

C preprocessor

PC FAF 2019

The **C preprocessor** executes *before* a program is compiled. Some actions it performs are:

- the inclusion of other files into the file being compiled,
- definition of **symbolic constants** and **macros**,
- **conditional compilation** of program code and
- **conditional execution of preprocessor directives**.

Preprocessor directives begin with `#`, and only whitespace characters and comments delimited by `/*` and `*/` may appear before a preprocessor directive on a line.

#include

preprocessor directive

The **#include preprocessor directive** has been used throughout this course. It causes a *copy* of a specified file to be included in place of the directive. The two forms of the #include directive are:

```
#include <numefisier>
```

```
#include "numefisier"
```

The difference between these is the location at which the preprocessor begins searches for the file to be included. If the filename is enclosed in angle brackets (< and >)—used for **standard library headers**—the search is performed in an *implementation-dependent* manner, normally through predesignated compiler and system directories.

If the filename is enclosed in *quotes*, the preprocessor starts searches in the *same* directory as the file being compiled for the file to be included. This method is normally used to include programmer-defined headers.

How to create header files

Step 1 : Make header file
(write functions code)

Step 2 : Save Code with [.h] Extension .
(let the name is **myLib.h**)

Step 3 : Write the main program code.
In the headers section add
#include "myLib.h"

Note : be sure that both files are in the
same directory!




Demo

```
[*] directives_001.c  mystring.h
1  int mystrlen(char *x)
2  {
3      int i = 0;
4      while (x[i] != '\0') i++;
5      return i;
6  }
7
8  void mystrconcat(char *x, char *y)
9  {
10     int i = 0, j = 0;
11     while (x[i] != '\0') i++;
12     while (y[j] != '\0') { x[i] = y[j]; i++; j++; }
13     x[i] = '\0';
14 }
15
16 long mystoi(char *x)
17 {
18     long n = 0;
19     int i = 0;
20     while (x[i] != '\0') { n = n * 10 + (x[i] - '0'); i++; }
21     return n;
22 }
23
24 int mystrcomp (char *x, char *y)
25 {
26     int i = 0;
27     while ( x[i] == y[i] && x[i] != '\0' && y[i] != '\0') i++;
28     if (x[i] == '\0' && y[i] == '\0') return 0;
29     else if (x[i] == '\0' && y[i] != '\0') return -1 ;
30     else if (x[i] != '\0' && y[i] == '\0') return 1;
31     else if (x[i] > y[i] ) return 1; else return -1;
32 }
```



```
1 #include <stdio.h>
2 #include "mystring.h"
3
4 int main()
5 {
6     char a[100], b[100];
7     long n;
8     scanf("%s", &a);
9     scanf("%s", &b);
10    printf("%d", mystrcomp(a, b));
11    // mystrconcat(a, b);
12    // printf("%s", a);
13    // n = mystoi(a);
14    // printf("%ld", n);
15 }
```

d Files

Name	Date modified	Type
 directives_001.c	01.12.2019 15:08	C Source File
 directives_001.exe	01.12.2019 15:08	Application
 mystring.h	01.12.2019 15:06	C Header Fi...

#define directive

How

#define

works

The **#define directive** creates *symbolic constants*—constants represented as symbols—and **macros**—operations defined as symbols. The #define directive format is

#define identifier replacement-text

When this line appears in a file, all subsequent occurrences of *identifier* that do *not* appear in string literals or comments will be replaced by *replacement text* automatically *before* the program is compiled. For example,

#define PI 3.14159

replaces all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159. Symbolic constants enable you to create a name for a constant and use the name throughout the program.



Error-Prevention Tip

Everything to the right of the symbolic constant name replaces the symbolic constant. For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of the identifier `PI` with `= 3.14159`. This is the cause of many subtle logic and syntax errors. For this reason, you may prefer to use `const` variable declarations, such as `const double PI = 3.14159`; in preference to the preceding `#define`.



Common Programming Error

Attempting to redefine a symbolic constant with a new value is an error.



Software Engineering Observation

Using symbolic constants makes programs easier to modify. Rather than search for every occurrence of a value in your code, you modify a symbolic constant once in its `#define` directive. When the program is recompiled, all occurrences of that constant in the program are modified accordingly.



Good Programming Practice

By convention, symbolic constants are defined using only uppercase letters and underscores.



Good Programming Practice

By convention, symbolic constants are defined using only uppercase letters and underscores.

#define

Preprocessor Directive:

Macros

A macro is an identifier defined in a `#define` preprocessor directive. As with symbolic constants, the **macro-identifier** is replaced with **replacement-text** before the program is compiled.

Macros may be defined with or without **arguments**. A macro without arguments is processed like a symbolic constant. In a **macro with arguments**, the arguments are substituted in the replacement text, then the macro is **expanded**—the replacement-text replaces the identifier and argument list in the program. A symbolic constant is a type of macro.

Macro with One Argument

Consider the following one-argument *macro definition* that calculates the area of a circle:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

Expanding a Macro with an Argument

Wherever CIRCLE_AREA(y) appears in the file, the value of y is substituted for x in the replacement-text, the symbolic constant PI is replaced by its value (defined previously) and the macro is expanded in the program. For example, the statement

```
area = CIRCLE_AREA(4);
```

is expanded to

```
area = ((3.14159) * (4) * (4));
```

then, at compile time, the value of the expression is evaluated and assigned to variable area.

The role of parentheses

The *parentheses* around each x in the replacement-text force the proper order of evaluation when the macro argument is an expression. For example, the statement

```
area = CIRCLE_AREA(c + 2);
```

is expanded to

```
area = ((3.14159) * (c + 2) * (c + 2));
```

which evaluates *correctly* because the parentheses force the proper order of evaluation. If the parentheses in the macro definition are omitted, the macro expansion is

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates *incorrectly* as

```
area = (3.14159 * c) + (2 * c) + 2;
```

because of the rules of operator precedence.



Error-Prevention Tip

Enclose macro arguments in parentheses in the replacement-text to prevent logic errors.

Example

```
directives_002.c  mystring.h
1  #include <stdio.h>
2
3  #define PI 3.14159
4
5  #define CIRCLE_AREA(x) (PI * (x) * (x))
6
7  int main()
8  {
9      float circlearea, a;
10
11      scanf("%f", &a);
12      circlearea = CIRCLE_AREA(a);
13      printf("%f", circlearea);
14
15      return 0;
16 }
```

```
C:\Users\CTI UST\Desktop\directives_002.exe
4
50.265442
-----
Process exited after 3.084 seconds with return value 0
Press any key to continue . . .
```

Example 2

Change
measurement
units

```
directives_003.c  mystring.h
1  #include <stdio.h>
2
3  #define PI 3.14159
4
5  #define RADIANS(x) (PI * (x) / 180)
6
7  int main()
8  {
9      float unghi_radiani, unghi_grade;
10
11      scanf("%f", &unghi_grade);
12      unghi_radiani = RADIANS(unghi_grade);
13      printf("%f", unghi_radiani);
14
15      return 0;
16 }
```

```
C:\Users\CTI UST\Desktop\di...
30
0.523598
-----
Process exited after 3.183 seconds with return value 0
Press any key to continue . . .
```

Two argument macros

The following two-argument macro calculates the area of a rectangle:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever RECTANGLE_AREA(x, y) appears in the program, the values of x and y are substituted

in the macro replacement-text and the macro is expanded in place of the macro

name. For example, the statement

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

is expanded to

```
rectArea = ((a + 4) * (b + 7));
```

The value of the expression is evaluated at runtime and assigned to variable rectArea.

4 variables

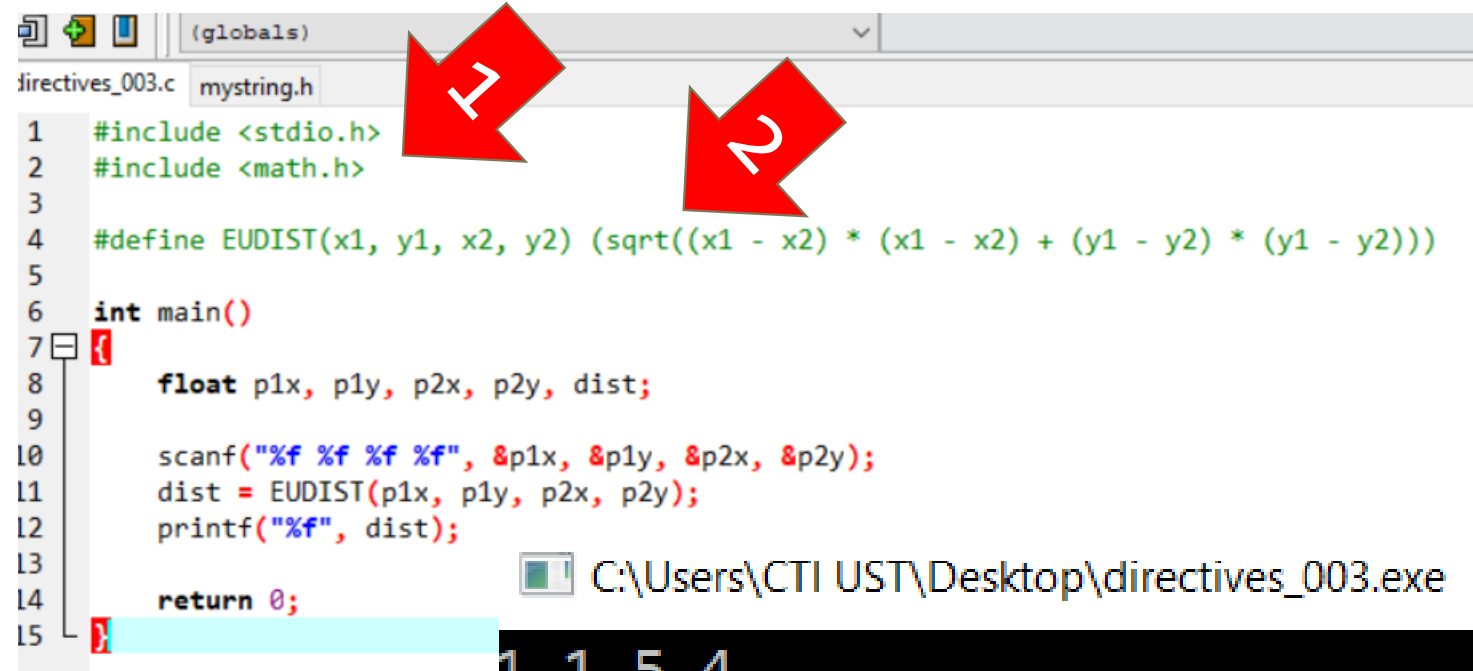
#define

directive

```
directives_003.c  mystring.h
1  #include <stdio.h>
2
3
4  #define MHTDIST(x1, y1, x2, y2) ( x1>x2 ? x1 - x2 : x2 - x1) + (y1 > y2 ? y1 - y2 : y2 - y1)
5
6  int main()
7  {
8      int p1x, p1y, p2x, p2y, dist;
9
10     scanf("%d %d %d %d", &p1x, &p1y, &p2x, &p2y);
11     dist = MHTDIST(p1x, p1y, p2x, p2y);
12     printf("%d", dist);
13
14     return 0;
15 }
```

```
C:\Users\CTI UST\Desktop\dire...
1 1 5 3
6
-----
Process exited after 4.462 seconds
with return value 0
Press any key to continue . . .
```

#define & #include example



```
(globals)
directives_003.c mystring.h
1 #include <stdio.h>
2 #include <math.h>
3
4 #define EUDIST(x1, y1, x2, y2) (sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)))
5
6 int main()
7 {
8     float p1x, p1y, p2x, p2y, dist;
9
10    scanf("%f %f %f %f", &p1x, &p1y, &p2x, &p2y);
11    dist = EUDIST(p1x, p1y, p2x, p2y);
12    printf("%f", dist);
13
14    return 0;
15 }
```

C:\Users\CTI UST\Desktop\directives_003.exe

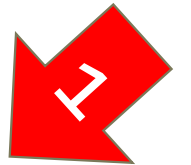
```
1 1 5 4
5.000000
-----
Process exited after 6.906 seconds
Press any key to continue . . .
```

Macro continuation character

\

The replacement-text for a macro or symbolic constant is normally any text on the line after the identifier in the `#define` directive. If the replacement-text for a macro or symbolic constant is longer than the remainder of the line, a backslash (`\`) continuation character must be placed at the end of the line, indicating that the replacement-text continues on the next line.

```
4  #define EUDIST(x1, y1, x2, y2) (sqrt((x1 - x2) \
5      | * (x1 - x2) + (y1 - y2) * (y1 - y2)))
6
```



Conditional compilation

Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code. Each conditional preprocessor directive evaluates a constant integer expression.

Cast expressions, sizeof expressions and enumeration constants *cannot* be evaluated in preprocessor directives.

#if...#endif

Preprocessor Directive

The conditional preprocessor construct is much like the if selection statement. Consider the following preprocessor code:

```
#if !defined(MY_CONSTANT)
    #define MY_CONSTANT 0
#endif
```

which determines whether MY_CONSTANT is *defined*—that is, whether MY_CONSTANT has already appeared in an earlier `#define` directive. The expression `defined(MY_CONSTANT)` evaluates to 1 if MY_CONSTANT is defined and 0 otherwise. If the result is 0, `!defined(MY_CONSTANT)` evaluates to 1 and MY_CONSTANT is defined. Otherwise, the `#define` directive is skipped.

Every `#if` construct ends with `#endif`.

Directives `#ifdef` and `#ifndef` are shorthand for `#if defined(name)` and `#if !defined(name)`.

Comment blocks

During program development, it's often helpful to "comment out" portions of code to prevent them from being compiled. If the code contains multiline comments, /* and */ cannot be used to accomplish this task, because such comments cannot be nested. Instead, you can use the following preprocessor construct:

```
#if 0
```

```
code prevented from compiling
```

```
#endif
```

To enable the code to be compiled, replace the 0 in the preceding construct with 1.

Debugging code

Conditional compilation is sometimes used as a *debugging* aid. Debuggers provide much more powerful features than conditional compilation, but if a debugger is not available, printf statements can be used to print variable values and to confirm the flow of control.

These printf statements can be enclosed in conditional preprocessor directives so the statements are compiled only while the debugging process is *not* completed. For example,

- **#ifdef DEBUG**
- **printf("Variable x = %d\n", x);**
- **#endif**

compiles the printf statement if the symbolic constant DEBUG is defined (**#define DEBUG**) before **#ifdef DEBUG**. When debugging is completed, you remove or comment out the **#define** directive from the source file and the printf statements inserted for debugging purposes are ignored during compilation.


```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define f(x) x*x
5  #define DEBUG
6
7  int main()
8  {
9      int x, y = 10, sum;
10
11     for (x =1; x <= y; x++)
12     {
13         sum = sum + f(x);
14         #ifdef DEBUG
15             printf("%d %d\n", x, f(x));
16         #endif
17     }
18
19     return 0;
20 }

```

C:\Users\CTI UST\De

```

1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100

```

```

-----
Process exited
Press any key

```

```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define f(x) x*x
5  // #define DEBUG
6
7  int main()
8  {
9      int x, y = 10, sum;
10
11     for (x =1; x <= y; x++)
12     {
13         sum = sum + f(x);
14         #ifdef DEBUG
15             printf("%d %d\n", x, f(x));
16         #endif
17     }
18
19     return 0;
20 }

```

C:\Users\CTI UST\Desktop\directives_005.exe

```

-----
Process exited after 0.02209
Press any key to continue . .

```

#

and

##

The **#** operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition:

```
#define HELLO(x) puts("Hello, " #x);
```

when HELLO(John) appears in a program file, it's expanded to the string "John" replaces **#x** in the replacement-text. Strings separated by whitespace are concatenated during preprocessing, so the preceding statement is equivalent to the **#** operator must be used in a macro with arguments because the operand of **#** refers to an argument of the macro.

The **##** operator concatenates two tokens.

Consider the following macro definition:

```
#define TOKENCONCAT(x, y) x ## y
```

When TOKENCONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, TOKENCONCAT(O, K) is replaced by OK in the program. The **##** operator must have two operands.

Assert.h

The **assert** macro—defined in **<assert.h>**—tests the value of an expression at execution time. If the value is false (0), assert prints an error message and calls function **abort** (of the general utilities library—**<stdlib.h>**) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable *x* should never be larger than 10 in a program. An assertion may be used to test the value of *x* and print an error message if the value of *x* is greater than 10. The statement would be

```
assert(x <= 10);
```

If *x* is greater than 10 when the preceding statement executes, the program displays an error message containing the line number and filename where the assert statement appears, then *terminates*. You then concentrate on this area of the code to find the error.

Debugging

Definition

What is Debugger?

In general, debugger is utility that runs target program in controlled environment where you can control execution of program and see the state of program when program is paused.

Use online compilers with debugger: GDB is such debugger, which is used to debug C/C++ programs.

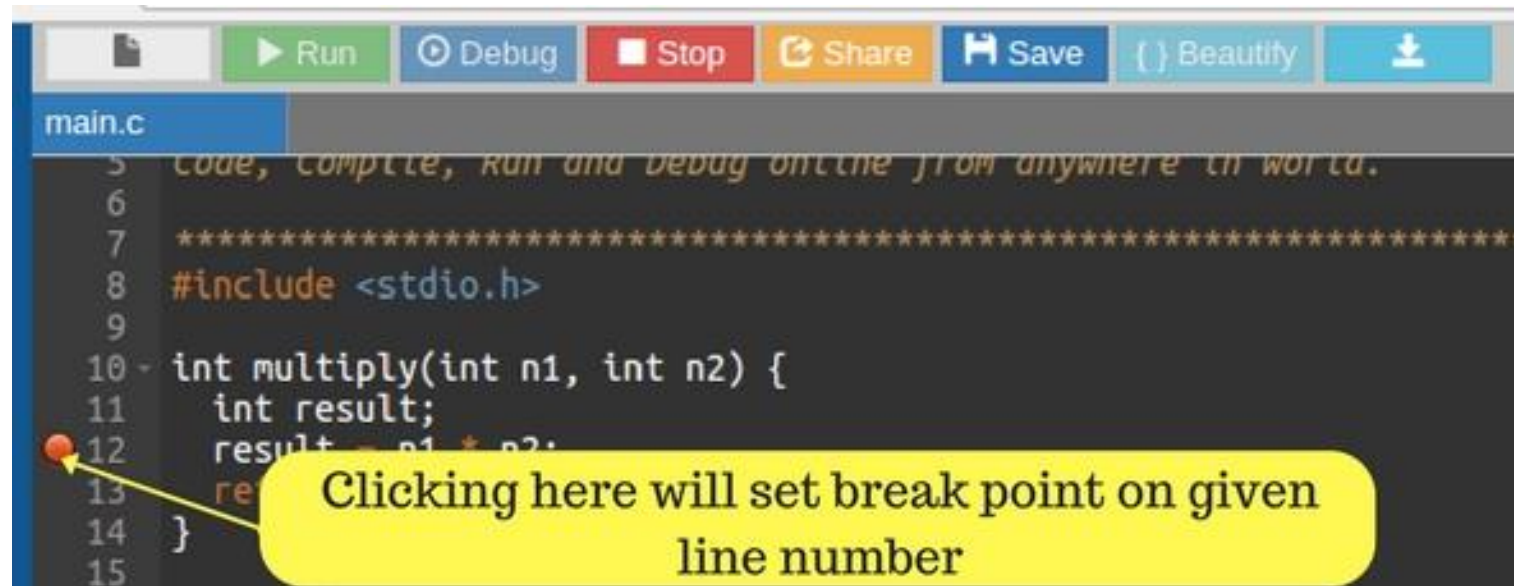
OnlineGDB provides an interface to use GDB in cloud environment from your browser.

How to use

How can I control execution of program?

We can tell debugger when to pause a program by setting breakpoints.

To set a breakpoint, click on blank area seen on left side of line number in editor. When you click it, it should display red circle; which means breakpoint is set on that line number. Here is image of how it looks like.



How to use

You can [set breakpoint via GDB console](#) as well.

Once you set breakpoint, when you start program in debug mode, it will pause execution when program reaches the line where breakpoint is set.

```
main.c
1 Code, compile, run and debug online from anywhere in world.
2
3 *****
4 #include <stdio.h>
5
6
7
8
9
10 int multiply(int n1, int n2) {
11     int result;
12     result = n1 * n2;
13     return result;
14 }
15
16 int factorial(int n) {
17     int fact = 1;
18     int i;
19
20     for(i=1; i<n; i++)
21         fact = multiply(fact, i);
22
23     return fact;
24 }
25
26 int main() {
27
28     printf("Factorial of 5 is %d\n", factorial(5));
29
30     return 0;
31 }
32
```

Breakpoint on line number 12 is hit.
Program paused here.

#	Function	File:Line
0	multiply	main.c:12
1	factorial	main.c:21
2	main	main.c:28

Variable	Value
result	0

Expression	Value
Enter expression to watch	

#	Description	
1	in multiply at main.c:12	X

input | Debug Console

start pause continue step over step into step out

Starting program: /home/a.out

Breakpoint 1, multiply (n1=1, n2=1) at main.c:12
12 result = n1 * n2;
(gdb) ~

main.c

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

```
1
2 #include <stdio.h>
3 int n = 5;
4
5 int main()
6 {
7     int i, j, sum = 0, prod;
8
9     for (i = 1; i <= n; i++)
10     {
11         for (j = 1; j <= n; j++)
12         {
13             prod = i * j;
14             sum = sum + prod;
15         }
16     }
17     printf("%d", sum);
18     return 0;
19 }
```

Call Stack

#	Function	File:Line
0	main	main.c:10

Local Variables

Variable	Value
i	2
j	1
sum	17
prod	2

Display Expressions

Expression	Value	
i	2	✖
j	1	✖

Enter expression to watch

Breakpoints and Watchpoints

	#	Description	
<input checked="" type="checkbox"/>	1	in main at main.c:8	✖

input

Debug Console

start

pause

continue

step over

step into

step out

help

12

prod = i * j;

(gdb) step

13

sum = sum + prod;

(gdb) step

10

for (j = 1; j <= n; j++)

(gdb)





More about functions

Variable length parameters list

It's possible to create functions that receive an *unspecified* number of arguments. Most programs in the text have used the standard library function `printf`, which, as you know, takes a variable number of arguments. As a minimum, `printf` must receive a string as its first argument, but `printf` can receive any number of additional arguments. The function prototype for `printf` is

```
int printf(const char *format, ...);
```

The **ellipsis** (...) in the function prototype indicates that the function receives a *variable number of arguments of any type*. The ellipsis must always be placed at the *end* of the parameter list.

The macros and definitions of the **variable arguments headers** **<stdarg.h>**

Identifier	Explanation
va_list	A <i>type</i> suitable for holding information needed by macros va_start, va_arg and va_end. To access the arguments in a variable-length argument list, an object of type va_list must be defined.
va_start	A <i>macro</i> that's invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with va_list for use by the va_arg and va_end macros.
va_arg	A <i>macro</i> that expands to the value of the next argument in the variable length argument list—the value has the type specified as the macro's second argument. Each invocation of va_arg modifies the object declared with va_list so that it points to the next argument in the list.
va_end	A <i>macro</i> that facilitates a normal return from a function whose variable length argument list was referred to by the va_start macro.

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

double average(int i, ...); // function prototype

int main()
{
    double a = 12.6, b = 42.7, c = 41.6, d = 51.4;
    printf("%.1lf %.1lf %.1lf %.1lf\n", a,b,c,d);
    printf("average of %.1lf and %.1lf is %.1lf\n", a,b, average(2, a, b));
    printf("average of %.1lf, %.1lf and %.1lf is %.1lf\n", a,b,c, average(3, a,b,c));
    printf("average of %.1lf, %.1lf, %.1lf and %.1lf is %.1lf\n", a, b, c, d,
    average(4, a, b, c,d));
}
```

```
// function average
```

```
double average(int i, ...)
```

```
{
```

```
    double total = 0;
```

```
    int j;
```

```
    va_list ap;
```

```
    va_start(ap, i);
```

```
    for (j = 1; j <= i; j++)
```

```
        total = total + va_arg(ap, double);
```

```
    va_end(ap);
```

```
    return total / i;
```

```
}
```

Crearea obiectului
care continue lista
de argumente

stocarea listei
curente de
argumente

Accesarea
argumentului urmator
cu specificarea tipului

lichidarea obiectului
care continue lista
de argumente

C:\Users\CTI UST\OneDrive\Work\UTM\PC\Exemple surse\directives\directives_006.exe

12.6 42.7 41.6 51.4

average of 12.6 and 42.7 is 27.7

average of 12.6, 42.7 and 41.6 is 32.3

average of 12.6, 42.7, 41.6 and 51.4 is 37.1

Process exited after 0.01859 seconds with return value 45

Press any key to continue . . .