

Chapter 16

Program Organization

16.1 Steps in the compiler

- Compiler performs multiple steps when creating a program
 - Preprocessor
 - * Direct manipulation of source code (on source code level)
 - Compiler
 - * Translation of source files into machine language.
 - * Syntax errors are found in this step.
 - * Names of function calls are kept and not resolved. -This step creates object files. Library files (.dlls in Windows, .so in *nix) are such files.
 - Linker
 - * Connects function calls. If a function does not exist, this step will trigger an error.

16.2 Preprocessor Directives

16.2.1 Preprocessor

- Preprocessor: Modifies source files before (!) they are translated.
- Statements always start with #.
 - # statements are ignored by other compiler steps.
 - Are *not* terminated with a ; because they are handled before compilation.
- Example: `#include <stdio.h>`
 - `#include <filename>`: Inserts the file `filename` from system directories (depends on compiler configuration).
 - `#include "filename"`: Inserts the local file `filename`.
 - Copy-paste.

16.2.2 A weird example

- Hello World split up into two files
- Content of `main.c`

```
#include <stdio.h>

int main() {
    #include "somefile.bla"
}
```

- Content of `somefile.bla`:

```
printf("Hello World\n");
```

16.2.3 Macro definitions

- `#define identifier replacement`
 - All occurrences of `identifier` are replaced by `replacement`
 - Pure copy paste.
 - Code conventions: Usually written in ALL_CAPS.
 - `replacement` is optional.
- Useful for
 - constants (eg `BUFFER_SIZE`)
 - Conditional translation

16.2.4 (Bad) Example

- What is the output of the following program?
 - Hint: replace `FIVE` by (exactly) `1+4`.

```
#include <stdio.h>

#define FIVE 1+4

int main() {
    printf("%d %d", -FIVE, 3 * FIVE);
}
```

16.2.5 (Bad) Example cont'd

- Solution: 3 7
 - `-FIVE` becomes `-1+4`
 - `3 * FIVE` becomes `3 * 1+4`.
- For correct behavior use brackets.

```
#include <stdio.h>

#define FIVE (1+4)

int main() {
    printf("%d %d", -FIVE, 3 * FIVE); // prints -5 15
}
```

16.2.6 Useful Example

```
#define LEN 100
int a[LEN];
```

- Helps making program more readable
- Avoids inconsistencies and syntax errors
- Makes it easier to modify existing programs

16.2.7 Parameterized Macros

- Useful for tiny expressions like MAX, MIN or in the following example IS_EVEN.
- Parameters should be put into brackets because parameters are strict text replacements.

```
#define IS_EVEN(n) ((n)%2 == 0)
#define BAD_IS_EVEN(n) (n%2 == 0)

int main() {
    int i = 0;
    if(IS_EVEN(i+1)) i++;
    if(BAD_IS_EVEN(i+1)) i++; // What is the output here?
}
```

16.2.8 Conditional Translation

- Properties that are available on compile time can be used for conditional execution.
- Conditional preprocessor directives are:
 - #if, #elif, #else, #endif
 - #ifdef, #ifndef
- Allows to include or exclude parts of the program based on some conditions.
 - Eg for increased portability between operating systems

```
#if defined(WIN32)
...// code for windows
#elif defined(MAC_OS)
...// code for OSX
#elif defined(LINUX)
...// code for Linux
#endif
```

16.2.9 Conditional Translation (2)

- Based on macros, debugging statements can be masked.
 - In the following program, SHOWALL is not defined, hence the printf-statement is not executed.

```
int main() {
    for(int i = 0; i < 10; ++i) {
        #ifdef SHOWALL
            printf("%d\n", i);
        #endif
    }
    return 0;
}
```

- Compiling the code with `$ gcc theprogramabove.c -DSHOWALL` creates a macro, hence in the final program printf will be printed.

16.2.10 Conditional Translation (3)

- Introduce a default value for a macro if it is not defined (eg in some other header file).

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

16.3 C Implementation files

16.3.1 Source Code

- Implementation Files:
 - Suffix *.c
 - Contains program code
- Header Files (Includes)
 - Suffix *.h
 - Contains declarations of
 - * Constants (`#defines`)
 - * Prototypes of functions
 - * **No program code**

16.3.2 Encapsulation

- Source files can become very large and as a consequence unreadable.
 - Split large programs into logical units
 - Encapsulation: Parts of the program that depend on each other are kept in files that are separated from other files
 - Makes it easier to change implementation
 - Faster compilation by reusing object files

16.3.3 Header files

- Header files contain declarations that are used by possibly multiple implementation files
 - Access via prototypes in header files.
 - No implementation in header files
 - * Would be compiled multiple times (one time for each `#include`).
 - * Implementation in C file is only compiled once

16.3.4 Include Guard

- Header files usually include other header files
 - One header file could be included multiple times because of these dependencies, causing a compiler error.
 - Include guards prevent that a header file is included multiple times.

```
#ifndef MY_INCLUDE
#define MY_INCLUDE

// Content of header file.

#endif
```

16.3.5 Example

```
hello.h
#ifndef HELLO_H
#define HELLO_H
void hello(void);
#endif

hello.c
```

```
#include "hello.h"
#include <stdio.h>
void hello(void) {
    printf("Hello World\n");
}
```

16.3.6 Compiling a .c file

- Creates an object file (extension .o)
 - Object files are collections of procedures in machine code (format depends on operating system and is not portable to other systems)
 - Procedures are annotated with information like their name and type
 - Procedure calls inside code are *not* resolved.
 - A dynamic library (.dll in Windows, .so in OSX and Linux) is such an object file.

16.3.7 Example

File: main.c

```
#include "hello.h"
int main(void) {
    hello();
    return 0;
}
```

16.3.8 Linking object files

- Procedures and external variables are not resolved after compiling.
 - Separate program (Linker) responsible for connecting them.
- Dynamic and static binding
 - Dynamic: Program loads and binding procedures when the program is started (dll in Windows, so in Unix).
 - * Advantage: Smaller programs (not all procedures are part of the program)
 - Static: All procedures are included in the program.
 - * Faster, but not always feasible.
- Linker shows errors when procedures cannot be linked or multiple procedures exist of the same name.

16.3.9 Example

```
$ gcc -c main.c hello.c
$ ls
hello.c  hello.h  hello.o  main.c  main.o
$ gcc main.o -o main
main.o: In function `main':
main.c:(.text+0x5): undefined reference to `hello'
collect2: error: ld returned 1 exit status
$ gcc main.o hello.o -o main
```

16.3.10 Example Hello2

File: hello2.c

```
#include <stdio.h>
#include "hello.h"
void hello(void) {
    printf("Other Hello\n");
}
```

```
$ gcc main.o hello.o hello2.o -o main
hello2.o: In function `hello':
hello2.c:(.text+0x0): multiple definition of `hello'
hello.o:hello.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

16.4 Memory management

16.4.1 Stack and Heap

- Two memory segment
 - Stack: Local variables of a procedure (Stackframe)
 - * Reserved by the compiler (stackframes are stacked onto each other)
 - * Usually very fast.
 - Heap: Global variables
 - * Must be reserved at runtime
 - * suitable for larger datastructures

16.5 Storage classes of variables

- Storage classes: auto, extern, register, static, volatile
- **auto**
 - Normal behavior: Variable is created automatically and deleted at the end of its life time (this is the default behavior, therefore hardly used).
 - In C++11 **auto** is replaced.
- **register**
 - Variable should be stored in a CPU Register (very fast).
 - Often ignored by compiler (compilers are actually very good in finding such variables).
- **volatile**
 - Variable might be modified asynchronously.
 - Prevents optimizations that might otherwise cause a faulty behavior.

16.6 Extern Variables

- Globale Variablen, die von mehreren C-Files geteilt werden müssen im Header-File als **extern** deklariert werden
- Linker sorgt für Verbinden externer Variablen.
- Funktionen sind implizit immer **extern**

16.7 Statische Variablen

- Static variables in functions keep their value.
- Visibility of a **static**-Variable in a function is limited to the function (compared to global variables).
- Life time of a static variable is the whole runtime.

16.8 Example static

- When the program is called, count is initialized with 1.
- After that, its value is kept.

```
void inc() {  
    static int count = 1;  
    printf("%d\n", count);  
    count++;  
}  
  
void main() {  
    inc(); inc(); inc();  
}
```