategy better fits arrays of variable size. These two strategies have good performance and keep the compiler do most of its type-checking work. Another important feature of both of them is that they allow for somewhat more legible code for applications and for data structures.

We also showed how quite a few C idioms and coding techniques actually interact. In particular, you are free to use macros or function-like macros when you design your interfaces. The relevant choice is often guided by the kind of syntactic constructs the user should be able to use.

Reliable and reusable code is preferably short and simple, as we tried to illustrate. It should be tested and validated by actually *using it* and *building more code* with it.

Chapter 3

Four tales on iteration

Very often, we have to execute some code on every element of (a subset of) a data structure. This is called the *visitor* programming pattern. We actually consider the equivalent problem of iterating the addresses of the elements in a data structure. Many algorithms use this capability as a basic building block.

We first review a few technical solutions to implementing iteration over a data structure, and then we decide for a global strategy. We illustrate all along with iteration over an array for simplicity.

3.1 The *visitor* pattern with function pointers

The first strategy relies on function pointers (*callbacks*) that are passed as parameters to some `*_visit` functions, which encapsulate the whole iterations over the data structure.

We would need as many `*_visit` functions as we would have different types of callbacks times the number of traversals to support. Hence, people often decide for a single callback type that is "generic" enough (for instance, returning an `int` and accepting a `void*` parameter).

```
int array_visit( void *array, int (*callback)(void*,void*), void *parm ) {
    int ret = 0;
    for ( ptrdiff_t id = 0 , n = array_length( array ) ; id < n ; id++ )
        ret += callback( (char*)array + id * array_stride( array ), parm );
    return ret;
}

/* Pointless example callbacks: */
int count( void *addr, void *parm )       { return 1; }
int addr_fprint( void *addr, void *parm ) { return fprintf( (FILE*)parm, "%p␣", addr ); }
```

The big issue with this approach is that we get very much constrained by the type of the callback. This often leads to code that is difficult to read because of the layer that has to be added in order to write application code as callbacks of the required type.

3.2 Macros that introduce local scope

The main idea behind the visitor pattern, is that the iteration state information (`id` and `n`) is added on the fly around arbitrary code execution. It turns out that we can do the same by extending the local scope with iteration state information right inside a dedicated macro:

```
#define array_foreach( variable, array )                                       \
    for ( ptrdiff_t id = 0 , n = array_length( array ) ;                       \
          id < n && variable = (void*)((char*)array + id * array_stride( array )) ; id++ )
```

This is much better, as it allows to write quite clean and obvious code right to the point at hand. Also, the iteration state information is allocated on the stack, which frees us from managing it explicitly.

We would need as many `*_foreach` macros as we have different ways of iterating over data structures. Each of them would use nearly as many `for` constructs as there are variables in the iteration state. While it is possible to write such macros in general, this peculiar coding technique may quickly lead to clumsy code that is hard to write and maintain (think for example of keeping *also* the next link of a list in the iteration state, just in case the user deletes the current link, *etc.*)

3.3 Traversals: laziness on the stack with external iteration state

Basically, we should find a cleaner way to the `foreach` macro approach so it gets more general. The two techniques above share the common feature of manipulating the explicit iteration state information at the interface level. But we sort of want "iteration as a callback": we want to generate the next address *on demand*. This is called *lazy iteration*.

So we should abstract the iteration state and assume we have some *external* iteration state somewhere that we can reference as `iter_state` in one particular function that uses it to provide the next iteration.

Let `structure` the address of a data structure, we define a `traversal_t` callback function like:

```
  typedef void *(*traversal_t)( void *structure, void *iter_state );
```

A `traversal_t` must handle its whole lifetime on its own. On the first call to a `traversal_t`, `iter_state` is initialized for iteration on `structure` and the address of the first element in `structure` is returned, or `END_OF_TRAVERSAL` if there is none. On subsequent calls, iteration information is used to find the next element in `structure`, which gets returned (or `END_OF_TRAVERSAL` if we have arrived at the end of the structure).

Note that we use a special value for `END_OF_TRAVERSAL` because in the general case we might iterate over `NULL` as well (think of iterating the values of a hashmap).

Once we have our arbitrary `iter_traversal_t` structure for iteration state information, all we nned is a `iter_traversal_init` constant, so we eventually define a generic `foreach` macro that instantiates the actual traversal:

```
#define END_OF_TRAVERSAL ((void*)-1)

#define foreach( variable, traversal, structure )                              \
for ( iter_ ## traversal ## _t _iter = iter_ ## traversal ## _init , *iter = &_iter ;   \
    END_OF_TRAVERSAL != ( foo = traversal ## _next( structure, iter ) ) || ( variable = NULL )
     ; )
```

This implements our visitor pattern using only one generic macro, without being constrained by function pointers or by clumsy dedicated macros.

Here is an example definition for the obvious `array_forward` traversal:

```
typedef struct {
  char      *array;
  ptrdiff_t  id;
} iter_array_forward_t;

#define iter_array_forward_init { NULL , -1 }

void *array_forward_next( void *array, void *_iter ) {
  iter_array_forward_t *iter = _iter;

  if ( -1 == iter->id ) {
    if ( NULL == array )
        return END_OF_TRAVERSAL;
    iter->array = array;
  }

  iter->id++;

  if ( iter->id == array_length( iter->array ) && ( iter->id = -1 ) )
      return END_OF_TRAVERSAL;

  return iter->array + iter->id * array_stride( iter->array );
}
```

That's all there is to define a new traversal:

- The definition of the `iter_traversal_t` iteration state;
- The definition of its initial state `iter_traversal_init`;
- The definition of the `traversal_next` function.

It is our core tool for iterating any data structure the way we want. And we have our `foreach` macro that works with iteration state information declared on the stack. This allows to write code like:

```c
typedef struct {
    int   bar;
    float baz;
} foo_t;

void foo_print( foo_t *foo ) { printf( "%d,%g␣", foo->bar, foo->baz ); }

foo_t *make_array_of_foos( ptrdiff_t n ) {
    foo_t *arr = array( n, foo_t );

    for ( ptrdiff_t i = 0 ; i < n ; i++> )
        arr[ i ] = (foo_t) { 2*i , i };

    return arr;
}

int main () {
    foo_t *this;
    foo_t *foos = make_array_of_foos( 10 );

    foreach( this, array_forward, foos )
        foo_print( this );

    array_delete( &foos );
}
```

And we would have very few code to change if we wanted our `foos` to be in a list or in any other data structure instead.

The `foreach` macro above is arguably the closest we can get to the `for` construct in Python. The major work is designing the `traversal_t`, but it is often simple and the rest is boiler plate code.

3.4 Iterators

Iterators are traversals on the heap. It means that the very same iteration may now be continued across different sections/functions in the code.

Our interface for iterators boils down to:

- Providing a skeleton constructor `iterator__`, to be used by dedicated iterator constructors;
- Implementing the generic naming convention for calling the dedicated iterator constructor for a given traversal (the `iterator` macro);
- Providing a `yield` function that updates the next iteration address from an iterator, returning `1` if it existed, or `0` otherwise. The iterator is freed and set to `NULL` upon reaching `END_OF_TRAVERSAL`.

Very few code is needed to enable lazy iteration on the heap:

```
typedef struct iterator_t {
  traversal_t   next;
  void          *structure;
  void          *state;
  ptrdiff_t      count;
} *iterator_t;


iterator_t iterator__( void *structure, ptrdiff_t iter_state_size );
#define    iterator( traversal, structure ) traversal ## _iterator( structure )

int        yield__( iterator_t *iter_ptr, void **address );
#define    yield( iter_ptr, variable ) yield__( iter_ptr, (void**)&variable )

#define    iterator_iterations( iterator ) ( iterator ? (iterator)->count : 0 )
#define    iterator_delete( iter_ptr )       free( *iter_ptr ) , *iter_ptr = NULL
```

And the implementation is equally straightforward:

```
iterator_t iterator__( void *structure, ptrdiff_t iter_state_size ) {
    if ( structure ) {
        iterator_t iter = malloc( sizeof( *iter ) + iter_state_size );
        assert( iter && "Alloc␣failed!" );

        iter->structure = structure;
        iter->state     = iter + 1;
        iter->next      = NULL;
        iter->count     = 0;

        return iter;
    }
    return NULL;
}

int yield__( iterator_t *iter_ptr, void **address ) {
    if ( iter_ptr && *iter_ptr && (*iter_ptr)->next ) {
        iterator_t iter = *iter_ptr;
        void       *ret = iter->next( iter->structure, iter->state );

        if ( END_OF_TRAVERSAL == ret )
            iterator_delete( iter_ptr );

        iter->count++;

        if ( address )
            *address = END_OF_TRAVERSAL == ret ? NULL : ret;

        return 1;
    }
    return 0;
}
```

Now to create an `array_forward_iterator` constructor from the `array_forward` traversal:

```
iterator_t array_forward_iterator( void *array ) {
    iter_array_forward_t init = iter_array_init;
    iterator_t iter = iterator( array, sizeof( iter_array_forward_t ) );

    memcpy( iter->state, &init, sizeof init );

    iter->next  = array_forward_next;

    return iter;
}
```

Eventually, emphasizing on spreading the iteration over different functions, our running example with `foo_t` above is now equivalent to:

```c
void print_foos( iterator_t *iter ) {
    for ( foot_t *next ; yield( &iter, next ) ; )
        foo_print( next );
}

int main () {
    foo_t      *foos = make_array_of_foos( 10 );
    iterator_t iter = iterator( array_forward, foos );

    print_foos( &iter );

    array_delete( &foos );
}
```

Iterators are our last and most powerful way of expressing iteration, when it has to be spread across different sections/functions in the code.

3.5 Concluding remarks

We have not talked about *serialization* yet, which is the ability to read/write an arbitrary data structure on disk. One reason, which is now obvious, is that we had to present this material so you have an idea of how iterating a data structure may be done. The other reason is that we do not plan to build a language: we only ought to output a few numerical values in CSV format out of a benchmark on synthetic (randomly generated) data.

Traversals and iterators may be used to support *generators* along infinite data structures or combinatorial enumerations (*e.g.* all consecutive even integers, all permutations of an array, *etc.*) In which case one may need to add private auxiliary data to the iterator, quite likely right after the iteration state information. So the modifications, again, would be minor.

Implementations of combinatorial enumerations are both fast and quite legible using this technique. For instance, many algorithms on graphs are expressed with statements like "for all permutations of all sets of k neighbors of the current node..."—you want generators for cases like these. Not to mention that is it easy to use generators and visitors as bulding blocks for others. Going further, one may want to make `iterator` a variadic macro so actual iterators and generators may be parameterized (like visiting only a subset of an array).

Chapter 4

Code bootstrap

This chapter is a quick visit of the code you have just downloaded. Specific information regarding data structures are in the `report/*.tex` skeleton.

You are expected to take some time to make sure / understand how your working environment works, including the bootstrap. Numerous examples are provided that you can easily duplicate, tweak and learn from.

First thing first: the `Makefile` expects that you work from the project root directory it is in. Always stay in this root directory.

The `Makefile` uses a number of options for the C compiler. Some of them turn the most dangerous warnings into plain compilation errors. There is empirical evidence that doing so starts to improve your coding skills in the first few hours of practice. Plus, this will also save you a considerable amount of nasty segfaults.

4.1 Testing: `make tests`

Testing code is fundamental and extremely handy for several reasons:

- Regularly running tests allows early detection of new bugs (preventing *code regression*);
- The source code of the tests may serve as an *automatically up-to-date reference for your code*;
- Testing may (should!) be code-templated so *different implementations are validated against the very same testing procedure.*

Testing code is difficult for several reasons:

1. In general, it is not possible to test all code (to have complete *code coverage*);
2. Not only do you want to test correct code (*positive tests*), but also incorrect code (*negative tests*);
3. Some tests have to ensure that the code under test actually aborts in certain circumstances, and continue testing nevertheless (you may lack knowledge in system programming to do that);
4. Writing good tests requires great skills in paranoia.

As for the first point, we are quite lucky: data structures are mathematical objects that do not have many parameters (except maybe for performance tuning but this is related to benchmarking). It is therefore expected that your tests will achieve (near) complete code coverage.

The second and third points are the main reason we provide you with a testing environment.

The last point may be mitigated by having one person writing the tests and another in charge of the implementation. The two only have to agree on the interface beforehand. Tests are written first and then passed to the implementation developer. An implementation is deemed working when all tests pass.

Tests should be located in `libellul/test/props` (we are testing the properties of our data structures). Unused tests should be moved to another directory, as `make tests` will compile everything in `libellul/test/props`.

Tests are compiled with a debug version of the library code, which is instrumented by `AddressSanitizer` as a more integrated replacement for `valgrind` (memory leaks are not checked if running from a debugger). Both are uncompatible so you have to set `USE_ASAN=no` in the `Makefile` if you do want to switch to `valgrind`.

Tests compilation *and execution* are triggered with `make checks`. The output is the list of properties that your code actually verifies.

Most features of the testing environment are documented in `libellul/test/howto` with numerous expected failures upon typing `make tests-howto` (due to illustrating tests that do not pass). This testing howto is not expected to run inside a debugger. Sketchy code templating for testing is also outlined.

Regarding code templating specifically, you may want/need to check the preprocessor output only. Tests in `libellul/test/prepr` are not supposed to use the testing environment we provide, they are only used as a stratchpad for you to play with the preprocessor and examine its output. These tests are triggered with `make prepr-tests`, the resulting files are in the same directory and have the `.prepr` extension.

4.2 Benchmarking: `make benchmarks`

A benchmark is a C program in `libellul/bench/` that will generate data (often timings) in text format, from which plots are made: each `benchmark/foo.c` benchmark must also have one associated `benchmark/foo.plt` Gnuplot script.[1]

Compilation and execution of the benchmarks is triggered with `make benchmarks`.

4.2.0.1 Mechanism for benchmarks in the `Makefile`

The benchmark output should be in CSV format: the `Makefile` will automatically save it as `bench/foo.csv`, which is then used by `bench/foo.plt`.

CSV (Comma-Separated Values) is a text format where data columns are separated by an arbitrary character, which is expected to be a comma by default. Check `bench/00-dummy.c` to see how `printf()`'s are used to output values of interest separated by commas.

4.2.0.2 Mechanism for plots in the `Makefile`

Each `bench/foo.plt` may produce as much plots as needed from one `bench/foo.csv` file (that may have as many lines and columns as needed).

For instance, `bench/00-dummy.plt` will produce `bench/first.png` and `bench/second.png`.

4.3 Reporting: `make report`

This section shows how simulations and code may be handled for inclusion in your report. We assume you want to live the LaTeX way, although it is perfectly acceptable to write your report as Markdown files in your repository (the reason we generate PNG plots instead of PDF).

4.3.1 Including code excerpts

You may include any several parts of your own fancy code (compared to [15]) and reference it like is done for Code 1. Automatically referencing lines of code is tedious, so do it by hand if you need it.

4.3.2 Including a simulation plot

The overall operation is rather straightforward and should suffice for most uses. Of course, you are free to tweak the whole thing if you feel like it. Here is how to include a simulation plot that was produced with `make benchmarks`:

[1]You are free to write your own Python scripts to use `matplotlib` instead.

```
#ifndef _ARRAY_H_
#define _ARRAY_H_

#endif
```

libellul/include/libellul/type/array.h

Code 1: Excerpt showing how feature X is implemented (X=C guard).

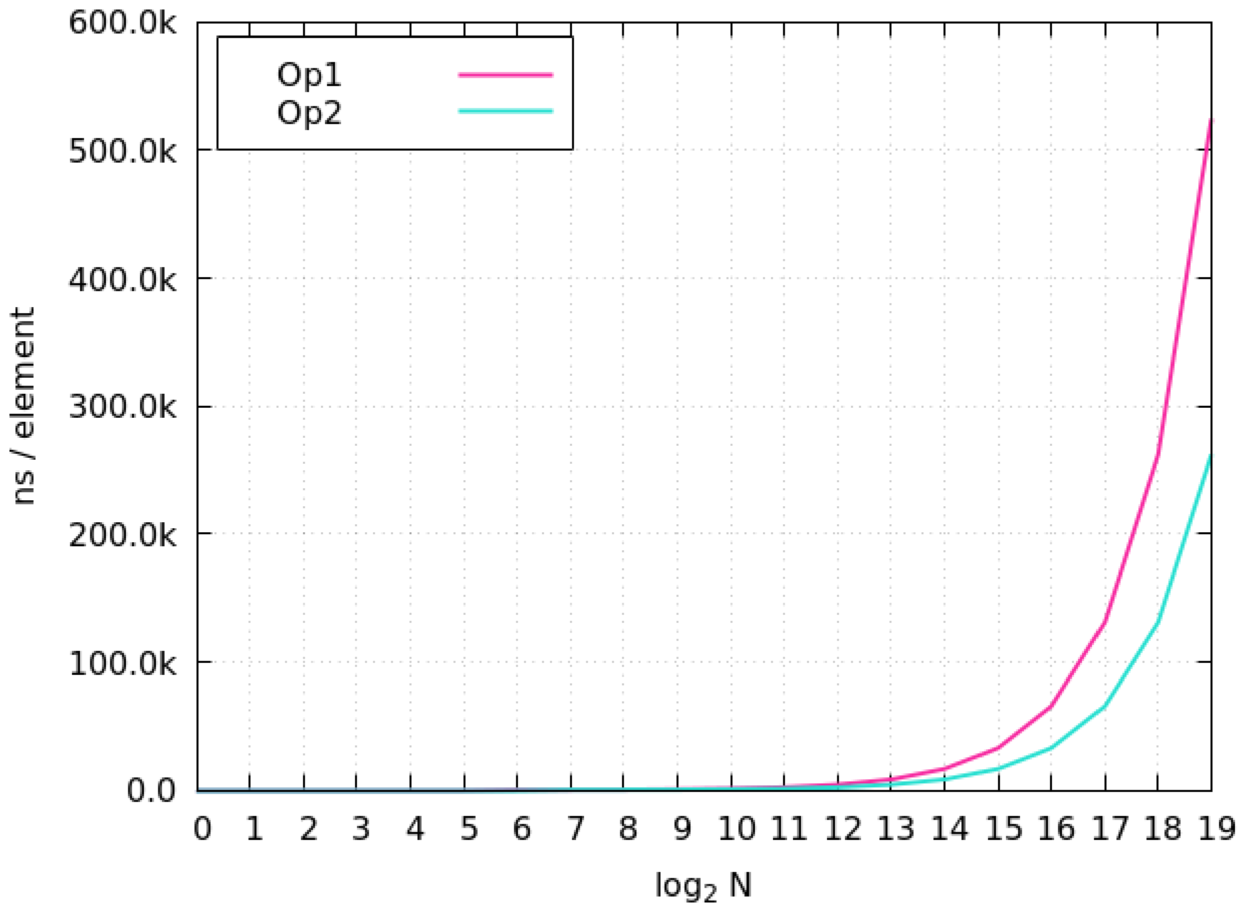

Figure 4.1: Description and salient features for the reader to observe.

Then comes the commentary of Fig. 4.1.

Observe that only the figure basename is needed (*e.g.* `first`) in the LaTeX code above. This is because common extensions (including `.png`) are automatically tested and we added `bench/` to the list of directories containing plots:

```
\graphicspath{{bench/}{report/schema/}} % Where the Makefile puts the plots (+schemas)
```

report/preamble.tex

4.3.3 Schemas

Unless you plan to learn TikZ, use any external tool to generate PNG or PDF for your schemas and place them into `report/schema` to import them like we did for plots.

Alternatively, you may want to revive the fine tradition of ASCII-art like in Fig. 4.2. It is quite handy to make schemas right in the C code and have them reused here.

```
alloc_ptr
   v
   +---------+---------------+
   |  header |  data slots   |
   +---------+---------------+
             ^
        user_ptr
```

Figure 4.2: ASCII-art is not a crime.

4.3.4 What could go wrong?

Only to show a glimpse of the kind of problems you may run into, we plot in Fig. 4.3 the time (in nanoseconds per element) to allocate, generate, and compute the sum of the first N integers. *Incipit tragœdia.*

4.3.4.1 High expectations

Obviously, the plots in Fig. 4.3 are expected to be unconditionnally flat because we normalize linear-time operations (see the loops of `on{stack,heap}` in `bench/00-dummy.c`).

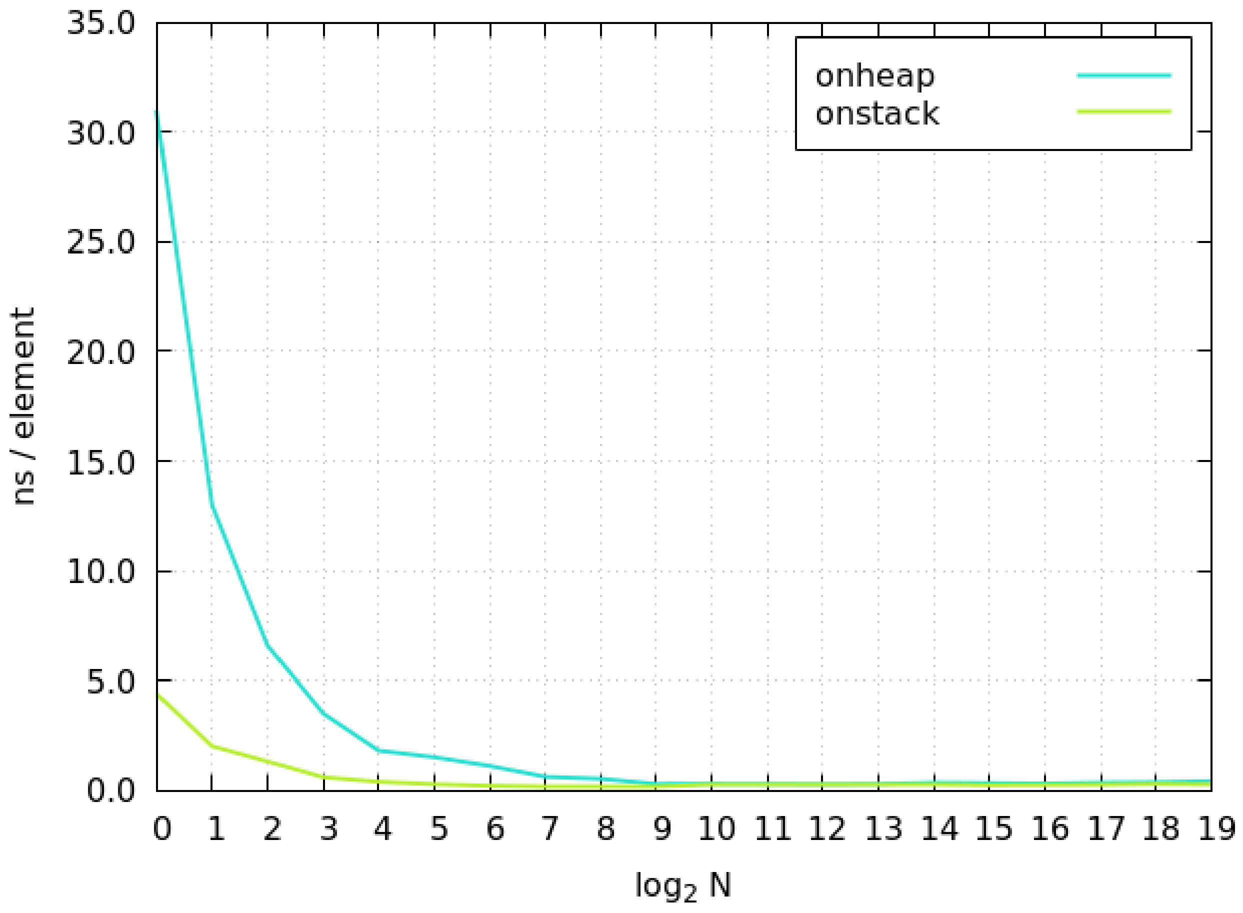

Figure 4.3: Time to allocate, generate, and compute the sum of the first N `int`'s on the heap and on the stack. The leftmost outlier values are due to the benchmark itself.

Instead, we only observe *convergence* with different rates depending whether our values are on the stack or on the heap. The good news however, is that theoreticians are right when they look at $N \to \infty$.

4.3.4.2 Where to go from there

Here are only but a few hints:

1. The system will load code from a library only when it is needed-[16] (because it takes time!);
2. The hardware has to make room for code and data in the cache during execution [17];
3. The `calloc()` allocator has different codepaths for different values of memory needs [18];
4. The hardware is optimized for successively accessing contiguous data in memory [19];
5. The compiler may generate optimized code with automatic loop unrolling [20];
6. Normalization by N will amortize costly constant-time operations (like `calloc()`) less efficiently for small values of N.

As for the first point, `elapsed_nsec()` has to be loaded from `libellul.so`, and `calloc()` and `fprintf()` have to be loaded from `libc.so`. This is related to the second point and it is known as *cache warmup*. One way of warming up the cache is to force dummy execution of a sufficiently large benchmark. To show the effect of cache warmup, try to regenerate the plots after commenting these lines:

```
/* Warmup: Force dummy calls to all useful functions beforehand */
fprintf( stderr, "%g\r", elapsed_nsec() );
onstack( LOG2_N_MAX );
```

`bench/00-dummy.c`

Code 2: Cache warmup is an important issue when writing a benchmark.

The third point basically shows why serious people write their own memory allocator when they know how to optimize for their specific needs.

The fourth point tells us whether our particular problem will behave nicely on the hardware or not. In our example, we are constantly accessing `vec[i+1]` after `vec[i]` so we cannot have a pattern of accessing memory that is more fit for the hardware. Dynamic, recursive data structures and hashtables are not expected to have such nice patterns when accessing memory.

The fifth point tells you that the compiler is able to rewrite short loops so their body are duplicated for different values of `i`. In turn, this allows to take advantage of the superscalar architecture of modern CPU pipelines. This also explains why we have such low values for $N \to \infty$: the actual *throughput* is of several instructions per CPU cycle. Again, our basic benchmark code could not be more fit for the hardware.

The last point advocates for a more clever general design of the benchmark, where smaller values of $\log_2 N$ would incur more runs. At least we can speed up our benchmark this way.

Other parameters may influence a benchmark, like CPU frequency scaling, alignment on cache line size, or system load.

That is the kind of observations you are expected to make and take into account in designing a benchmark that is as reproducible and meaningful as possible. You have to know your hardware and the system that runs it. This is instrumental for understanding other effects like cache line size or branch prediction.

Bibliography

[1] Pat Morin. *Open Data Structures In Pseudocode.* URL: https://opendatastructures.org/ods-python/ods-python-html.html (visited on 08/10/2025).

[2] David A. Wheeler. *Program Library HOWTO.* Ed. by The Linux Documentation Project. 2003. URL: https://tldp.org/HOWTO/Program-Library-HOWTO/index.html (visited on 08/10/2025).

[3] Wikibooks Contributors. *C Programming.* Ed. by Wikibooks. URL: https://en.wikibooks.org/wiki/C_Programming (visited on 08/10/2025).

[4] Sean E. Anderson. *Bit-Twiddling Hacks.* Ed. by Stanford University. 1997. URL: https://graphics.stanford.edu/~seander/bithacks.html (visited on 08/10/2025).

[5] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Publishing, 2002.

[6] Piotr Duperas. *NaN Boxing or How to Make the World Dynamic.* 2020. URL: https://piotrduperas.com/posts/nan-boxing (visited on 08/10/2025).

[7] Robert Nystrom. *Crafting Iinterpreters.* 2015. URL: https://craftinginterpreters.com/ (visited on 08/10/2025).

[8] Ryan Fleury. *Untangling Lifetimes: The Arena Allocator.* 2022. URL: https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator (visited on 08/10/2025).

[9] Chris Wellons. *Arena Allocator Tips and Tricks.* 2023. URL: https://nullprogram.com/blog/2023/09/27/ (visited on 08/10/2025).

[10] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software.* Addison-Wesley Publishing, 1996.

[11] Simon Tatham. *Coroutines in C.* 2000. URL: http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html (visited on 08/10/2025).

[12] Allan Jackson. *Convenient Containers.* Ed. by GitHub. 2023. URL: https://github.com/JacksonAllan/CC (visited on 08/10/2025).

[13] Allan Jackson. *Verstable.* Ed. by GitHub. 2023. URL: https://github.com/JacksonAllan/Verstable (visited on 08/10/2025).

[14] Nicolas Frattaroli. *Linked Lists in Linux.* URL: https://docs.kernel.org/next/core-api/list.html (visited on 08/10/2025).

[15] Dennis M. Ritchie and Brian W. Kernighan. *The C Programming Language.* Bell Laboratories, 1988.

[16] Basile Starynkevitch. *Answer to Question 429188.* Ed. by Stack Exchange. 2018. URL: https://unix.stackexchange.com/a/429209 (visited on 08/10/2025).

[17] Gabriel G. Cunha. *Exploring How Cache Memory Really Works.* 2024. URL: https://pikuma.com/blog/understanding-computer-cache (visited on 08/10/2025).

[18] GNU GlibC Contributors. *Malloc Internals.* 2022. URL: https://sourceware.org/glibc/wiki/MallocInternals (visited on 08/10/2025).

[19] Ulrich Drepper. *What Every Programmer Should Know About Memory.* Tech. rep. RedHat, 2007.

[20] Michael E. Lee. *Optimization of Computer Programs in C.* 1997. URL: https://icps.u-strasbg.fr/~bastoul/local_copies/lee.html (visited on 08/10/2025).