Computer Programming

Preprocessor Directives. Modular Programming

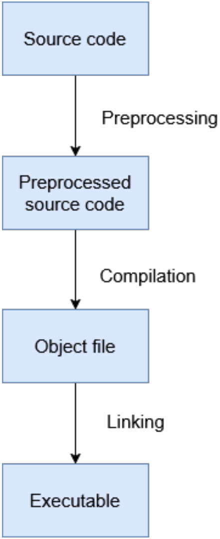
Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 5

Contents

From source code to program



- preprocessing
  - text editing operations (copy, replace)
  - the output is the preprocessed source file
- compiling
  - the syntax is checked
  - files are scanned from top to bottom
  - generates compile error in case of mistakes
  - the output is the object file
- linking
  - multiple object files are linked together
  - typically user object files with library ones
  - the output is the executable

Preprocessing directives

- they are commands for the preprocessor
- they all begin with #
- they represent text-edit operations which are performed before compilation

```
#include
#define #undef
#if #ifdef #ifndef #else #elif #endif
```

Including files

- can be done using `#include`
- the content of the included file is copied to the position where the include directive is found
- most often, it is used to include library header files
- header files contain:
  - constants, type defines, structures, enumerations
  - function prototypes
- user-defined header files should be included given using quotation marks " " = look for the files in the project folder
- library header files should be given using angular brackets <> = look for the files in the system include folder

```
#include "user_header.h"
#include <library_header.h>
```

Including files - example

```c
#include <stdio.h>
int main(){
  return 0;
}
```

The preprocessed file can be generated with the gcc compiler with the following command:

gcc −E main.c > main.pre

```c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "c:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/x86_64-w64-mingw32/include/stdio.h" 1 3
//...
int __attribute__((__cdecl__)) printf(const
char * __restrict__ _Format,...);
//...
# 3 "main.c"
int main(){
  return 0;
}
```

Symbolic constants

- can be defined using `#define`
- they represent text-replace rules
- can be employed to define compile-time constants
- replaces all occurrences of the first argument with the second argument, when it appears as a separate word

  ```
  #define to_replace replace_with
  ```
- if the second argument is longer than a line, the backspace character should appear at the end of each line
- the replace rule is valid until `#undef to_replace` is encountered, or the end of the file is reached

Preexisting symbolic constants

__DATE__ contains the date when the current file was compiled

__CDECL__ indicates that the function conforms to C language conventions

__STDC__ indicates that the function conforms to ANSI C syntax rules

__FILE__ contains the name of the current file

__FUNCTION__ contains the name of the current function

__LINE__ evaluates to the line number where it is found

Program 5.1 - Symbolic constants example

```
1  #include <stdio.h>
2
3  #define ALPHA 30
4  #define BETA ALPHA+10
5
6  int main(){
7    printf("%d\n", ALPHA);
8    printf("%d\n", BETA*BETA);
9    printf("%d\n", __LINE__);
10   return 0;
11 }
```

- $BETA * BETA$ is replaced with $30 + 10 * 30 + 10$
- *__LINE__* will be replaced with 9

Type-safe alternative to symbolic constants

- the language allows for the declaration of constants
- syntax is the same as for declaring variables
- uses the `const` keyword before the type

```
const int a = 1;
const double b = 0.5;
```

- constants must be initialized at the moment of their declaration, so the following is wrong and generates a compile error:

```
const float x;
x = 1;
```

Macros

- they also use the `#define` directive
- similar behavior to functions
- however, they are not type-safe, operator precedence is not checked, expression evaluation order becomes hard to follow
- recommended only when they are short: min/max, interchange, absolute value

```
#define macro_name(p1, p2, ...) macro_body
```

- `macro_name` behaves similarly to a function name
- p1, p2, ... are parameters - there can any number, even none
- the name and the parameters of the macro are replaced throughout the file with the macro body or until `#undef macro_name` is encountered

Program 5.2 - Macros example

```c
#include <stdio.h>
#define MIN(a,b) (((a)<(b))?(a):(b))
#define ABS1(x) (x<0)?-x:x
#define ABS2(x) (((x)<0)?-(x):(x))
#define INTER(tip,a,b) \
{tip c; c=a; a=b; b=c;}

int main(){
  int a = 1, b = 2;
  int c = MIN(a, b);
  int d = -ABS1(4-2);
  int e = -ABS2(4-2);
  INTER(int, a, b);
  printf("%d %d %d %d %d",
    a, b, c, d, e);
  return 0;
}
```

- $-(4-2<0)?-4-2:4-2$
- $-0?-4-2:4-2$
- $d=2$

Conditional compilation

- it can be achieved using `#if` and its related directives
- if the expression after the `#if` is true, the part after it is included during compilation until the next `#endif`, `#else` or `#elif`
- if the expression is false, the part between the next `#else` and `#endif` is included during compilation
- these directives do not control the execution flow
- useful if we want activate certain parts of the code before compilation

```
#if expression
    text true
#else
    text false
#endif
```

Conditional compilation

- compilation can be conditioned on the existence of a a symbolic constant using `#ifdef`
- if the expression was defined previously, the part up to the next `#endif` is included during compilation
- can use `#else` clause for alternate branch
- there is also a negated version, if not defined `#ifndef`
- most often employed to avoid multiple inclusion of header files

```
#ifdef expression
    text constant exists
#endif
```

```
#ifndef expression
    text constant undefined
#endif
```

Program 5.3 - Header guard

header.h

```c
#ifndef HEADERFILE_H
#define HEADERFILE_H

int magic = 42;

#endif
```

main.c

```c
#include "header.h"
#include "header.h"
int main(){
  printf("%d", magic);
  return 0;
}
```

- if the HEADERFILE_H constant is undefined
- define it now
- include the contents during compilation
- the first include copies the file content
- the second include doesn't copy the variable declaration because of the header guard
- multiple inclusion is a common issue for large projects

Modular programming

- it refers to dividing the source code into multiple files
- each file is an independent module
- there is a distinction between:
  - public interface = header file = this includes only the function headers, public constants and structures, and shows how the module can be employed
  - private implementation = source file = this contains the function definitions
- there are several reasons why implementation should be kept private: to hide it, to avoid repeating it, for security reasons = access to the implementation can be exploited
- modules can be included in other projects

Modular programming conventions in C

- each module has two dedicated files
  - the header file = contains public constants, definitions and function prototypes
  - the source file = contains the private implementation = function definitions
- when we want to use a module from another
  - it must include the header file with `#include`
  - it must link the corresponding object file = the source file must be included in the build process
- the source file should also include the header file
- it is not recommended to use `#include` with source files

Program 5.4 - Modular programming example

conversions.h

```
#ifndef CONVERSIONS_H
#define CONVERSIONS_H

void showbinary(long long nr);
int to_digits(long long nr, int digits[], int b);
long long from_digits(int digits[], int n, int b);

#endif
```

conversions.c

```
#include <stdio.h>
#include "conversions.h"
void showbinary(long long nr){
  if (nr > 1)
    showbinary(nr/2);
  printf("%d", nr&1);
}
```

main.c

```
#include <stdio.h>
#include "conversions.h"
int main(){
  showbinary(23);
  printf("\n");
  return 0;
}
```

Variable types - scope, visibility, access

Global variables

- they are declared outside functions
- they are accessible in the whole file after their declaration
- automatically initialized with 0
- stored in the global memory area (data segment)
- not recommended:
  - they allow access to data from multiple functions
  - changes to variables are hard to track
  - can lead to naming conflicts = multiple variables with the same name
  - you should learn how to properly send data to functions and how to save the return value

Variable types - scope, visibility, access

Local variables

- they are declared inside functions
- they live while the function is being executed
- they are not initialized
- stored on the stack
- each function call has a separate instance of them
- recommended:
  - encourages functions without secondary effects
  - implementation is encapsulated inside functions

Variable types - scope, visibility, access

Static variables

- place the `static` keyword before the declaration
- when applied to global variables
  - the variable is only visible in its own source file (private)
- when applied to local variables
  - the variable keeps its value throughout multiple function calls

External variables

- non-static global variables can be accessed from other source files
- when using them in other source files they must be declared as `extern`

Static functions

Static functions (private)

- it is possible to declare a function as static
- its access is restricted to the current file

General remarks about access modifiers:

- function and variable access specifiers are cumbersome in C
- the static keyword is overloaded and operates differently depending on the context
- more modern solutions are available in newer languages (C++, Java, Python)

Program 5.5 - Variable/function access example

static.c

```c
#include <stdio.h>
#include "static.h"

static const double PI = 3;
const double E = 2;

static void private_f(){
  puts("in private f");
}

void f(){
  private_f();
}

void add(int dx){
  static int s = 4;
  s += dx;
  printf("s = %d\n", s);
}
```

static.h

```c
void f();
void add(int dx);
```

main.c

```c
#include <stdio.h>
#include "static.h"

extern const double PI;
extern const double E;

int main(){
  //private_f();
  f();
  add(5);
  add(6);
  printf("%f", E);
  return 0;
}
```