Symbols, Compilation Units, and Pre-Processing

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

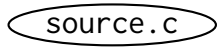
November 8, 2021

Outline

- Compilation process
- Symbols: compilation units and linking
- The C pre-processor

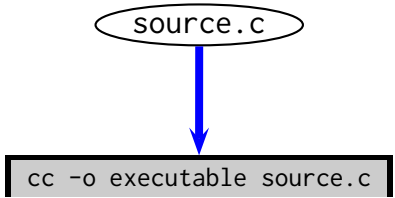
Compilation Process

- How do you compile a program?



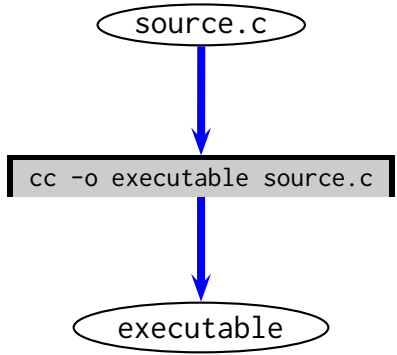
Compilation Process

- How do you compile a program?

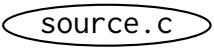


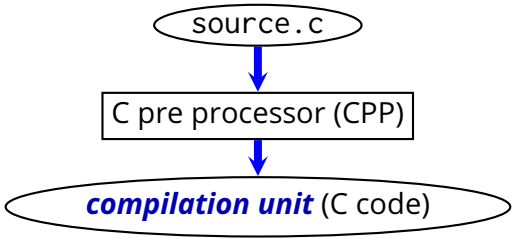
Compilation Process

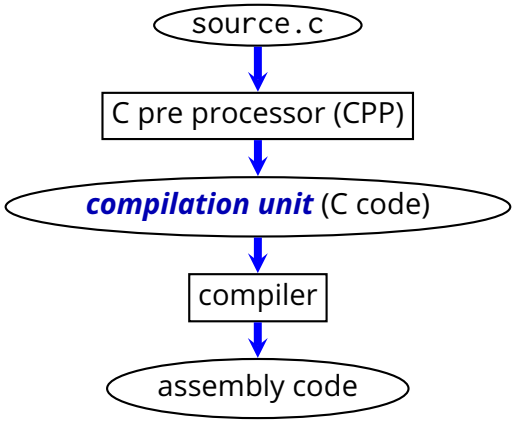
- How do you compile a program?

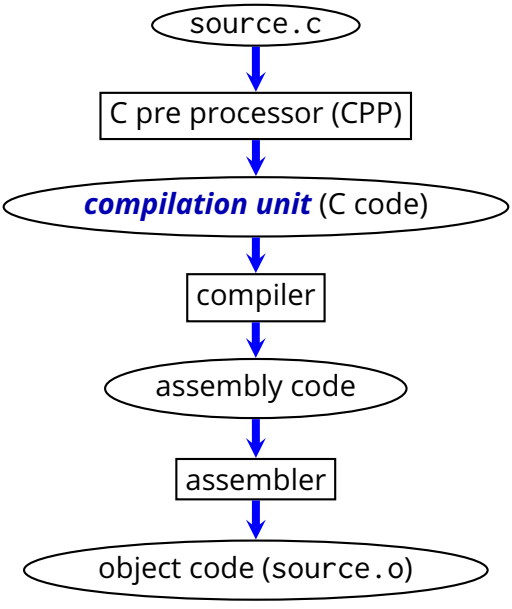


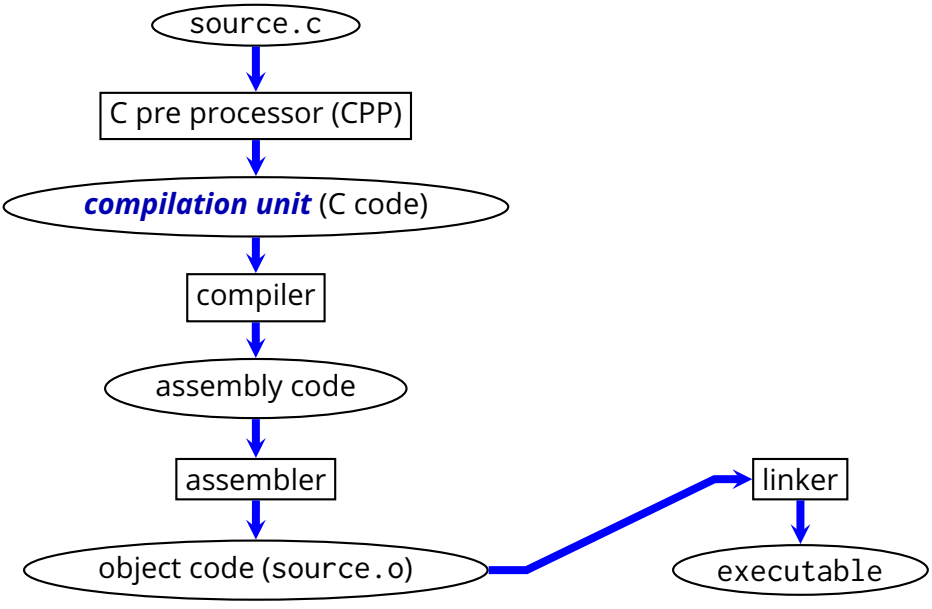
- Simple?

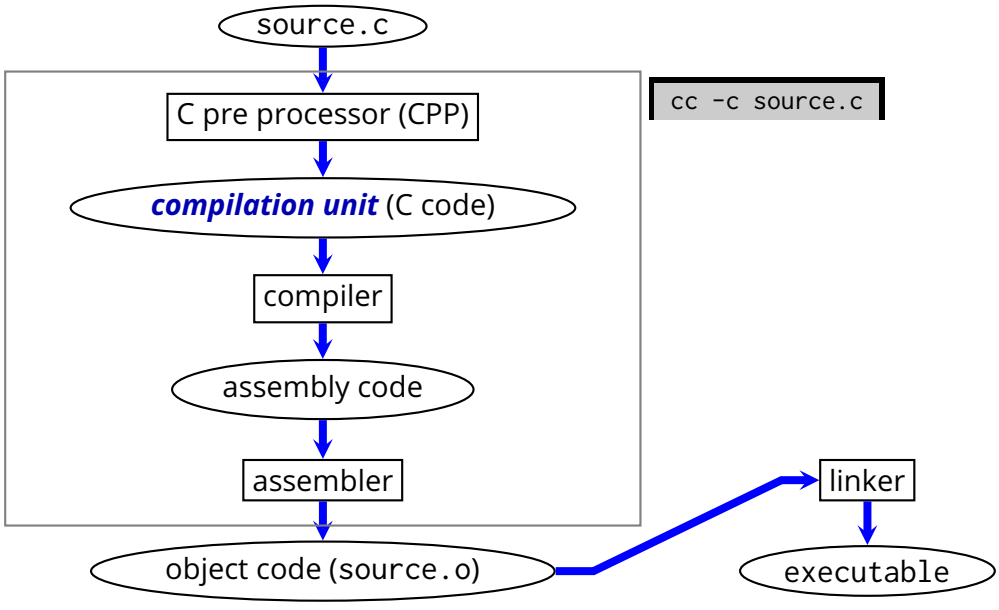

source.c

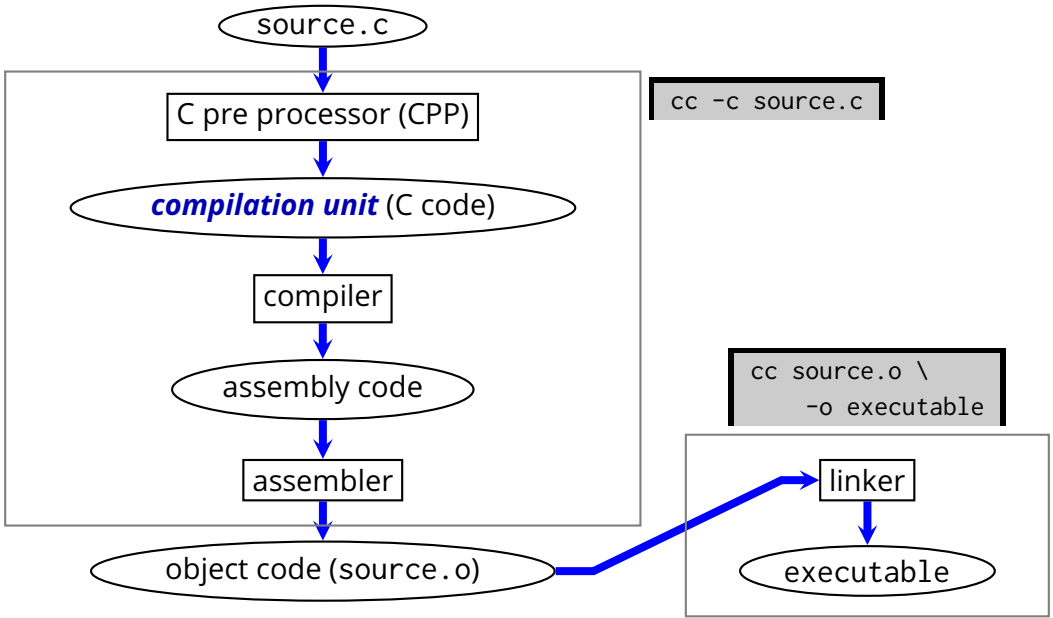

source.c
C pre processor (CPP)
compilation unit (C code)


source.c
C pre processor (CPP)
compilation unit (C code)
compiler
assembly code


source.c
C pre processor (CPP)
compilation unit (C code)
compiler
assembly code
assembler
object code (source.o)


source.c
C pre processor (CPP)
compilation unit (C code)
compiler
assembly code
assembler
object code (source.o)
linker
executable


source.c
C pre processor (CPP)
compilation unit (C code)
compiler
assembly code
assembler
object code (source.o)
cc -c source.c
linker
executable


source.c
C pre processor (CPP)
compilation unit (C code)
compiler
assembly code
assembler
object code (source.o)
cc -c source.c
cc source.o \
-o executable
linker
executable

Example

```c
#include <stdio.h>

#define ARRAY_SIZE 100

int A[ARRAY_SIZE];

void print_array(int * begin, int * end) {
    while(begin != end)
        printf("%d\n", *begin++);
}

int main() {
    int i;
    for(i = 0; i < ARRAY_SIZE; ++i)
        A[i] = 0;

    print_array(A, A + ARRAY_SIZE);
}
```

Example (commands for *gcc*)

Example (commands for *gcc*)

- To see the output of the pre-processing stage:

```
% gcc -E printarray.c
```

Example (commands for *gcc*)

- To see the output of the pre-processing stage:

```
% gcc -E printarray.c
```

- To see the assembly-code output of the compiler:

```
% gcc -S printarray.c
% cat printarray.s
```

Example (commands for *gcc*)

- To see the output of the pre-processing stage:

```
% gcc -E printarray.c
```

- To see the assembly-code output of the compiler:

```
% gcc -S printarray.c
% cat printarray.s
```

- To see the object-code:

```
% gcc -c printarray.c
% objdump -d printarray.o
```

Example (commands for *gcc*)

- To see the output of the pre-processing stage:

```
% gcc -E printarray.c
```

- To see the assembly-code output of the compiler:

```
% gcc -S printarray.c
% cat printarray.s
```

- To see the object-code:

```
% gcc -c printarray.c
% objdump -d printarray.o
```

- To see the executable:

```
% gcc printarray.c
% objdump -d a.out
```

Multi-Source Projects

- Projects almost always consist of multiple source files

Multi-Source Projects

- Projects almost always consist of multiple source files

  Example:

  `program.c`
  - *defines* `main()`
  - *declares and uses* `void print_array(int * begin, int * end)`
  - *declares, defines, and uses* `int A[100]`

  `printarray2.c`
  - *defines* `void print_array(int * begin, int * end)`

Multi-Source Projects

- Projects almost always consist of multiple source files

  Example:

  `program.c`
  - *defines* `main()`
  - *declares and uses* `void print_array(int * begin, int * end)`
  - *declares, defines, and uses* `int A[100]`

  `printarray2.c`
  - *defines* `void print_array(int * begin, int * end)`

- How do we build the `example` program?

Multi-Source Projects

- Projects almost always consist of multiple source files

  Example:

  `program.c`
  - *defines* `main()`
  - *declares and uses* `void print_array(int * begin, int * end)`
  - *declares, defines, and uses* `int A[100]`

  `printarray2.c`
  - *defines* `void print_array(int * begin, int * end)`

- How do we build the `example` program?
  1. `cc -c program.c`

Multi-Source Projects

- Projects almost always consist of multiple source files

  Example:

  `program.c`
  - *defines* `main()`
  - *declares and uses* `void print_array(int * begin, int * end)`
  - *declares, defines, and uses* `int A[100]`

  `printarray2.c`
  - *defines* `void print_array(int * begin, int * end)`

- How do we build the `example` program?
  1. `cc -c program.c`
  2. `cc -c printarray2.c`

Multi-Source Projects

- Projects almost always consist of multiple source files

  Example:

  `program.c`
  - *defines* `main()`
  - *declares and uses* `void print_array(int * begin, int * end)`
  - *declares, defines, and uses* `int A[100]`

  `printarray2.c`
  - *defines* `void print_array(int * begin, int * end)`

- How do we build the `example` program?
  1. `cc -c program.c`
  2. `cc -c printarray2.c`
  3. `cc printarray2.o program.o -o example`

Symbols

- Each ***symbol*** used in a compilation unit must be declared within that compilation unit

Symbols

- Each ***symbol*** used in a compilation unit must be declared within that compilation unit
  - variables
  - functions
  - types
  - …

Symbols

- Each ***symbol*** used in a compilation unit must be declared within that compilation unit
  - variables
  - functions
  - types
  - …
- The compiler outputs (in the object file) all symbols

Symbols

- Each ***symbol*** used in a compilation unit must be declared within that compilation unit
  - variables
  - functions
  - types
  - …
- The compiler outputs (in the object file) all symbols
  - some symbols will be ***defined*** within the object file
    - *variable* definitions specify the memory required for them
    - *function* definitions include their (machine) code

Symbols

- Each ***symbol*** used in a compilation unit must be declared within that compilation unit
  - variables
  - functions
  - types
  - …
- The compiler outputs (in the object file) all symbols
  - some symbols will be ***defined*** within the object file
    - *variable* definitions specify the memory required for them
    - *function* definitions include their (machine) code
  - some will be ***undefined***

Linking

- It is the job of the ***linker*** to put together the executable

- It is the job of the ***linker*** to put together the executable
  1. The linker reads all the object files and libraries

- It is the job of the ***linker*** to put together the executable
  1. The linker reads all the object files and libraries
  2. The linker assigns an address to each defined symbol

- It is the job of the ***linker*** to put together the executable
  1. The linker reads all the object files and libraries
  2. The linker assigns an address to each defined symbol
  3. The linker replaces every reference to a symbol (a “use”) with its actual address (“definition”)
     - i.e., it *links* uses with definitions

- It is the job of the ***linker*** to put together the executable
  1. The linker reads all the object files and libraries
  2. The linker assigns an address to each defined symbol
  3. The linker replaces every reference to a symbol (a “use”) with its actual address (“definition”)
     - i.e., it *links* uses with definitions

- Actually, not all symbols will be visible outside their object file
  - symbols defined with `static` linkage
  - `static` linkage is used for “private” variables and functions

Sharing Declarations

- What if a function `f` is used in several source files?

Sharing Declarations

- What if a function `f` is used in several source files?
  - remember that `f` must be declared in all compilation units

Sharing Declarations

- What if a function `f` is used in several source files?
  - remember that `f` must be declared in all compilation units

- It is convenient to have the declaration in one file and then to *include* that file in every compilation unit

Sharing Declarations

- What if a function `f` is used in several source files?
  - remember that `f` must be declared in all compilation units

- It is convenient to have the declaration in one file and then to *include* that file in every compilation unit

- This is done by the *C pre processor*
  - e.g.,

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

    includes the "header file" `stdio.h`, which declares

    ```
    extern int printf (const char * format, ...);
    ```

    (and many other functions, types, and variables)

Sharing Declarations

Sharing Declarations

(in *person.h*)

```
struct person {
 char * name;
 int age;
};
```

Sharing Declarations

(in *person.h*)

```
struct person {
 char * name;
 int age;
};
```

```
#include "person.h"

void print(struct person * p) {
 printf("Name: %s\n",
 p->name);
}
```

Sharing Declarations

(in *person.h*)

```
struct person {
 char * name;
 int age;
};
```

```
#include "person.h"

void print(struct person * p) {
 printf("Name: %s\n",
 p->name);
}
```

```
#include "person.h"

void input(FILE * inputfile, struct person * p) {
 /* ... */
}
```

Sharing Declarations

(in *person.h*)

```
struct person {
 char * name;
 int age;
};
```

```
#include "person.h"

void print(struct person * p) {
 printf("Name: %s\n",
 p->name);
}
```

```
#include "person.h"

void input(FILE * inputfile, struct person * p) {
 /* ... */
}
```

The C Pre Processor

- Includes multiple files into a *compilation unit*
  - with `#include`

- Includes multiple files into a *compilation unit*
  - with `#include`
- Expands macros

```
#define MAX_LINE_LENGTH 1024

int main() {
    char buffer[MAX_LINE_LENGTH];

    while (fgets(buffer, MAX_LINE_LENGTH, stdin)) {
        /* ... */
    }
}
```

The C Pre Processor

- Macros with parameters

- Macros with parameters

```
#define IS_NULL(x) (x == 0)
#define NEXT(x) (x + 1)
#define MAX(x,y) (( x > y ) ? x : y)

int i = MAX(10, getchar());
int * p1 = /* ... */;
int * p2 = NEXT(p);
```

- Macros with parameters

```
#define IS_NULL(x) (x == 0)
#define NEXT(x) (x + 1)
#define MAX(x,y) (( x > y ) ? x : y)

int i = MAX(10, getchar());
int * p1 = /* ... */;
int * p2 = NEXT(p);
```

Warning: subtle problems here!

- Macros with parameters

```
#define IS_NULL(x) (x == 0)
#define NEXT(x) (x + 1)
#define MAX(x,y) (( x > y ) ? x : y)

int i = MAX(10, getchar());
int * p1 = /* ... */;
int * p2 = NEXT(p);
```

Warning: subtle problems here!

Good practice: avoid macros for *programming!*

Use them only for configuration purposes

- i.e., conditional compilation (coming up next)

- Conditionally includes lines into a compilation unit

- Conditionally includes lines into a compilation unit

```
#include "search.h"

int main(int argc, const char * argv[]) {
#ifdef USING_TST_ALGORITHM
    int result = tst_search(argv[1]);
#else
    int result = bsearch(argv[1]);
#endif
    if (result > 0)
        printf("%s is here.\n", argv[1]);
    else
        printf("who is %s?\n", argv[1]);
    return 0;
}
```

- Pre-processor symbols can be defined within the source

```
#define WITH_TST_ALGORITHM
```

- Pre-processor symbols can be defined within the source

```
#define WITH_TST_ALGORITHM
```

or they can be passed as command-line parameters to the compiler

```
cc -DWITH_TST_ALGORITHM -c test.c
```

- A more complex example

- A more complex example

```
#include "config.h"
#if (SET_SIZE > 20000) || (ALPHABET_SIZE > 256)
#ifdef WITH_TST_ALGORITHM
#undef WITH_TST_ALGORITHM
#endif
#include "big.h"
#else
#include "small.h"
#if HAVE_UNISTD_H
#include <sys/types.h>
#include <unistd.h>
#else
#error you need unistd.h to compile this program
#endif /* HAVE_UNISTD_H */
#endif /* SET_SIZE etc. */
```

Exercise

- Implement a program that reads lines from the standard input and outputs the lines in reverse order
- The program uses either a linked list or an array to store lines
  - you must implement both methods
- The pre-processor variable `WITH_ARRAY` can be passed to the compiler to select the array version
- The numeric pre-processor variable `MAX_LINE_SIZE` defines the maximum accepted line size
- The numeric pre-processor variable `MAX_INPUT_SIZE` defines the maximum number of lines accepted by the array implementation

Exercise (2)

- If you have not done so already, separate the previous implementation into three “modules”
  - the *list* module defines the list-based container
  - the *array* module defines the array-based container
  - the *main* module defines the `main` function, reads the input file, and uses one of the two container data structures to store and then print the lines