KU LEUVEN

C: an introduction

preprocessor

C Preprocessor

- The preprocessor is a text substitution tool that modifies the source code before it is compiled

1. Inclusion of header files.
   ```
   #include <stdio.h>
   #include "allconstants.h"
   ```
2. Macro expansion.
   define macros, the C preprocessor will replace the macros with their definitions throughout the program.
   ```
   #define MAXITERATIONS 10000
   ```
3. Conditional compilation.
   Include or exclude parts of the program according to various conditions.

Preprocessor: essentials

- Preprocessing encompasses all tasks that logically precede the compilation of a program.
- The preprocessor is controlled by special command-lines, beginning with the hash symbol **#**.
- The preprocessor handles the logic behind all the # directives in C.
  It runs in a single pass, and essentially is just a substitution engine.
- Preprocessor commands have file-scope. They are visible from the point at which they are defined until the end of the source file.

Preprocessor commands

Directive	Description
#include	Inserts a particular header from another file.
#define	Substitutes a preprocessor macro.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.

https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm

C Preprocessor in Action

- `gcc -E program.c`

  puts program.c only through the preprocessor and sends the results to standard output
- Reroute output via

  ```
  gcc -E program.c > program
  ```

  Or

  ```
  gcc -E program.c | less
  ```


Inclusion of header files

Inclusion of Header Files

Typical *includes*

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

- The `#include` directives "paste" the contents of the files `stdio.h`, `stdlib.h` into the source code, at the very place where the directives appear.
- These files contain information about some library functions used in the program
- "*filename*" is searched for in the directory of the source code, and the search is continued afterwards using the same search scheme as used for *<filename>*
- search strategy is implementation dependent

#include examples

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

Looks in `/usr/include` for `stdio.h` inserts contents into source file

```
#include "defs.h"
```

Looks for `defs.h` in local directory and inserts contents into source file

files insert

create own .h files

- Anything can be put into a header file.
  - Good programming practice allows only definitions and function prototypes and preprocessor commands
- large programs
  - collect all related functions in a .c file
  - write a .h file containing all the prototypes of the functions
  - #include header file in the files using the functions
- small programs -> sequence:
  - prototypes
  - main() function
  - functions

Macro processing

Macro Processing

- When does it occur?
  - Before compilation
- What is it?
  - Text substitution
  - *substituting text: can be continued on a next line in code file using* \ at the end of the line
- Good practice: symbolic constants are usually given UPPERCASE names to distinguish them from variables and functions.

Macro Processing: why?

- To save time: define a macro for long sequences that need to be repeated many times.
- To make the software easy to change/maintain: changing the macro definition, automatically updates the entire software.

Symbolic Constants

```
#define BUFFERSIZE  256
#define MIN_VALUE   -32
#define PI          3.14159
```

- Syntax is quite different from regular C.
  - **No ;**
  - **No =**
- Note, a name defined by **`#define`** can be undefined using the command **`#undef`**. The name may then be redefined to represent a different replacement text.
- Good practice: symbolic constants are usually given UPPERCASE names to distinguish them from variables and functions.

Symbolic Constants

- *Extremely bad* practice to have “magic numbers” in code. It may be difficult to see what the number stands for, and code changes become error-prone.
- Use **`#define`** to define named constants, all in the one place

```
#define ARRAY_LENGTH     2500
#define BLOCK_SIZE       4096
#define TRACK_SIZE       16*BLOCK_SIZE
#define STRING     "Hello World!\n"
```

- Symbolic constants mean making changes of constants is easy and safe.

Symbolic Constants: Good Style

- By convention, constant terms are given **`UPPERCASE`** names. This distinguishes them from variables and functions, which usually begin with a lowercase letter.
- Variables qualified by **`const`** are often given an Uppercase first letter.

```
const double Threshold = 5.4;
```


Macro Processing

- *name* is not substituted
  - between quotes
  - part of another name

```
#define ETERNITY for (;;)
#define STRING “This is a STRING“
#define BEGIN {
#define END }
```

```c
/*
preproc_01.c
show define and undef
*/
#include<stdio.h>

#define NUMBER 15

int main(void){

printf("NUMBER has the value %d \n", NUMBER);

#undef NUMBER
#define NUMBER  33

printf("NUMBER has the value %d \n", NUMBER);

return 0;
}

```

```
frankvp@CRD-L-08004:.../preprocessor$ gcc preproc_01.c -o preproc_01
frankvp@CRD-L-08004:.../preprocessor$ ./preproc_01
NUMBER has the value 15
NUMBER has the value 33
frankvp@CRD-L-08004:.../preprocessor$
```



```c
#include <stdio.h>
// preproc_02.c
#define BUFSIZE     5
#define NELEMS(a)   (sizeof(a) / sizeof(a[0]))
#define MIN(x,y)    ((x)<(y) ? (x) : (y))
#define SQR(x)      ((x)*(x))

int main(void)
{
    int i;
    double array[BUFSIZE] = { 56.3, -981.3, 23 };
    double s = array[0];

    for(i=1; i<NELEMS(array); ++i)
        s = MIN(s, array[i]);

    printf("s = %f\t SQR(s+5) = %f\n", s, SQR(s+5));

    return 0;
}
```

```
frankvp@CRD-L-08004:.../preprocessor$ gcc preproc_02.c -o preproc_02
frankvp@CRD-L-08004:.../preprocessor$ ./preproc_02
s = -981.300000  SQR(s+5) = 953161.690000
frankvp@CRD-L-08004:.../preprocessor$
```

Macro Processing

- a single macro which will work for different types

```
#define max(a, b) a >= b ? a : b
int x = 7;
int y = 8;
float p = 78.6;
float q = 29.2;
printf("%d %f", max(x, y), max(p,q));
```

- Speed:
  - Macros can perform function-like operations without the overhead of a function call.
  - Macro code is expanded inline, while function calls require various extra runtime operations.

Predefined macros

- The preprocessor defines a number of predefined macros.

__FILE__ - a string that holds the path/name of the compiled file;

__LINE__ - an integer that holds the number of the current line number;

__DATE__ - a string that holds the current system date;

__TIME__ - a string that holds the current system time;

__STDC__ - defined as the value '1' if the compiler conforms with the ANSI C standard;

```c
/*
preproc_05.c
predefined macros
taken from: https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm
*/

#include <stdio.h>

int main(void) {

   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );

return 0;
}
```

```
frankvp@CRD-L-08004:.../preprocessor$ gcc preproc_05.c -o preproc_05
frankvp@CRD-L-08004:.../preprocessor$ ./preproc_05
File :preproc_05.c
Date :Jan 16 2021
Time :17:38:36
Line :14
frankvp@CRD-L-08004:.../preprocessor$
```


Watch out

```
#define SQUARE (X) (X * X)

int z = SQUARE(2);
```

becomes

```
int z = (X) (X * X) (2);
```

```
#define SQUARE(X) (X * X)

int z = SQUARE(2);
int z = SQUARE(x + y);
```

Fix:

```
#define SQUARE(X) ((X) * (X))
```

Parentheses are your friend!

File: preproc_06.c

What happens?

```
/* Macro */

SQUARE(x)
.
.
.
SQUARE(x)
.
.
.
SQUARE(x)
```

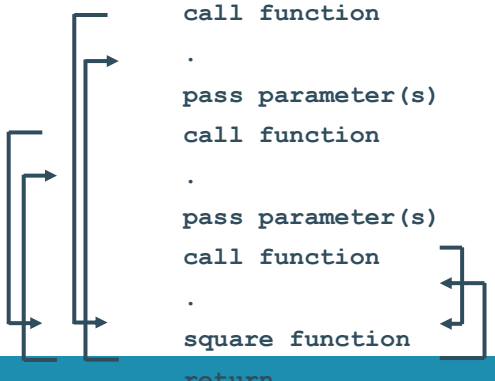
```
/* Function */

square(x)
.
.
.
square(x)
.
.
.
square(x)
```


What happens?

```
/* Macro */

((x) * (x))
.
.
.
((x) * (x))
.
.
.
((x) * (x))
```



```
/* Function */

pass parameter(s)
call function
.
pass parameter(s)
call function
.
pass parameter(s)
call function
.
square function
return
```

Macros vs. Functions

- Macros
  - Text substitution at Translation (compile) time
  - May have problems: e.g. square(x++)
  - Will work with different types due to operator overloading
    - floats, doubles, ints, …
  - Difficult to implement if complex
  - Macro optimizes for speed. (nowadays of less importance)
- Functions
  - Separate piece of code
  - Overhead of passing arguments and returning results via stack
  - Fixes ambiguity problems: e.g. square(x + y) or(x++)
  - Function optimizes for space.

Conditional compiling

Conditional compiling

- Why?
  - Experiment with code
    - Add additional code
  - Develop portable code
  - Protect header-files from multiple inclusion.

Conditional compiling

- The conditional directives are:
  - `#ifdef` - If this macro is defined
  - `#ifndef` - If this macro is not defined
  - `#if` - Test if a compile time condition is true
  - `#else` - The alternative for #if
  - `#elif` - #else an #if in one statement
  - `#endif` - End preprocessor conditional

Conditional compiling

```
#ifdef DEBUG
    printf("Some debug info here");
#endif

#define DEBUG
      or
gcc –DDEBUG
```



```c
/*
cond_compil_1.c
example conditional compiling

compile with gcc -DDEBUG
or define DEBUG
*/

#include<stdio.h>


// # define DEBUG

int main()
{
#ifdef DEBUG
    printf("Some debug info part 1 here \n");
#endif

    printf("Hello World \n");

#ifdef DEBUG
    printf("Some debug info part 2 here \n");
#else
    printf("DEBUG macro not defined\n");
#endif

return 0;
}
```

```
frankvp@CRD-L-08004:.../preprocessor$ gcc -DDEBUG cond_compil_1.c -o cond_compil_1
frankvp@CRD-L-08004:.../preprocessor$ ./cond_compil_1
Some debug info part 1 here
Hello World
Some debug info part 2 here
frankvp@CRD-L-08004:.../preprocessor$ gcc cond_compil_1.c -o cond_compil_1
frankvp@CRD-L-08004:.../preprocessor$ ./cond_compil_1
Hello World
DEBUG macro not defined
frankvp@CRD-L-08004:.../preprocessor$
```

Conditional compiling

- Portable code: depending on the conditions tested, some parts of the code can be executed or not

```
#ifdef WIN_32
// Win32 code
#else
// Win64 code
#endif
```


Conditional compiling

- `#if`-instruction tests a constant integer expression
- preprocessor evaluates the expression
  - not zero is true
  - zero is false
- same mechanism as with other if-else construction
  - #else and #elif alternatives
  - #endif end of the construction

Header guard

- A header file should be included in a given source file at most once. Often header files themselves include other header files.
- Multiple inclusions in a source file may mean multiple definitions of certain symbols, and hence, compilation errors

```
#ifndef A_HEADER_H_
    #define A_HEADER_H_
    /* Contents of header file is contained here. */
    #endif
```

  - contents of a_header.h is only taken and defined if A_HEADER_H_ was not yet defined