

EXPERIMENTS ON BUILDING HIGH LEVEL ABSTRACTIONS WITH C

PART 1: SMART POINTERS

WITH A TASTEFUL HINT OF GNU EXTENSIONS

Franklin "Snaipe" Mathieu – 2017

4/12/2015

EPITA – GCONFS

SMART POINTERS FOR GNU C99

Memory management is an unsolved problem.

Known techniques rely on RAI, Reference counting, Garbage collection

There is no silver bullet, each has its drawbacks

C++ solves the problem with RAI and smart pointers:

C++ solves the problem with RAI and smart pointers:

File I/O and RAI:

```
{  
    std::fstream file("foo.txt");  
    file << "Hello world";  
} // file is closed when exiting the scope
```

A CASE STUDY ON C++

C++ solves the problem with RAII and smart pointers:

File I/O and RAII:

```
{  
    std::fstream file("foo.txt");  
    file << "Hello world";  
} // file is closed when exiting the scope
```

Memory allocation:

```
{  
    auto ptr = std::make_unique<int>(42);  
    std::cout << *ptr;  
} // the memory is freed when exiting the scope
```

There are no RAII facilities in standard C – cleanup is mainly done with goto

```
FILE *f = fopen("foo.txt", "w");
if (!f)
    goto cleanup;

// do things

if (error_case)
    goto cleanup;

// do more things

cleanup:
    if (f) fclose(f);
```

EXTENSIONS TO THE RESCUE

Thankfully, some compilers leverage these limitations with extensions:

EXTENSIONS TO THE RESCUE

Thankfully, some compilers leverage these limitations with extensions:

MSVC: Structured exception handling and try/finally

```
FILE *f = NULL;
__try {
    f = fopen("foo.txt", "w");
} __finally {
    if (f) fclose(f);
}
```

EXTENSIONS TO THE RESCUE

Thankfully, some compilers leverage these limitations with extensions:

MSVC: Structured exception handling and try/finally

```
FILE *f = NULL;
__try {
    f = fopen("foo.txt", "w");
} __finally {
    if (f) fclose(f);
}
```

GCC, Clang/LLVM, ICC: cleanup attribute

```
void fclose_stack(FILE **f) { if (*f) fclose(*f); }

__attribute__((cleanup(fclose_stack)))
FILE *f = fopen("foo.txt", "w");
```

The cleanup attribute provides the basic building block for an automatic pointer:

```
void free_stack(void **f) {  
    free(*f);  
}
```

```
#define auto_ptr __attribute__((cleanup(free_stack)))
```

```
auto_ptr int *val = malloc(sizeof (int));  
*val = 31>174;
```

BUT IT'S NOT GOOD ENOUGH.

```
auto_ptr<int> *val = malloc(sizeof(int));  
*val = 24;
```

VS.

```
auto_ptr ptr = std::make_unique<int>(24);
```

- Not convenient to use
- No support for structure cleanup
- Resource not shareable

In other words, it's useless garbage.

IMPLEMENTING DESTRUCTORS

We need to store a destructor function.

We need to store metadata.

We need a custom allocator.

Introducing smalloc/sfree:

```
void *smalloc(size_t size, void (*dtor)(void*));  
void sfree(void *);  
#define smart __attribute__((cleanup(sfree_stack)))
```

```
|-----|-----|-----//-----|  
| metadata | padding | actual data          // |  
|-----|-----|-----//-----|  
^                               ^  
returned address (word aligned)  
  
|_ start of allocated  
   block
```

SIMPLE IMPLEMENTATION

```
typedef void (f_dtor)(void*);
typedef struct {
    f_dtor *dtor;
} s_meta;

void *smalloc(size_t size, f_dtor *dtor) {
    size_t prefix_sz = align(sizeof (s_meta));
    s_meta *ptr = malloc(size + prefix_sz);
    ptr->dtor = dtor;
    return (char *) ptr + prefix_sz;
}

void sfree(void *ptr) {
    s_meta *meta = get_meta(ptr);
    if (meta->dtor) meta->dtor(ptr);
    free(ptr);
}
```

```
void *smalloc(size_t size, void (*dtor)(void*));
```



```
void *smalloc(size_t size,  
              void (*dtor)(void*, void*),  
              void *meta,  
              size_t metasize);
```

```
void *smalloc(size_t size,  
              void (*dtor)(void*, void*),  
              void *meta,  
              size_t metasize);
```

:(

```
struct smalloc_args {  
    size_t size;  
    void (*dtor)(void*, void*);  
    struct {  
        void *data;  
        size_t metasize;  
    } meta;  
};  
  
void *smalloc(struct smalloc_args *);  
  
#define smalloc(...) \br/>    smalloc(&(struct smalloc_args) { \br/>        __VA_ARGS__ \br/>    })
```

```
smart int *ptr = smalloc(sizeof (int));
```

```
smart int *ptr = smalloc(sizeof (int), my_int_dtor);
```

```
smart int *ptr = smalloc(  
    sizeof (int),  
    my_int_dtor,  
    { my_meta, sizeof (my_meta) }  
);
```

```
smart int *ptr = smalloc(  
    .size = sizeof (int),  
    .dtor = my_int_dtor,  
    .meta = { my_meta, sizeof (my_meta) }  
);
```

AWESOME!

A LOOK BACK ON WHAT WE DID

`sfree` is the universal deallocator, just like `delete` in C++

ISO C99:

```
whatever *ptr = smalloc(  
    .size = sizeof (whatever),  
    .dtor = dtor_whatever  
);  
// manipulate ptr  
sfree(ptr);
```

C99 with GNU extensions:

```
smart whatever *ptr = smalloc(  
    .size = sizeof (whatever),  
    .dtor = dtor_whatever  
);  
// manipulate ptr
```

We could leave it at that, but...

BUT IT'S STILL NOT ENOUGH.

Unique pointers (no shared ownership)

```
smart int *ptr = smalloc(UNIQUE, sizeof (int));
```

Shared pointers (shared ownership)

```
void *sref(void *); // increments reference counter
```

```
smart int *ptr = smalloc(SHARED, sizeof (int));  
smart int *ref = sref(ptr);
```



```
smart int *ptr = smalloc(SHARED, sizeof (int));  
if (!ptr)  
    abort();  
  
*ptr = 42;  
  
list_add(some_list, sref(ptr));  
list_add(some_list, sref(ptr));  
list_add(some_list, sref(ptr));
```

The boilerplate code strikes back.

```
smart foo *ptr = smalloc(UNIQUE,  
    sizeof (foo),  
    foo_dtor);  
if (!ptr)  
    abort();  
*ptr = (foo) {...};
```

The boilerplate code strikes back.

```
smart foo *ptr = smalloc(UNIQUE,  
    sizeof (foo),  
    foo_dtor);  
if (!ptr)  
    abort();  
*ptr = (foo) {...};
```

- Obligatory NULL check.
- We almost always define a value right away.

Nobody wants to do that hundred of times.

Let's introduce two macros: `unique_ptr` and `shared_ptr`.

```
smart int *yes = unique_ptr(int, 1);  
smart int *yes = unique_ptr(int, .value = 1);  
  
typedef struct { float x; float y; } point;  
  
smart point *zero = unique_ptr(point);  
smart point *p = shared_ptr(point, { 1.3, 4.2 });
```

`smalloc/sfree` is the equivalent of `new/delete` in C++.

There is no equivalent for `new[]` and `delete[]`.

We need to change the implementation of `smalloc` to handle arrays and change `unique_ptr` and `shared_ptr`.

smalloc/sfree is the equivalent of new/delete in C++.

There is no equivalent for new[] and delete[].

We need to change the implementation of smalloc to handle arrays and change `unique_ptr` and `shared_ptr`.

```
smart int *arr = unique_ptr(int[5], {1, 2, 3, 4, 5});
```

smalloc/sfree is the equivalent of new/delete in C++.

There is no equivalent for new[] and delete[].

We need to change the implementation of smalloc to handle arrays and change `unique_ptr` and `shared_ptr`.

```
smart int *arr = unique_ptr<int>(int[5], {1, 2, 3, 4, 5});  
  
for (size_t i = 0; i < array_length(arr); ++i)  
    printf("%d\n", arr[i]);
```

WHAT'S NEXT?

WHAT'S NEXT?

- weak pointers
- move semantics for ownership transfer

Any sensible contribution is welcome.

 <https://snai.pe/git/libcsptr>

 franklinmathieu+libcsptr@gmail.com

Snaipe on [#criterion-dev](#) @ [irc.freenode.net](#)

QUESTIONS?

ADDENDUM

EXTRACTING THE SIZE OF AN ARRAY

Let `T` be a type and `dummy` be a variable of type `T[1]`.

Then:

- `sizeof (dummy[0])` is the size in bytes of the compound type of `T`.
- `sizeof (dummy) / sizeof (dummy[0])` is the length of the array if `T` is an array type, 1 otherwise.

Example:

```
T[1] v;  
size_t size = sizeof (v[0]);  
size_t length = sizeof (v) / sizeof (v[0]);  
printf("%lu, %lu\n", size, length);
```

If `T` is an `int`, this prints 4, 1.

If `T` is an `int[20]`, this prints 4, 20.