ALX LESSON 0x0D C - Preprocessor

C - Programming

TABLE OF CONTENTS

01

OVERVIEW topics

Topics

C Programming

Topics

What are macros and how to use them

What are the most common predefined macros

How to include guard your header files

Slides On Telegram

https://t.me/alx_2023

C Programming

Topics

@ALX_2023

02

Learning Objectives

What are macros and how to use them

An object-like macro is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it.

You create macros with the ‘#define’ directive.

‘#define’ is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's body

```
#define BUFFER_SIZE 1024
```

defines a macro named BUFFER_SIZE as an abbreviation for the token 1024.

What are macros and how to use them

```
#define BUFFER_SIZE 1024
```

defines a macro named BUFFER_SIZE as an abbreviation for the token 1024.

If somewhere after this '#define' directive there comes a C statement of the form

```
foo = (char *) malloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and expand the macro BUFFER_SIZE. The C compiler will see the same tokens as it would if you had written

```
foo = (char *) malloc (1024);
```

The macro's body ends at the end of the '#define' line. You may continue the definition onto multiple lines, if necessary, using backslash-newline.

```
#define NUMBERS 1, \
                2, \
                3

int x[] = { NUMBERS };
        ==> int x[] = { 1, 2, 3 };
```

macros

```c
#include <stdio.h>

#define NUMBERS 1, \
                2, \
                3

int main() {
    int x[] = { NUMBERS };

    // Print out the elements of the array
    for (int i = 0; i < sizeof(x)/sizeof(int); i++) {
        printf("%d ", x[i]);
    }
    printf("\n");

    return 0;
}
```

What are macros and how to use them

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

macros

```
#include <stdio.h>

#define SUMNUM(X) X+X

int main() {
    int n = 0;
    n =  SUMNUM(5) * SUMNUM(2);
    printf("%d\n",n);
    return 0;
}
```

What are macros and how to use them

When the preprocessor expands a macro name, the macro's expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```
#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
     ==> BUFSIZE
     ==> 1024
```

TABLESIZE is expanded first to produce BUFSIZE, then that macro is expanded to produce the final result, 1024.

macros

```c
#include <stdio.h>

#define TABLESIZE BUFSIZE
#define BUFSIZE 1024

int main() {
    printf("TABLESIZE: %d\n", TABLESIZE);
    return 0;
}
```

What are macros and how to use them

This makes a difference if you change the definition of BUFSIZE at some point in the source file. TABLESIZE, defined as shown, will always expand using the definition of BUFSIZE that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now TABLESIZE expands (in two stages) to 37.

macros

```c
#include <stdio.h>

#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37

int main() {
    printf("%d\n", TABLESIZE);  // should print 37
    return 0;
}
```

What are macros and how to use them

The C preprocessor scans your program sequentially. Macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces

```
foo = X;
bar = 4;
```

macros

```
1  #include <stdio.h>
2
3  int main(void) {
4      int foo, bar;
5      foo = X;
6      #define X 4
7      bar = X;
8      printf("foo = %d, bar = %d\n", foo, bar);
9      return 0;
10 }
```

```
In function 'main':
5    error: 'X' undeclared (first use in this function)
```

Macro Arguments

To invoke a macro that takes arguments, you write the name of the macro followed by a list of actual arguments in parentheses, separated by commas. The number of arguments you give must match the number of parameters in the macro definition. When the macro is expanded, each use of a parameter in its body is replaced by the tokens of the corresponding argument. (You need not use all of the parameters in the macro body.)

As an example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs, and some uses.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
  x = min(a, b);          ==>  x = ((a) < (b) ? (a) : (b));
  y = min(1, 2);          ==>  y = ((1) < (2) ? (1) : (2));
  z = min(a + 28, *p);    ==>  z = ((a + 28) < (*p) ? (a + 28) : (*p));
```

macros

```c
#include <stdio.h>

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

int main() {
    int a = 5, b = 10;
    int x = min(a, b);
    printf("x = %d\n", x);

    int y = min(1, 2);
    printf("y = %d\n", y);

    int p_value = 15;
    int *p = &p_value;
    int z = min(a + 28, *p);
    printf("z = %d\n", z);

    return 0;
}
```

Macro Arguments

You can leave macro arguments empty; this is not an error to the preprocessor (but many macros will then expand to invalid code). You cannot leave out arguments entirely; if a macro takes two arguments, there must be exactly one comma at the top level of its argument list. Here are some silly examples using min:

```
min(, b)        ==> ((   ) < (b) ? (   ) : (b))
min(a, )        ==> ((a  ) < ( ) ? (a  ) : ( ))
min(,)          ==> ((   ) < ( ) ? (   ) : ( ))
min((,),)       ==> (((,)) < ( ) ? ((,)) : ( ))

min()     error--> macro "min" requires 2 arguments, but only 1 given
min(,,)   error--> macro "min" passed 3 arguments, but takes just 2
```

Macro

You can leave macro arguments empty; this is not an error to the preprocessor (but many macros will then expand to invalid code). You cannot leave out arguments entirely; if a macro takes two arguments, there must be exactly one comma at the top level of its argument list. Here are some silly examples using min:

```
min(, b)        ==> ((   ) < (b) ? (   ) : (b))
min(a, )        ==> ((a  ) < ( ) ? (a  ) : ( ))
min(,)          ==> ((   ) < ( ) ? (   ) : ( ))
min((,),)       ==> (((,)) < ( ) ? ((,)) : ( ))

min()      error--> macro "min" requires 2 arguments, but only 1 given
min(,,)    error--> macro "min" passed 3 arguments, but takes just 2
```

Pre Processor Directives

#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	Uses the value of Macro

Pre Processor Directives

#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#include	Inserts a particular header from another file
#error	Prints error message on stderr and halts compilation
#pragma	Issues special commands to the compiler

#include

```c
#include <stdio.h>


int main() {
    printf("Hello, world!\n");
    return 0;
}
```

#define:

```
#include <stdio.h>

#define PI 3.14159
#define AREA(radius) (PI * (radius) * (radius))

int main() {
    float radius = 5.0;
    printf("The area of the circle with radius %f
is %f\n", radius, AREA(radius));
    return 0;
}
```

#ifdef, #ifndef, #else and #endif

```c
#include <stdio.h>

#define DEBUG

int main() {
#ifdef DEBUG
    printf("Debug mode is on.\n");
#else
    printf("Debug mode is off.\n");
#endif

#ifndef PI
#define PI 3.14159
#endif

    float radius = 5.0;
    printf("The area of the circle with radius %f is %f\n", radius, PI * (radius) * (radius));
    return 0;
}
```

#ifdef, #ifndef, #else and #endif

```c
#include <stdio.h>

#ifdef DEBUG
#error Debug mode is not allowed in this program!
#endif

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

#pragma

```c
#include <stdio.h>

//we use the #pragma directive to instruct the GCC
compiler to optimize the code with level 3
optimization.
#pragma GCC optimize("O3")

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Standard Predefined Macros

The standard predefined macros are specified by the relevant language standards, so they are available with all compilers that implement those standards. Older compilers may not provide all of them. Their names all start with double underscores.

__FILE__

This macro expands to the name of the current input file, in the form of a C string constant. This is the path by which the preprocessor opened the file, not the short name specified in ‘#include’ or as the input file name argument. For example, "/usr/local/include/myheader.h" is a possible expansion of this macro.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its “definition” changes with each new line of source code.

Standard Predefined Macros

__DATE__

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Feb 12 1996". If the day of the month is less than 10, it is padded with a space on the left.

__TIME__

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like "23:59:01".

__STDC__

In normal operation, this macro expands to the constant 1, to signify that this compiler conforms to ISO Standard C. If GNU CPP is used with a compiler other than GCC, this is not necessarily true; however, the preprocessor always conforms to the standard unless the -traditional-cpp option is used.

Pre defiened macros

```c
#include <stdio.h>

int main() {
    printf("Current file name: %s\n", __FILE__);
    printf("Current line number: %d\n", __LINE__);
    printf("Current date: %s\n", __DATE__);
    printf("Current time: %s\n", __TIME__);
    printf("Standard compliance: %d\n", __STDC__);
    return 0;
}
```

#include guard

The following C code demonstrates a real problem that can arise if #include guards are missing:

File "grandparent.h" [edit]

```c
struct foo {
    int member;
};
```

File "parent.h" [edit]

```c
#include "grandparent.h"
```

File "child.c" [edit]

```c
#include "grandparent.h"
#include "parent.h"
```

#include guard

Result [edit]

```
struct foo {
    int member;
};
struct foo {
    int member;
};
```

Here, the file "child.c" has indirectly included two copies of the text in the header file "grandparent.h". This causes a compilation error, since the structure type foo will thus be defined twice. In C++, this would be called a violation of the one definition rule.

#include guard

the same code is used with the addition of #include guards. The C preprocessor preprocesses the header files, including and further preprocessing them recursively. This will result in a correct source file, as we will see.

File "grandparent.h" [edit]

```
#ifndef GRANDPARENT_H
#define GRANDPARENT_H

struct foo {
    int member;
};

#endif /* GRANDPARENT_H */
```

File "parent.h" [edit]

```
#include "grandparent.h"
```

#include guard

File "child.c" [edit]

```c
#include "grandparent.h"
#include "parent.h"
```

#include guard

Result [edit]

```
struct foo {
    int member;
};
```

Here, the first inclusion of "grandparent.h" has the macro GRANDPARENT_H defined. When "child.c" includes "grandparent.h" at the second time (while including "parent.h"), as the #ifndef test returns false, the preprocessor skips down to the #endif, thus avoiding the second definition of struct foo. The program compiles correctly.

04

Hands on lab Practice


Have a Question
Leave a Comment!

Subscribe

To stay updated with latest videos

Share

To let the others know more

Thanks