Agile programming & C

C can be agile too

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

Introduction

Coding in *C* is usually seen as cumbersome, slow, painful

- No automatic memory management
- No support for oriented object programming
- No exception mechanism
- Very easy to get messy
- Very limited standard library

Introduction

But you might sometime have little other options

- Resource-limited environment
- The only decent compilers for the environment
- Cross-platform & high-performance
- Core for a cross-language library

C & agility

You can use C yet be agile

- Reusable, easily extensible modules
- Scalable error handling
- Code ready for changes

C & agility

The techniques introduced here are used in a lot of libraries

- Toolkits like *GTK–Gnome*
- Interfaces with scripting languages like *Python*, *Lua*
- Picture loading libraries *libpng*, *libjpeg*
- Many parsing libraries for *XML*, *JSON*, ...
- ...

Table of Contents

A basic module

Let's write a pseudo-random number generator in C module, as an example

42 $\Rightarrow$	5	18	45	11	37	12	78	...

A basic module

The module will have the following functions

- *seed* initialize the generator with a *seed* value
- *next* returns the next number in the sequence
- *nextUniform* returns a number in $[0, 1]$ with a uniform probability

A robust header

The header of the module, *Randomizer.h*

```
#ifndef KUAI_GUI_RANDOMIZER_H
#define KUAI_GUI_RANDOMIZER_H

#ifdef __cplusplus
extern "C" {
#endif


...


#ifdef __cplusplus
}
#endif

#endif /* KUAI_GUI_RANDOMIZER_H */
```

A robust header

The header and footer of the module allows simple usage

- No multiples declarations
- Just include that file, and that's it
- In case of $C/C++$ mix, no linking issues
- A fairly unique identifier, to avoid name clashes

Declaration

A structure type contains all the variables for one module

```c
#include <stdint.h>

/*
  Complementary-multiply-with-carry random number generator, see
  George Marsaglia et al. " "Random number generators".
  Journal of Modern Applied Statistical Methods.
    - state size of 32x4 bytes
    - period of about 2^1053
    - 0 is not a valid seed
*/

typedef struct {
  uint32_t array[32];
  uint32_t carry;
  uint32_t index;
} Randomizer;
```

And document your code !!!

Declaration

Using a structure rather than global variables

- Different part can use a different instance of the module
- Improved re-usability
- Ready for multi-thread

Declaration

Declaring the functions of the module

```
/* Seed the random number generator. 0 should be avoided */
extern void
Randomizer_seed(Randomizer* self, uint32_t seed);


/* Return the next integer of the sequence */
extern uint32_t
Randomizer_next(Randomizer* self);


/* Return a floating point value in [0, 1] range */
extern double
Randomizer_nextUniform(Randomizer* self);
```

Declaration

A consistent scheme is *essential* here

- Functions names for the module start by the name of the module
- All functions take a pointer on a module as first parameter

Implementation

Implementation goes in *Randomizer.c*

```
void
Randomizer_seed(Randomizer* self, uint32_t seed) { ... }

uint32_t
Randomizer_next(Randomizer* self) { ... }

double
Randomizer_nextUniform(Randomizer* self) {
  double y;

  y = Randomizer_next(self);
  return y / 4294967295.0;
}
```

Usage

Using our module looks like this

```
#include "Randomizer.h"

Randomizer rnd;

Randomizer_seed(&rnd, 1337);
for(i = 0; i < 42; ++i)
  printf("%u\n", Randomizer_next(&rnd));
```

Objects

Our module looks really like an object, in the *object-oriented* sense

```c
typedef struct {
  uint32_t array[32];
  uint32_t carry;
  uint32_t index;
} Randomizer;

extern void
Randomizer_seed(Randomizer* self, uint32_t seed);
```

Listing 1: C code

```cpp
class Randomizer {
public:
  uint32_t array[32];
  uint32_t carry;
  uint32_t index;

  void seed(uint32_t seed);
};
```

Listing 2: C++ code

Objects

Our module looks really like an object, in the *object-oriented* sense

```c
double
Randomizer_nextUniform(Randomizer* self) {
  double y;

  y = Randomizer_next(self);
  return y / 4294967295.0;
}
```

Listing 3: C code

```cpp
double
Randomizer::nextUniform() {
  double y;

  y = next();
  return y / 4294967295.0;
}
```

Listing 4: C++ code

Objects

How far can we go in that direction ?

A generic dynamic string

Let's create a dynamic string module, *DString*

- *cat* append a single character
- *strcat* append a character string
- *clear* set the size to 0
- *copy* copy a dynamic string to another
- *size* of the string
- *value* actual character string

A generic dynamic string

The structure type for *DString*

```c
#include <stddef.h>

typedef struct {
  char* data;          /* The string, which size is increased as needed */
  size_t logicSize;    /* Size of the string */
  size_t physicSize;   /* Size of the 'data' array */
} DString;
```

A generic dynamic string

The functions of the module

```
extern void
DString_cat(DString* self, char c);


extern void
DString_strcat(DString* self, const char* str);


extern void
DString_clear(DString* self);


extern void
DString_copy(DString* self, const DString* src);


extern const char*
DString_value(const DString* self);


extern size_t
DString_size(DString* self);
```

Resource management

But we need 2 more things

1. Initializing the module (allocate array)
2. Terminating the module (free array)

```c
#include <stddef.h>

typedef struct {
  char* data;         /* The string, which size is increased as needed */
  size_t logicSize;   /* Size of the string */
  size_t physicSize;  /* Size of the 'data' array */
} DString;
```

Resource management

Let's add a *init* and *destroy* function

```
extern void
DString_init(DString* self);

extern void
DString_destroy(DString* self);
```

Resource management

Let's add a *init* and *destroy* function

```c
#define DString_INITIAL_PHYSIC_SIZE ((size_t)4)

void
DString_init(DString* self) {
  self->logicSize = 1;
  self->physicSize = DString_INITIAL_PHYSIC_SIZE;
  self->data = (char*)malloc(sizeof(char) * self->physicSize);
  self->data[0] = '\0';
}


void
DString_destroy(DString* self) {
  free(self->data);
}
```

Usage

Using our module looks like this

```
#include "DString.h"

DString a, b;

DString_init(&a);
DString_init(&b);

DString_strcat(&a, "Hello, world");
DString_cat(&a, '\n');
DString_copy(&b, &a);
printf("b='%s', size=%u\n", DString_value(&b), DString_size(&b));

DString_destroy(&a);
DString_destroy(&b);
```

Usage

We can use our string module within an other module

```c
#include "DString.h"

typedef struct {
  DString name;
  DString address;
} MyModule;


extern void
MyModule_init(MyModule* self);


extern void
MyModule_destroy(MyModule* self);
```

Usage

We can use our string module within an other module

```
void
MyModule_init(MyModule* self) {
  DString_init(&(self->name));
  DString_init(&(self->address));
}


void
MyModule_destroy(MyModule* self) {
  DString_destroy(&(self->name));
  DString_destroy(&(self->address));
}
```

Constructor, destructor

init and *destroy* are similar to *constructors* and *destructors* from common object-oriented languages

- Simple resource management (memory, files, ...)
- works best with a consistent name scheme
- allow modules composition and re–usage

Polymorphism

We are getting close to an object-oriented approach, in pure *C*. But we miss *polymorphism*.

String validator

Let's say that we want a string validation system, to check if we have

- a valid email address
- a valid *URL*
- a valid *IP* address
- ...

String validator

In *C++*, very simple, we have polymorphism

```
class Validator {
public:
  virtual ~Validator();

  virtual bool isValid(const std::string& inStr) = 0;
};
```

String validator

In *C++*, very simple, we have polymorphism

```
class URLValidator : public Validator {
public:
  URLValidator();

  virtual ~URLValidator();

  virtual bool isValid(const std::string& inStr);
};


class EmailValidator : public Validator {
public:
  EmailValidator();

  virtual ~EmailValidator();

  virtual bool isValid(const std::string& inStr);
};
```

Delegation

C does not offer class polymorphism. But we can use *delegation* to the same effect

Delegation

Function pointers are put in a *delegate* struct

```c
struct s_Validator;

typedef struct s_Validator Validator;



typedef struct {
  /* Called to validate a string */
  int(*isValid)(Validator* self, const char*);

  /* Called upon destruction */
  void(*destroy)(Validator* self);
} ValidatorDelegate;



struct s_Validator {
  void* data;
  ValidatorDelegate delegate;
};
```

Delegation

The *Validator* module will use call the functions of the *delegate*

```
int
Validator_isValid(Validator* self, const char* str) {
  return self->delegate.isValid(self, str);
}


void
Validator_destroy(Validator* self) {
  return self->delegate.destroy(self);
}
```

Delegation

The *EmailValidator* module implementation

```
int
EmailValidator_delegate_isValid(Validator* self, const char* str) { ... }

void
EmailValidator_delegate_destroy(Validator* self) { ... }


ValidatorDelegate EmailValidator_delegate = {
  EmailValidator_delegate_isValid,
  EmailValidator_delegate_destroy
};


void
EmailValidator_init(Validator* self) {
  self->data = ...;
  self->delegate = EmailValidator_delegate;
}
```

Delegation

The *URLValidator* module implementation

```
int
URLValidator_delegate_isValid(Validator* self, const char* str) { ... }

void
URLValidator_delegate_destroy(Validator* self) { ... }


ValidatorDelegate URLValidator_delegate = {
  URLValidator_delegate_isValid,
  URLValidator_delegate_destroy
};


void
URLValidator_init(Validator* self) {
  self->data = ...;
  self->delegate = URLValidator_delegate;
}
```

Usage

Polymorphic objects in pure *C* !

```
Validator validator;

EmailValidator_init(&validator);
/* URLValidator_init(&validator); */

if (Validator_isValid(&validator, "sunwukong@kuaigui.org")) {
  ...
}
else {
  ...
}

Validator_destroy(&validator);
```

Object systems

Several toolkits provides complete object framework for *C*

- *GTK* and *GObject* 200000 LOCs
- *COS* "*C Object System*" 7000 LOCs
- ...

GTK

It's the object system to code Gnome and all its applications

- dynamic (can build new type at run-time)
- very complete ⇒ garbage collection, inheritance, containers, signals, ...
- complex need ⇒ huge code

But so tedious to code with, people created a special extension of *C* for it ⇒ *Vala*

COS

It's a general purpose object system

- dynamic (can build new type at run-time)
- very complete
- design documented in publications
- short enough to study the whole code in a few days
- very efficient (faster than Objective-C code)

Can be very nice for a large *C* project

A lightweight approach

The approach I propose here is *not* a full object framework

- very light-weight
- very efficient
- just a way to organize your code
- enough to implement most design pattern

Closer to the definition of *agile*, appropriate for small to medium projects

Table of Contents

Error handling

Handling of error case in C is very tedious

```
FILE* file;

file = fopen("spam.txt", "r");
if (!file) {
  fprintf(stderr, "unable to open file : %s\n", strerror(errno));
  return 0;
}

if (fread(&nbElements, sizeof(nbElements), 1, file) != 1) {
  fprintf(stderr, "read error : %s\n", strerror(errno));
  fclose(file);
  return 0;
}

for(i = 0; i < nbElements; ++i) {
  if (fread(&element, sizeof(element), 1, file) != 1) {
    fprintf(stderr, "read error for element %d : %s\n", i, strerror(errno));
    fclose(file);
    return 0;
  }

  ...
}

...
```

Error handling

When an error occur, you might want to

- free some resources
- do some action and try again
- consider the error fatal and quit

Error handling

To handle errors, you have to check each return value all the time

- lot of repeated code
- makes the code harder to maintain
- somebody *will* forget to test a value

Exceptions

Most object-oriented languages propose *exceptions* for this

```
try {
  /*
    Do some actions, which might trigger IO errors
   */
}
catch(IOException e) {
  /*
    Do some actions in case of IO error
   */
}
```

setjmp.h

C standard library provides a *non-local jump* mechanism

```c
int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int value);
```

setjmp.h

longjmp restore a state of the function call stack, as stored previously by a *setjmp* call

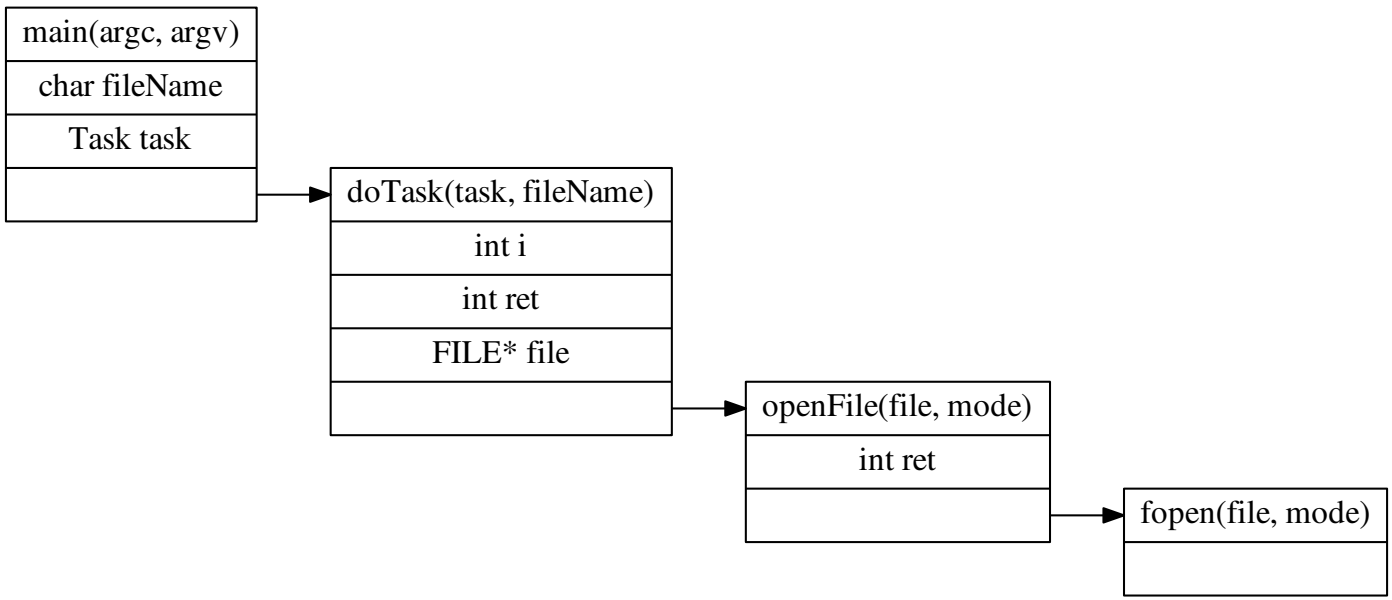

Figure : Example of call stack

setjmp.h

setjmp records into a variable the current state of the function call stack

- first call, just return 0 and save the call stack state
- after a *longjmp* call, returns a value specified my *longjmp*

Simple example

1. start in *main*
2. save state
3. goes in *first*
4. goes in *second*
5. print “second”
6. restore state
7. print “main”

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void
second() {
  printf("second\n");
  longjmp(buf, 1);
}

void
first() {
  second();
  printf("first\n");
}

int
main(int argc, char* argv) {
  if (!setjmp(buf))
    first();
  else
    printf("main\n");

  return 0;
}
```

Exception in C

We can use this to implement exceptions in C

Error module

Let's create an error module

```c
#include <setjmp.h>
#include "DString.h"

typedef struct {
  jmp_buf state;
  DString message;
} Error;



extern void
Error_init(Error* self);

extern void
Error_destroy(Error* self);

extern int
Error_try(Error* self);

extern void
Error_throw(Error* self);

extern void
Error_setMessage(Error* self, const char* fmt, ...);

extern const char*
Error_getMessage(Error* self);
```

Error module

- *init* constructor
- *destroy* destructor
- *throw* called to report an error
- *try* to initialize error reporting
- *setMessage* to set the error message
- *getMessage* to get the error message

Error module

The constructor and destructor of the module

```
void
Error_init(Error* self) {
  DString_init(&(error->message));
}

void
Error_destroy(Error* self) {
  DString_destroy(&(error->message));
}
```

Error module

Message handling

```
void
Error_setMessage(Error* self, const char* fmt, ...) {
  DString_clear(&(self->message));
  DString_fcat(&(self->message), fmt, ...);
}

const char*
Error_getMessage(Error* self) {
  return DString_value(&(self->message));
}
```

(variable argument list handling not shown here)

Error module

The *try* and *throw* functions

```
int
Error_try(Error* self) {
  DString_clear(&(self->message));
  return !setjmp(self->state);
}

void
Error_throw(Error* self) {
  longjmp(self->state, 1);
}
```

Usage

We can now implement functions that will trigger errors

```
FILE*
kg_fopen(const char* path, const char* mode, Error* error) {
  FILE* file;

  file = fopen(path, mode);
  if (!file) {
    Error_setMessage(error, "unable to open file '%s' : %s\n", strerror(errno));
    Error_throw(error);
  }

  return file;
}
```

Usage

We can now implement functions that will trigger errors

```
void
kg_fread(void *ptr, size_t size, size_t n, FILE* file, Error* error) {
  size_t ret;

  ret = fread(ptr, size, n, file);
  if ((ferror(file)) || (ret != n)) {
    Error_setMessage(error, "read error : %s\n", strerror(errno));
    Error_throw(error);
  }
}
```

Usage

We have now a nice error handling system

```
FILE* file;
Error error;

Error_init(&error);

file = 0;
if (Error_try(&error)) {
  file = kg_fopen("spam.txt", "r", &error);

  kg_fread(&nbElements, sizeof(nbElements), 1, file, &error);
  for(i = 0; i < nbElements; ++i) {
    kg_fread(&element, sizeof(element), 1, file, &error);
    ...
  }
}
else {
  if (file)
    fclose(file);

  fprintf(stderror, "%s\n", Error_getMessage(&error));
  return 0;
}

...

Error_destroy(&error);
```

Usage

We obtained the following features

- Error treatment in one place
- No duplication of return value checking code
- Easier maintenance

Much more “agile” !

Table of Contents

Agile Programming & C

You can get agile code in *C* too

- proper naming scheme
- modular code (objects systems)
- proper error handling
- all OOP techniques we reviewed before

Agile Programming & C

If memory is not a problem, you might consider use a garbage collector, like *GC*

- much less memory management to code
- in most cases, no performance loss

But in low-memory environments or if you need very tight control, forget it ...