EPITA - Practical Programming

LSE
Security
System

Laboratory of Epita

02 - Basic Notions

Marwan.Burelle@lse.epita.fr

Overview

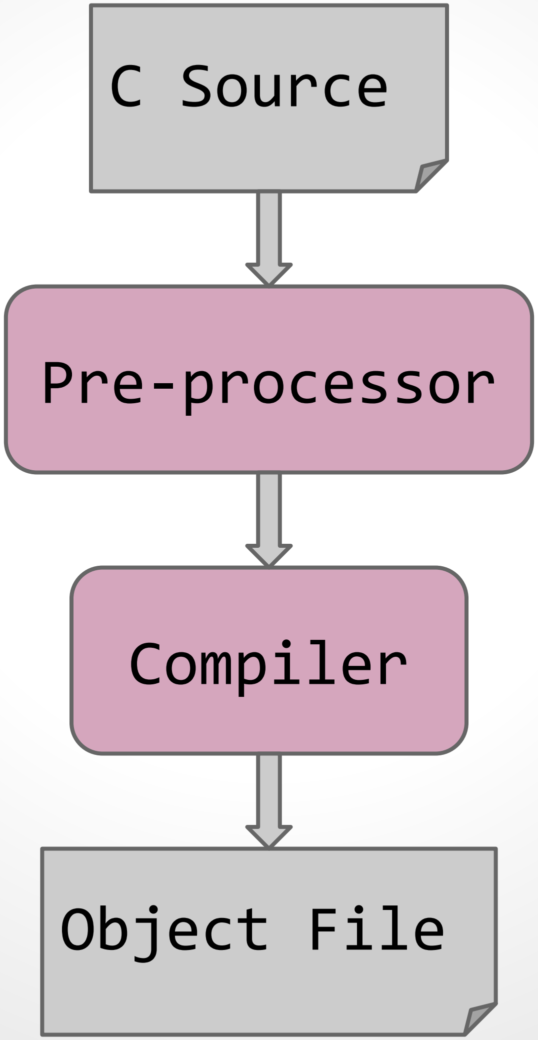
1. Multiple files projects
2. Code Samples

More than one file

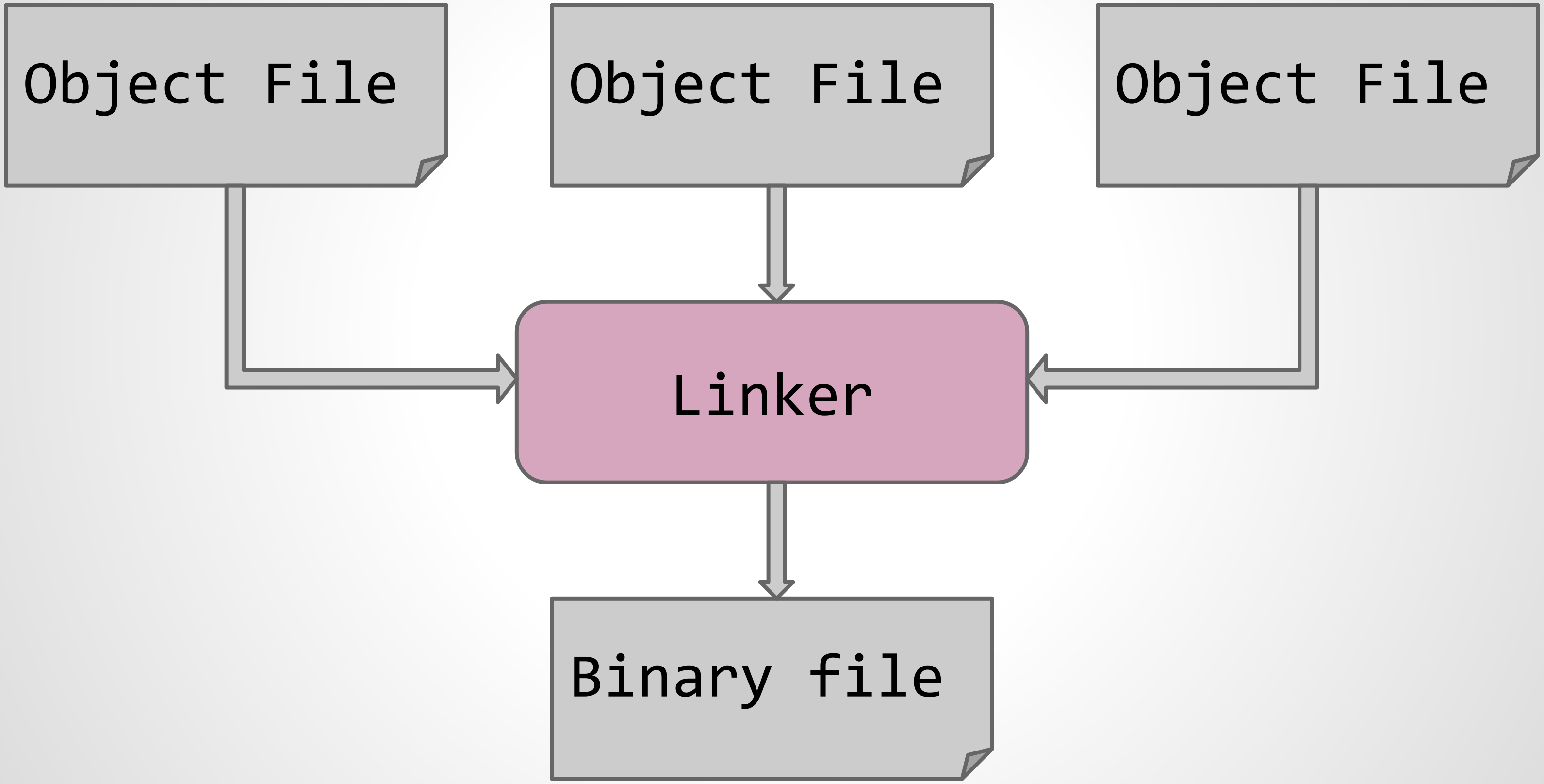
Compilation Model

1. Preprocessing: macro and includes
2. Object code compilation:
   a. Syntactic and semantic analyses
   b. IR code generation
   c. Machine code generation
   d. Assembling
3. Linking: assembling object-code files and linking symbol names to concrete memory locations.

Compilation Process



Compilation Process (link)



Forward declaration:

Provides required information in order to use a symbol without giving its full definition.

Forward declaration

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

/* forward declaration of fact */
unsigned long fact(unsigned long n);

int main()
{
  /* call for fact possible */
  printf("fact(5) : %lu\n", fact(5));
  return 0;
}

/* we need a complete def of fact, can be done here
 * or in any other file
 */

unsigned long fact(unsigned long n)
{
  unsigned long r = 1;
  for (; n > 0; n--)
    r *= n;
  return r;
}
```

Security System

Laboratory of Epita

Splitting code

- Compilation only need forward declarations
- Linking need concrete definitions

Splitting code

```c
/* main.c */
#include <stdio.h>
#include <stdlib.h>

/* forward declaration of fact */
unsigned long fact(unsigned long n);

int main()
{
  /* call for fact possible */
  printf("fact(5) : %lu\n", fact(5));
  return 0;
}
```

```c
/* fact.c */

unsigned long fact(unsigned long n)
{
  unsigned long r = 1;
  for (; n > 0; n--)
    r *= n;
  return r;
}
```

```
shell> gcc -c fact.c
shell> gcc -c main.c
shell> ls
fact.c fact.o main.c main.o
shell> gcc -o main fact.o main.o
```

Headers

- Replicating declarations is boring and dangerous
- Header files (.h) contain forward declarations
- `#include` just paste content of files
- Several strategies:
  - one header per C file
  - one big header for the project (don't scale)
  - group symbols per topic in several headers (like libc)

LSE Security System
Laboratory of Epita

How include works

```
#include "include_ex.h"

int f(int x)
{
  return x;
}
```

```
/* dummy header file */
int f(int x);

unsigned long fact(unsigned long n);

extern int global_var;

/* end of dummy header file */
```

```
shell> cpp include_ex.c
# 1 "include_ex.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "include_ex.c"
# 1 "include_ex.h" 1

int f(int x);

unsigned long fact(unsigned long n);

extern int global_var;
# 2 "include_ex.c" 2

int f(int x)
{
  return x;
}
```

Headers

```c
// file1.c
#include <stdlib.h>
#include “file1.h”

// unexported code
static int max(int a, int b)
{
  return a>b ? a : b;
}

int array_max(int tab[], long len)
{
  int           m = *tab;
  for (int *c = tab+1; c - tab < len; ++c)
    m = max(m, *c);
  return m;
}
```

```c
// file1.h
#ifndef _FILE1_H_
#define _FILE1_H_

int array_max(int tab[], long len);

#endif
```

Headers …

```c
// main.c
#include <stdio.h>
#include <stdlib.h>
#include "file1.h"

int main()
{
  int  *tab = malloc(16 * sizeof (int));
  for (int *c = tab; c - tab < 16; ++c) {
    *c = rand() % 256;
    printf("| %03d ", *c);
  }
  printf("|\n");
  printf("max = %d\n", array_max(tab,16));
  return 0;
}
```

Compiling

```
> ls
file1.c file1.h main.c
> gcc -Wall -Wextra -std=c99 -c file1.c
> gcc -Wall -Wextra -std=c99 -c main.c
> ls
file1.c file1.h file1.o main.c main.o
> gcc -o main file1.o main.o
> ls
file1.c file1.h file1.o main main.c main.o
```

Using Make correctly

https://slashvar.github.io/2017/02/13/using-gnu-make.html

- GNU Make is clever enough for the job
- You don't need more than dependencies
- Keep Makefiles as simple as possible
- Don't write rules !

Makefile

```
# Not So Simple Makefile

# Vars
CC= gcc
CPPFLAGS= -MMD                      # Flags for preprocessor
CFLAGS= -Wall -Wextra -std=c99 -O2 # Flags for the compiler
LDFLAGS=                           # Flags for the linker, usually empty
LDLIBS=                            # Listings libs

SRC= file1.c main.c                # Source files
OBJ= ${SRC:.c=.o}                  # object files
DEP= ${SRC:.c=.d}                  # dependency files

# Default rule: just ask for bin main
all: main

# main depends on object files, that's all we need
main: ${OBJ}

# cleaning rule
clean:
      ${RM} ${OBJ} ${DEP} main

# includes deps
-include ${DEP}

# END of Makefile
```

Project structures

Code splitting strategies:

- Keep files short
- Each unit must be self contained
- Split on functionalities
- Each header describes the API of the unit
- Each unit should be testable on its own

LSE Security System

Laboratory of Epita

Splitting …

Classical kinds of units:

- ➢ Data structure:

  a unit containing the data definition and the associated algorithms

- ➢ I/O operations or serialization/deserialization
- ➢ *Big algo* : an important algo with its sub-ops

Splitting ...

- Think first
- Split in large blocks of functionalities
- Establish API before coding
- Don't forget integration
- Test as soon as possible and as often as possible

More code

Arrays

```c
int array_sum(int array[], size_t size)
{
  int sum = 0;
  for (size_t i = 0; i < size; i++)
    sum += array[i];
  return sum;
}

int array_max(int array[], size_t size)
{
  int mval = array[0];
  for (size_t i = 1; i < size; i++)
    mval = array[i] > mval ? array[i] : mval;
  return mval;
}

size_t array_max_index(int array[], size_t size)
{
  size_t mpos = 0;
  for (size_t i = 1; i < size; i++) {
    if (array[i] > array[mpos])
      mpos = i;
  }
  return mpos;
}
```

```c
int is_sorted(int array[], size_t size)
{
  size_t i;
  for (i = 0; i < size - 1 && array[i] <= array[i + 1]; i++)
    continue;
  return i == size - 1;
}

int is_present(int array[], size_t size, int x)
{
  for (size_t i = 0; i < size; i++) {
    if (array[i] == x)
      return 1;
  }
  return 0;
}
```

Matrix

```
int matrix_get(int m[], size_t lines, size_t cols, size_t i, size_t j)
{
  assert(i < cols);
  assert(j < lines);
  return m[i + j * cols];
}

void matrix_sum(int m1[], int m2[], int r[], size_t lines, size_t cols)
{
  for (size_t j = 0; j < lines; j++) {
    for (size_t i = 0; i < cols; i++) {
      size_t p = i + j * cols;
      r[p] = m1[p] + m2[p];
    }
  }
}

void matrix_transpose(int m[], int r[], size_t lines, size_t cols)
{
  for (size_t j = 0; j < lines; j++) {
    for (size_t i = 0; i < cols; i++) {
      r[j + i * lines] = m[i + j * cols];
    }
  }
}
```

Square root

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

unsigned int_sqrt(unsigned n)
{
  if (n < 2) return n;
  unsigned r = n;
  while (r > n / r)
    r = (r + n / r) / 2;
  return r;
}

void test_sqrt(size_t times)
{
  while (times--) {
    unsigned n = rand();
    unsigned r = int_sqrt(n);
    printf("int_sqrt(%u) = %u (%u)\n", n, r, r * r);
    assert(r * r <= n && n < (r + 1) * (r + 1));
  }
}

int main()
{
  test_sqrt(10);
  return 0;
}
```