CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

How people use C in a professional way

Tools

Methods

Programming isn't simply about writing tests, loops, assignments and function calls. The development of big programs is organized as "projects", with everybody having specific tasks to solve. Without getting into the details of an IT project, we are going to see a very small part of the huge ecosystem that you have around a programming language, talk about some of the many practical issues and see the kind of tools and methods that are useful in the real world.

Let's start with simple things.

```
#include
#include "func.h"

#define
```

```
code of functions
```

```
main()
```

Function prototypes

func.h

One good way to organize your code is to write function prototypes into an included file (double quotes mean "look for the file in the current directory") rather than directly in your program.

You haven't only function prototypes in a header file.

But we also have

Structures

Constants

This way of working reminds of Object Oriented programming - structures correspond to the attributes of classes, and function prototypes to methods.

Kind of boring mathematical example

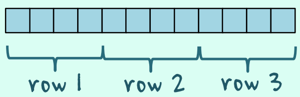
Let's see how to do it in practice

Where you would have a "matrix" class in an object-oriented language you will have a matrix structure in C.

```
typedef struct matrix {
           short rows;
           short cols;
           double *cells;
        } MATRIX_T;
```

You may dynamically allocate memory for every matrix, storing linearly what will be seen as two-dimensional.

```
typedef struct matrix {
           short rows;   3
           short cols;   4
           double *cells;
        } MATRIX_T;
```



```c
typedef struct matrix {
          short rows;
          short cols;
          double *cells;
        } MATRIX_T;

MATRIX_T *new_matrix(int rows, int cols);
void      free_matrix(MATRIX_T *m);

MATRIX_T *matrix_add(MATRIX_T *m1, MATRIX_T *m2);
MATRIX_T *matrix_scalar(MATRIX_T *m, double lambda);
MATRIX_T *matrix_mult(MATRIX_T *m1, MATRIX_T *m2);
MATRIX_T *matrix_inv(MATRIX_T *m);
double    matrix_det(MATRIX_T *m);
```

But you will have a lot of functions associated with your structure.

```c
#include "matrices.h"

#include "otherstuff.h"
```

You may have a problem if you include a header file and include something else that also requires it.

```c
#include "matrices.h"
```

multiple inclusions are a problem

You sometimes need to include a .h in a .h to get type definitions you want to use in your own structures, for instance (eg time_t).

```c
#ifndef MATRICES_H
#define MATRICES_H // No value required

typedef struct matrix {
          short rows;
          short cols;
          double *cells;
        } MATRIX_T;
MATRIX_T *new_matrix(int rows, int cols);
void      free_matrix(MATRIX_T *m);
MATRIX_T *matrix_add(MATRIX_T *m1, MATRIX_T *m2);
MATRIX_T *matrix_scalar(MATRIX_T *m, double lambda);
MATRIX_T *matrix_mult(MATRIX_T *m1, MATRIX_T *m2);
MATRIX_T *matrix_inv(MATRIX_T *m);
double    matrix_det(MATRIX_T *m);
#endif // ifndef MATRICES_H
```

The solution is to define a special symbol and make the inclusion of everything dependent on the non-existence of this symbol.

PREPROCESSOR (partial list)

#include

#define
- Constant value
- "flag" name to avoid multiple inclusions
- macros

Symbols are just a clever use of the preprocessor.

The preprocessor just substitutes text ...

Macros

The preprocessor also allows to define macros that look like functions.

```
#define  _max(a, b)   (a > b ? a : b)
```

Personal habit

```
...
maxval = _max(val1, val2);
...
```

Macros

```
#define  _max(a, b)   (a > b ? a : b)
```

After preprocessing, the macro will be simply replaced by its definition. "parameters" are replaced by the actual values.

```
...
maxval = (val1 > val2 ? val1 : val2);
...
```

As it's pure text replacement, always enclose your macro between parentheses to avoid unwanted side-effects.

Macros

Can include blocks between curly brackets

Can be on several lines

There MUSTN'T be any space behind

```
#define _init(some_struct)   { \
                  some_struct.a = 0; \
                  some_struct.b = 0; \
                  some_struct.str = ""; }
```

In a function, you'd normally use a pointer to a structure.

Have you noticed anything?

BIG difference between macros and functions

Macros are just substitution.

No call.

No copy of arguments.

Operating directly on variables.

This is why you want parentheses (or sometimes braces) around your macros.

```c
#include <stdio.h>

//#define _sum(a, b)      a + b

int main() {
    int x = 3;
    int y = 5;

    printf("%d\n", 3 * x + y * 10);
    // printf("%d\n", 3 * _sum(x, y) * 10);
    return 0;
}
```

59

9 50

You would have expected 240

Macros are just aliases or abbreviations. They are prone to side-effects, because there is no isolation as with functions that only know their parameters and know nothing of the caller.
On the other hand, functions require storing information into the stack, retrieving it, storing the result, jumping between adresses. It's extremely quick but if you call a function several millions or billions of times a day, those minuscule delays add up. They don't exist in macros, because after the preprocessor has done its job, it's simply as if you had typed the expression yourself in the program where the macro was used. Some standard library "functions" are sometimes macros.

Examples of good cases for macros

```c
#define _skip_spaces(p) {if (p){while(isspace(*p)) {p++;}}}

#define _right_trim(p)  {if (p){int len__ = strlen(p);\
                         while(len__ && isspace(p[len__ - 1])){\
                                len__--;\
                         }\
                         p[len__]='\0';}}
```

These are examples of macros that can be quite useful (especially to clean-up a string read by fgets()). Operations on multi-byte characters are also examples where a handful of macros may help a lot. You can create your own personal set of macros to include in your projects.

Very useful predefined variables:

`__FILE__`

`__LINE__`

I must mention two predefined variables that are set by the preprocessor and which I find most useful.

`__FILE__` contains the name of the current .c file. It's very useful in a project where you combine many files.
`__LINE__` is the current line number in the file.

I find these values particularly useful in debugging messages.

Very useful predefined variables:

```c
#include <stdio.h>

#define _dbg(msg)   dbgmsg(__FILE__, __LINE__, msg)

static void dbgmsg(char *fname, int line, char *txt) {
    fprintf(stderr, "%s/line %d: %s\n", fname, line, txt);
}

int main() {
    printf("Step 1\n");
    _dbg("First");
    printf("Step 2\n");
    _dbg("Second");
    return 0;
}
```

You can write a debugging function that takes file name and line number as arguments, and a macro that automatically calls it with these variables. The line number will not be defined, the one where the macro is defined, but where it's used.

Very useful predefined variables:

This is the output of the previous program, example.c:

```
$ ./example
Step 1
example.c/line 11: First
Step 2
example.c/line 13: Second
$
```

PREPROCESSOR (partial list)

#include

#define Constant value

"flag" name to avoid multiple inclusions

macros

#ifndef #ifdef

#else

#endif

The preprocessor allows for conditional compiling, which is very important when you want your program to run in several environments.

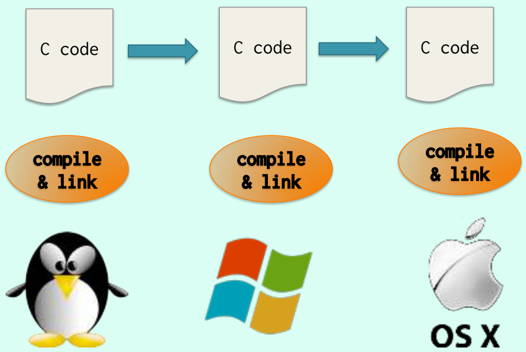
You cannot take a program and run it on different systems, even if the processor is the same.

UNLESS you are using an emulator

(WINE on Linux)

OR you are using a virtual machine

The only exceptions are emulators or virtual machines, which also include the Java Virtual Machine. But you are running in an environment on top of the native environment.



Having a (single) code that successfully builds in several environments is usually the goal.

Not that easy.

A lot of stuff is specific; just take / and \ in file names ...

`unistd.h` Only on Unix-like systems

`conio.h` Only on Windows

You also have header files that exist in some environments and not in other environments, usually because they define functions strongly linked to the operating system.

The closer you get to the system, the more differences you find.

Graphical interfaces

File-related stuff (UNIX links)

Very low-level stuff

Yet some people manage it.

These products are available on Windows, Linux and Mac.

GCC compiler

data management SQLite

database server PostgreSQL

http server

image editing

And many, many others ...

The secret weapon:

Conditional compiling

- __MSDOS__
- _WIN32 *(both 32 and 64)*
- _WIN64
- __CYGWIN__
- __APPLE__
- __linux__
- __gnu_linux__

All environments have at least one preprocessor symbol that is only defined in them.

Full list here.

http://sourceforge.net/p/predef/wiki/OperatingSystems/

```
#include <stdio.h>

int main() {
#ifdef _WIN32
    printf("I'm running on Windows\r\n");
#endif
#ifdef __APPLE__
    printf("I'm running on a Mac\n");
#endif
#ifdef __linux__
    printf("I'm running on Linux\n");
#endif
    return 0;
}
```

This can be compiled in all three environments.
Only one `printf()` (the good one) will remain after preprocessing.

Reminder: make

In a Linux environment, "make" is usually the tool you use for building programs. If "make" itself has been ported to multiple environments, you may also have problems with your makefile. Sometimes you may need some special libraries, or at least version xxx of a library. You have external dependencies, and these should be taken into account as well.

The GNU build system

GNU is an organisation that directly comes from the Free Software Foundation, founded by Richard Stallman, and is famous for its excellent and free products.

1983/1984

GCC GNOME

FREE SOFTWARE FOUNDATION

Richard Stallman
(born 1953)

The GNU build system

GNU has also produced a set of tools collectively known as "Autotools" to help with portability.

These tools are widely used in the Open Software community, and whenever you are offered the choice of installing some program "from the source", chances are that you'll have to type "configure", an "Autotools" command.

"Autotools"

The GNU build system
Using it

```
$ configure
$ make
$ make install
$
```

Using the GNU build system to install some software on your own machine is easy. You unzip, move to a directory and basically have three commands to type (each of them generates a lot of output and "make" can take time)

The GNU build system
Preparing it

Packaging your project is tougher.
A tool named "autoscan" analyses the source files and generate a "configure.scan", that you must inspect and possibly modify before renaming it "configure.ac"

autoscan [<*directory*>]

Scans all .c and .h files

configure.scan autoscan.log

The GNU build system

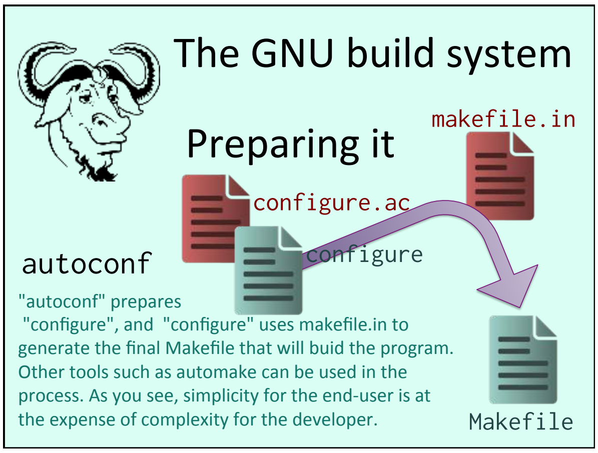
Preparing it

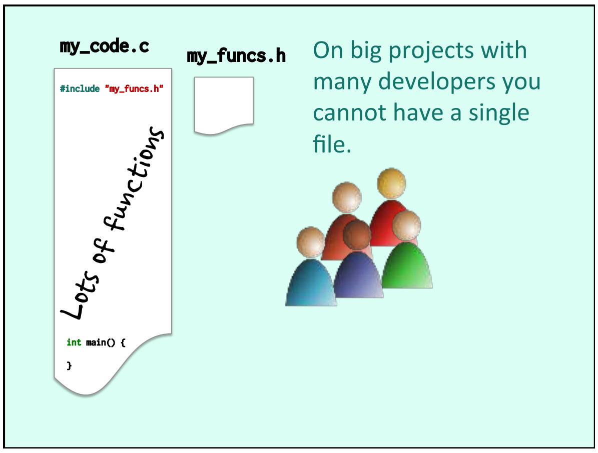
makefile.in

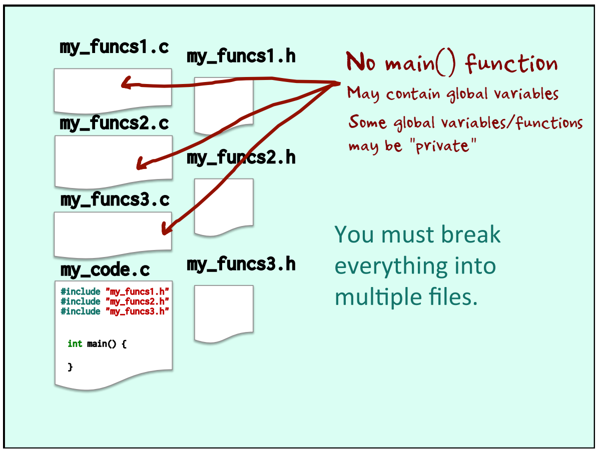
You must also manually prepare a "makefile.in" (for bigger projects you can have a top-level "Makefile.am")

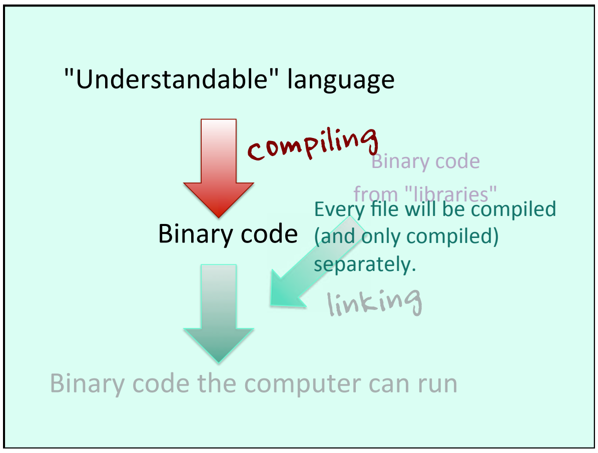
configure.ac

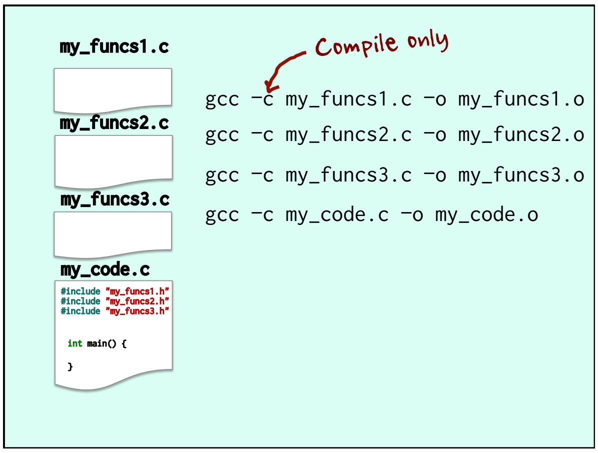
configure.scan

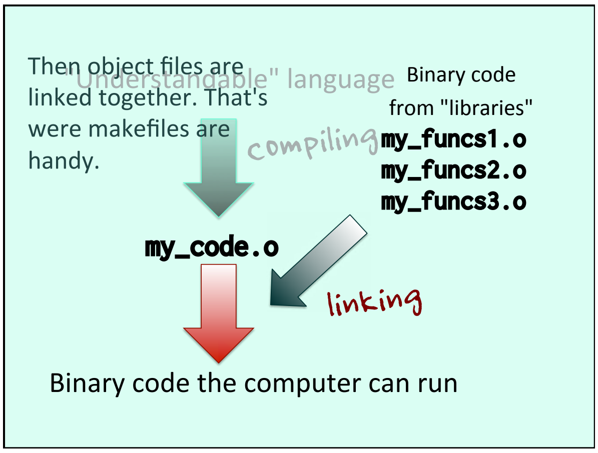

The GNU build system
Preparing it
makefile.in
configure.ac
configure
autoconf
"autoconf" prepares "configure", and "configure" uses makefile.in to generate the final Makefile that will buid the program. Other tools such as automake can be used in the process. As you see, simplicity for the end-user is at the expense of complexity for the developer.
Makefile

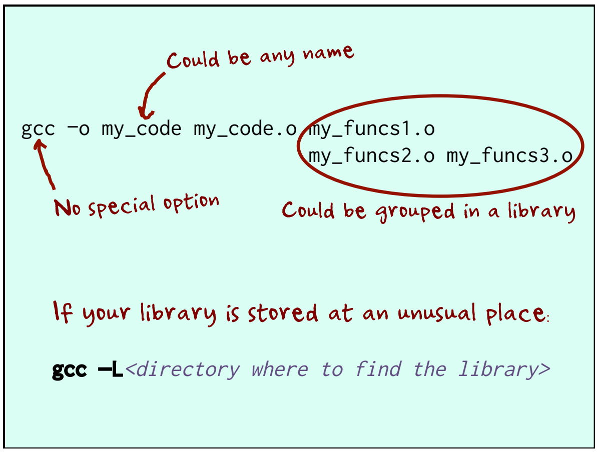

my_code.c
my_funcs.h
#include "my_funcs.h"
Lots of functions
int main() {
}
On big projects with many developers you cannot have a single file.

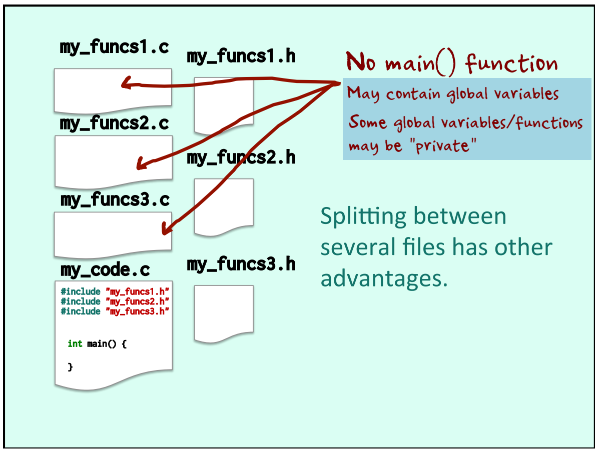

my_funcs1.c
my_funcs1.h
No main() function
May contain global variables
Some global variables/functions may be "private"
my_funcs2.c
my_funcs2.h
my_funcs3.c
my_funcs3.h
my_code.c
#include "my_funcs1.h"
#include "my_funcs2.h"
#include "my_funcs3.h"
int main() {
}
You must break everything into multiple files.


"Understandable" language
compiling
Binary code from "libraries"
Binary code
Every file will be compiled (and only compiled) separately.
linking
Binary code the computer can run


my_funcs1.c
my_funcs2.c
my_funcs3.c
my_code.c
#include "my_funcs1.h"
#include "my_funcs2.h"
#include "my_funcs3.h"
int main() {
}
Compile only
gcc -c my_funcs1.c -o my_funcs1.o
gcc -c my_funcs2.c -o my_funcs2.o
gcc -c my_funcs3.c -o my_funcs3.o
gcc -c my_code.c -o my_code.o


Then object files are linked together. That's were makefiles are handy.
"Understandable" language
Binary code from "libraries"
compiling
my_funcs1.o
my_funcs2.o
my_funcs3.o
my_code.o
linking
Binary code the computer can run


Could be any name
gcc -o my_code my_code.o my_funcs1.o my_funcs2.o my_funcs3.o
No special option
Could be grouped in a library
If your library is stored at an unusual place:
gcc -L<directory where to find the library>


my_funcs1.c
my_funcs1.h
my_funcs2.c
my_funcs2.h
my_funcs3.c
my_funcs3.h
my_code.c
#include "my_funcs1.h"
#include "my_funcs2.h"
#include "my_funcs3.h"
int main() {
}
No main() function
May contain global variables
Some global variables/functions may be "private"
Splitting between several files has other advantages.

```
char G_output[MAX_LEN];

char *initcap(char *input) {
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     G_output[i] = '\0';
     return G_output;
  }
  return NULL;
}
```

stringutil.c

If you remember, you can't declare an array in a function and return a pointer to it, because it is stored in the (volatile) stack. The array must be either global ...

```
char *initcap(char *input) {
  static char output[MAX_LEN];
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```

Static ...

... but also PRIVATE to the function

... or static. In both cases memory will be reserved in a more permanent area.

static has two meanings

for data, it means memory reserved in the data area of memory (not in the stack)

it ALSO means not visible from the outside.

The second meaning also applies to functions.

```
char G_output[MAX_LEN];

char *initcap(char *input) {
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     G_output[i] = '\0';
     return G_output;
  }
  return NULL;
}
```

stringutil.c

```
static char G_output[MAX_LEN];

char *initcap(char *input) {
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     G_output[i] = '\0';
     return G_output;
  }
  return NULL;
}
```

stringutil.c

Only global within

Opposite of **static** is **extern**

default

It also works for functions

(**static** has the *private* meaning)

module.c

```
#include ...

static int specific_func() {
...
}

extern int func() {
...
}
```

```
gcc -c module.c -o module.o
```

Only func() can be called from a different module.

A Linux /Unix command to see what is defined

```
$ nm -a module.o
0000000000000070 s EH_frame0
0000000000000000 T _func
0000000000000088 S _func.eh
0000000000000020 t _specific_func
00000000000000b0 s _specific_func.eh
$ nm -g module.o
0000000000000000 T _func
0000000000000088 S _func.eh
$
```

Upper case T/S: visible from the outside.

Only list "global" (callable from elsewhere) symbols.

C compared to Java

Let's see how C compares to Java, in the way you organize your code. In Java, you don't need to think too much (other than organizing your code in packages): the language, by construct, forces some kind of structure over your code. C leaves you free to organize your code as badly or as well as you want.

`class`

The `class` keyword defines, and only defines, the objects you work with, their attributes and methods, and the visibility of both for other objects than the current one..

attributes — `private` / `public`

methods — `private` / `public`

Definition only

C `header file`

The definition role is played in C by the header file. Contrary to Java, the actual code of a function (and as we shall see the same is true for C++ methods) doesn't belong to the definition that just defines the interface.

- *structures*
- *functions* — `extern`
- *constants*
- *macros*

C `header file`

Don't use extern variables

Although some standard libraries use them (I'm thinking of variable `errno` in `errno.h`) you shouldn't in your code use `extern` variables (which are necessarily `static`). Truly global variables are evil. However, in a non-multithreaded program global variables that remain local to a file are quite acceptable.

.c source file

If you consider that a .c source file is like the body of a class, then your `static` global variables are a bit like private variables for which you can have getters and setters.

global data → static

functions → static / extern

getters and setters

+ constructors, etc.

.c source file

There is one very significant difference though, which is that in Java you get a new set of private variables with each new object (remember that `new` in Java is just a C `malloc()`). In C, your static global variables should be limited to variables such as

- global flags
- references to collections, such as heads of lists or tree roots.
- file pointers, network sockets, database connections, and so forth.

.c source file

BEWARE when the code is shared between threads!

global data → static

functions → static / extern

Once again, if your code must be shared by several concurrent threads, you may run into serious issues. With threads, global variables are usually pointers.

You can have pointers to functions in structures:

```
struct my_struct {
       char *key;
       int value;
       int (*compar) (const char *key1,
                      const char *key2);
    };
```

Must be initialized

As already indicated, you can have function pointers in structures, that you must set to the address of actual functions when you initialize the structure.

C You can have pointers to functions in structures:

```
struct my_struct {
      char *key;
      int value;
      int (*compar) (const char *key1,
                     const char *key2);
   };
```

```
struct my_struct x;
x.compar = strcmp;
```

It can be done with, regular structure variables (unknown of Java, for which objects are always "references")

C You can have pointers to functions in structures:

```
struct my_struct {
      char *key;
      int value;
      int (*compar) (const char *key1,
                     const char *key2);
   };
```

Or it can be done with dynamically allocated structures as in Java

```
struct my_struct *p;
p = (struct my_struct *)malloc(sizeof(struct my_struct));
p->compar = strcmp;
```

C You can have pointers to functions in structures:

```
struct my_struct {
      char *key;
      int value;
      int (*compar) (const char *key1,
                     const char *key2);
   };
```

But the current "object" **isn't** an implicit parameter.

However, the current object is unknown of the function attached to it, there is no "this", and nothing can be private.

Using C in an Object-Oriented style

It's actually possible to code in a way that is very "Object-Oriented" with plain old C. It requires a lot of discipline, but it can be done.
Few areas are more suited to Object-Oriented programming than Graphical User Interfaces and Windows managers (part of the success of C++ is probably linked to its invention shortly before the time when character terminals disappeared). Yet, a significant number of Windows managers are written in C.

Main Windows Managers on Linux

They are all written in C or C++.

C, C++

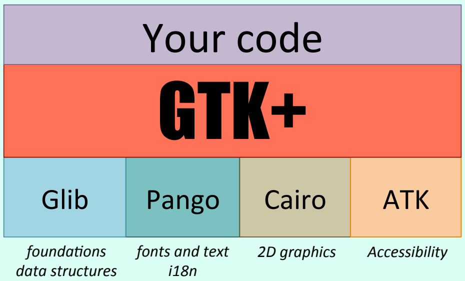
C

C

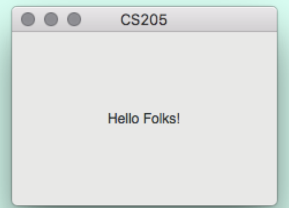
C++

C++

In particular , the Gnome Tool Kit (GTK) and its underlaying layers are all written in C.



To show to you how much GTK manages to implement Object-Oriented features I'm going to code a very simple and completely useless application.




```c
#include <stdio.h>
#include <gtk/gtk.h>

int main( int argc, char *argv[]) {
  GtkWidget *window;
  GtkWidget *label;
```

There is one header file, not tons of packages to import.

GtkWidget is the "base class" of everything, containers and widgets such as labels, entry fields of buttons. There is for everything a special "new" function that always returns a GtkWidget pointer. When you need a specific behavior, you "cast" the GtkWidget pointer to a specific type.

gtkwidget.h

```
struct _GtkWidgetClass
{
  GInitiallyUnownedClass parent_class;

  /*< public >*/

  guint activate_signal;

  /* seldomly overidden */
  void (*dispatch_child_properties_changed) (GtkWidget *widget,
                       guint        n_pspecs,
                       GParamSpec **pspecs);

  /* basics */
  void (* destroy)         (GtkWidget        *widget);
  void (* show)            (GtkWidget        *widget);
  void (* show_all)        (GtkWidget        *widget);
  ...
};
```

If you explore the header files, you'll relate GtkWidget to this structure, which contains many function pointers and is a "private part"

gtkwidget.h

```
struct _GtkWidget
{
  GInitiallyUnowned parent_instance;

  /*< private >*/

  GtkWidgetPrivate *priv;
};
```

gtktypes.h

```
typedef struct _GtkWidget              GtkWidget;
```

GtkWidget proper is a "wrapper" around the previous structure.

```
#include <stdio.h>
#include <gtk/gtk.h>

int main( int argc, char *argv[]) {
  GtkWidget *window;
  GtkWidget *label;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), "CS205");
```

Here is an example of "casting". Adding a title only makes sense for a window. The GTK_WINDOW() macro turns the GtkWidget pointer returned by gtk_window_new() into a "true" window pointer. It implements inheritance of a sort.

gtkwindow.h

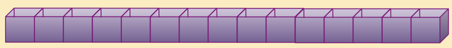
A look at the header files again looks that the cast checks that the pointer is indeed a window pointer.

```
#define GTK_WINDOW(obj) (G_TYPE_CHECK_INSTANCE_CAST ((obj), \
                          GTK_TYPE_WINDOW, GtkWindow))
```

from glib (gtype.h)

Inheritance can be achieved by organizing structures well.

Structures well organized



General → Specific

```c
#include <stdio.h>
#include <gtk/gtk.h>

int main( int argc, char *argv[]) {
  GtkWidget *window;
  GtkWidget *label;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), "CS205");
  gtk_window_set_default_size(GTK_WINDOW(window),
                              230, 150);
```

New casting for assigning a size to the Window.

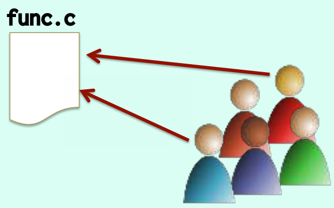
```c
  gtk_window_set_position(GTK_WINDOW(window),
                          GTK_WIN_POS_CENTER);
  label = gtk_label_new("Hello Folks!");
  gtk_container_add(GTK_CONTAINER(window), label);
  gtk_widget_show_all(window);

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
                           G_CALLBACK(gtk_main_quit),
                           NULL);

  gtk_main();

  return 0;
}
```

The Window is cast to a "container" when adding something to it, and to a mere "object" when associating the function that quits GTK to its "close" button.

Source Control Systems

There are days when everything you do is wrong

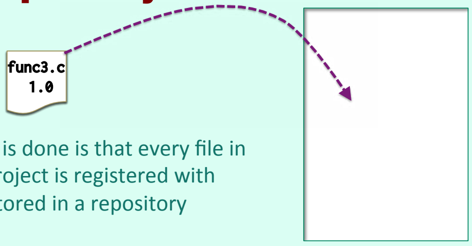
Source Control Systems



You need to coordinate developers. Everybody cannot work on the same code at the same time.

Source Control Systems

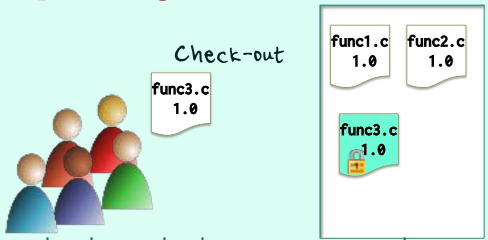
Repository



What is done is that every file in the project is registered with and stored in a repository

Source Control Systems

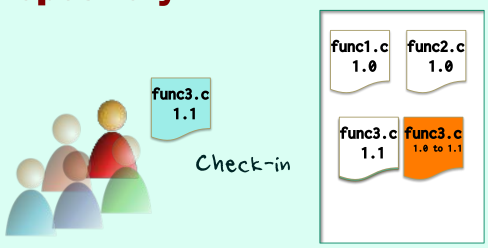
Repository



When a developer checks some source code out, it becomes read-only for the others.

Source Control Systems

Repository



Changes are recorded, and can be rolled back.

Source Control Systems

SCCS Bell Laboratories The ancestor (local)

SUBVERSION Client-server, distributed

git mercurial

Over the years, many tools have been THE tool to use. Most recent tools are thought for collaboration over the web. Principles remain the same.

Finding Bugs

```
printf(" ...", ...);
fflush(stdout);
```

Finding bugs is of course a common activity in programming, and especially in C that isn't the easiest of languages. Printing out messages (possibly between `#ifdef DEBUG`/`#endif` preprocessor instructions) is simple and effective. DON'T FORGET TO CALL `fflush()`. Otherwise the program may crash farther than you think if it crashes after it has successfully written a message but before this message was displayed/written to file. You can use format %p to display a pointer value.

Static Analysis

gcc -Wall

oclint

Finding potential bugs can be done by performing an analysis of constructs in the code that compile successfully but might in some cases prove dangerous (especially when compiling in another, possibly less forgiving, environment). I have already recommended compiling with the "-Wall" flag. Silencing all warnings (except perhaps those about unused functions when developing a project) is a good policy. The `oclint` tool is also helpful in getting industry-grade code.

Some bugs may escape a static analysis, though, and debuggers such as gdb (ddd is a graphical interface over gdb) or debuggers inside most IDE allow you tu run your code step by step, stop when a variable changes, and so forth. Slow process, but sometimes the best option.

Dynamic Analysis

gdb ddd

Visual C++ eclipse

Xcode

If you compile with -g the debugger will know the name of your variables.

Finally, some tools take particular care of memory management, and check among other things that you aren't wandering outside memory tha is reserved to you. Valgrind works like a debugger, Electric-Fence is just a library (libefence) you link with. Your program will run far slower, but every memory access will be checked. More useful than traditional debuggers in my opinion.

Dynamic Analysis

Valgrind

Electric-Fence