

CH-230-A

Programming in C and C++

C/C++

Lecture 6

Dr. Kinga Lipskoch

Fall 2020

The C Preprocessor (1)

- ▶ Before compilation, C source files are being preprocessed
- ▶ The preprocessor replaces tokens by an arbitrary number of characters
- ▶ Offers possibility of:
 - ▶ Use of named constants
 - ▶ Include files
 - ▶ Conditional compilation
 - ▶ Use of macros with arguments

The C Preprocessor (2)

- ▶ The preprocessor has a different syntax from C
- ▶ All preprocessor commands start with #
- ▶ A preprocessor directive terminates at the end-of-line
 - ▶ Do not put ; at the end of a directive
- ▶ It is a common programming practice to use all uppercase letters for macro names

The C Preprocessor: File Inclusion

- ▶ `#include <filename>`
 - ▶ includes file, follows implementation defined rule where to look for file, for Unix is typically `/usr/include`
 - ▶ Ex: `#include <stdio.h>`
- ▶ `#include "filename"`
 - ▶ looks in the directory of the source file
 - ▶ Ex: `#include "myheader.h"`
- ▶ Included files may include further files
- ▶ Typically used to include prototype declarations

The C Preprocessor: Motivation for Macros (1)

- ▶ Motivation for using named constants/macros
- ▶ What if the size of arrays has to be changed?

```
1 int data[10];
2 int twice[10];
3 int main()
4 {
5     int index;
6     for(index = 0; index < 10; ++index) {
7         data[index] = index;
8         twice[index] = index * 2;
9     }
10    return 0;
11 }
```

The C Preprocessor: Motivation for Macros (2)

More generic program if using named constants/macros

```
1 #define SIZE 20
2 int data[SIZE];
3 int twice[SIZE];
4 int main()
5 {
6     int index;
7     for(index = 0; index < SIZE; ++index) {
8         data[index] = index;
9         twice[index] = index * 2;
10    }
11    return 0;
12 }
```

Works but it no type information is associated with macros, so using `const` for this problem is a better solution.

The C Preprocessor: Macro Substitution (1)

- ▶ Definition of macro
 - ▶ `#define NAME replacement_text`
- ▶ Any name may be replaced with any replacement text
 - ▶ Ex: `#define FOREVER for (;;) defines new word FOREVER to be an infinite loop`
 - ▶ Ex: `#define ODD(A, B) { unsigned char abit=A & 1; \ unsigned char bbit=B & 1; \ ... }`

The C Preprocessor: Macro Substitution (2)

- ▶ Possible to define macros with arguments
 - ▶ `#define MAX(A, B) ((A) > (B) ? (A) : (B))`
- ▶ Each formal parameter (A or B) will be replaced by corresponding argument
 - ▶ `x = MAX(p+q, r+s);` will be replaced by
 - ▶ `x = ((p+q) > (r+s) ? (p+q) : (r+s));`
- ▶ It is type independent

The C Preprocessor: Macro Substitution (3)

- ▶ Why are the () around the variables important in the macro definition?
 - ▶ `#define SQR(A) (A)*(A)`
- ▶ Write a small program using this and see the effect without () in (A)*(A) by calling `SQR(5+1)`
- ▶ Try also `gcc -E program.c` sends the output of the preprocessor to the standard output
- ▶ What happens if you call `SQR(++i)`?

The C Preprocessor: Macro Substitution (4)

- ▶ Spacing and syntax in macro definition is very important
- ▶ See the preprocessor output of the following source code

```
1 #include <stdio.h>
2 #define MAX =10
3 int main()
4 {
5     int counter;
6     for(counter =MAX; counter > 0; --counter)
7         printf("Hi there!\n");
8     return 0;
9 }
```

wrong_macro.c

The C Preprocessor: Macro Substitution (5)

- ▶ Defined names can be undefined using
 - ▶ `#undef NAME`
- ▶ Formal parameters are not replaced within quoted strings
- ▶ If parameter name is preceded by `#` in replacement text, the actual argument will be put inside quotes
 - ▶ `#define DPRINT(expr) printf(#expr " = %g\n", expr)`
 - ▶ `DPRINT(x/y)` will be expanded to
 - ▶ `printf("x/y" " = %g\n", x/y);`

The C Preprocessor: Conditional Inclusion (1)

- ▶ Preprocessing can be controlled by using conditional statements which will be evaluated while preprocessor runs
- ▶ Enables programmer to selectively include code, depending on conditions
- ▶ `#if`, `#endif`, `#elif` (i.e., else if), `#else`

```
1 #if defined(DEBUG)    // short: #ifdef DEBUG
2     printf("x: %d\n", x);
3 #endif
```

The C Preprocessor: Conditional Inclusion (2)

- ▶ `#ifdef`, `#ifndef` are special constructs that test whether name is (not) defined
- ▶ `gcc` allows to define names using the `-D` switch
- ▶ Ex: `gcc -DDEBUG -c program.c`
- ▶ Previous line is equivalent to
`#define DEBUG`

The C Preprocessor: Conditional Inclusion (3)

- ▶ Write a small program in which you illustrate the use of conditional inclusion for debugging purposes
- ▶ **Ex:** If the name `DEBUG` is defined then print on the screen the message "This is a test version of the program"
- ▶ If `DEBUG` is not defined then print on the screen the message "This is the production version of the program"
- ▶ Also experiment with `gcc -D`

The Structure of a Header File with Conditional Inclusion

```
1  /* Student.h */
2  #ifndef _LIST_H
3  #define _LIST_H
4  struct list {
5      int info;
6      struct list *next;
7  };
8  void printList(struct list *);
9  struct list * push_front(struct list *, int);
10 ...
11 #endif // this matches the initial #ifndef
```

Bit Operations

- ▶ The bit is the smallest unit of information
 - ▶ Represented by 0 or 1
- ▶ Eight bits form one byte
 - ▶ Which data type could be used for representation?
- ▶ Low-level coding like writing device drivers or graphic programming require bit operations
- ▶ Data representation
 - ▶ Octal (format %o), hexadecimal (format %x, representation prefix 0x)
- ▶ In C you can manipulate individual bits within a variable

Bitwise Operators (1)

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal	128	64	32	16	8	4	2	1
Binary number	0	1	0	1	1	1	0	1

- ▶ Allow you to store and manipulate multiple states in one variable
- ▶ Allows to set and test individual bits in a variable

Bitwise Operators (2)

Operator	Function	Use
~	bitwise NOT	~expr
<<	left shift	expr1 << expr2
>>	right shift	expr1 >> expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
	bitwise OR	expr1 expr2
&=	bitwise AND assign	expr1 &= expr2
^=	bitwise XOR assign	expr1 ^= expr2
=	bitwise OR assign	expr1 = expr2

Bitwise and Logical AND

```
1 #include <stdio.h>
2 int main()
3 {
4     int i1, i2;
5     i1 = 6; // set to 4 and suddenly check 3 fails
6     i2 = 2;
7     if ((i1 != 0) && (i2 != 0))
8         printf("1: Both are not zero!\n");
9     if (i1 && i2)
10        printf("2: Both are not zero!\n");
11    // wrong check
12    if (i1 & i2)
13        printf("3: Both are not zero!\n");
14    return 0;
15 }
```

The Left-Shift Operator

- ▶ Moves the data to the left a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are 0's
- ▶ Ex: `10101101 << 3` results in `01101000`

The Right-Shift Operator

- ▶ Moves the data to the right a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are:
 - ▶ 0's if variable is unsigned
 - ▶ Value of the sign bit if variable is signed
- ▶ Ex:
 - ▶ $7 = 00000111 \gg 2$ results in 00000001
 - ▶ $-7 = 11111001 \gg 2$ results in 11111110