C/C++
(“rinforzino”)

Matteo Duranti

matteo.duranti@pg.infn.it

(cfr. https://www.tutorialspoint.com/cprogramming/c_file_io.htm

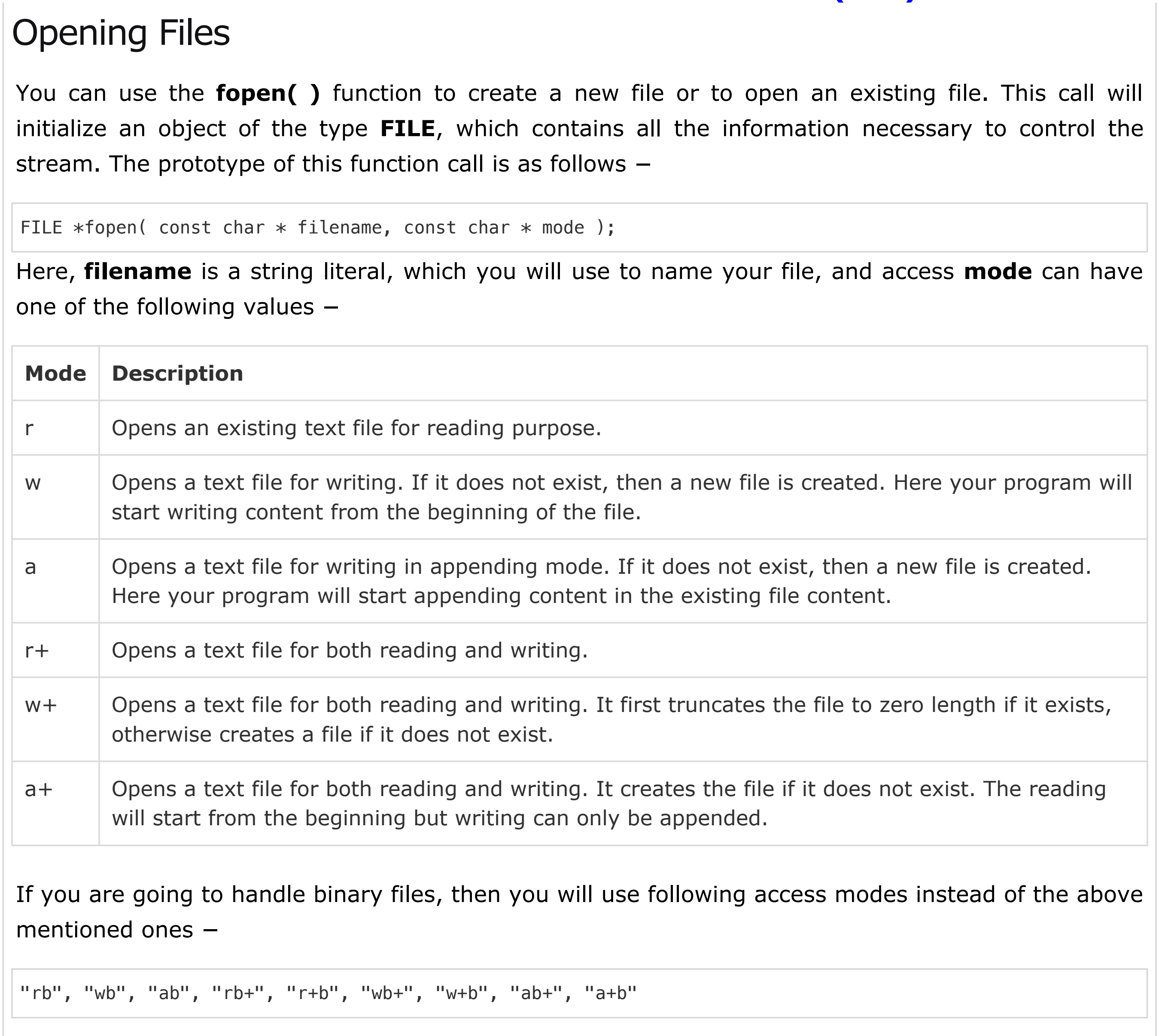
https://www.tutorialspoint.com/c_standard_library/c_function_feof.htm

http://www.cplusplus.com/doc/tutorial/files/

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

http://www.cplusplus.com/articles/2LywvCM9/)

Scrittura/Lettura file (C)



Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows −

```
FILE *fopen( const char * filename, const char * mode );
```

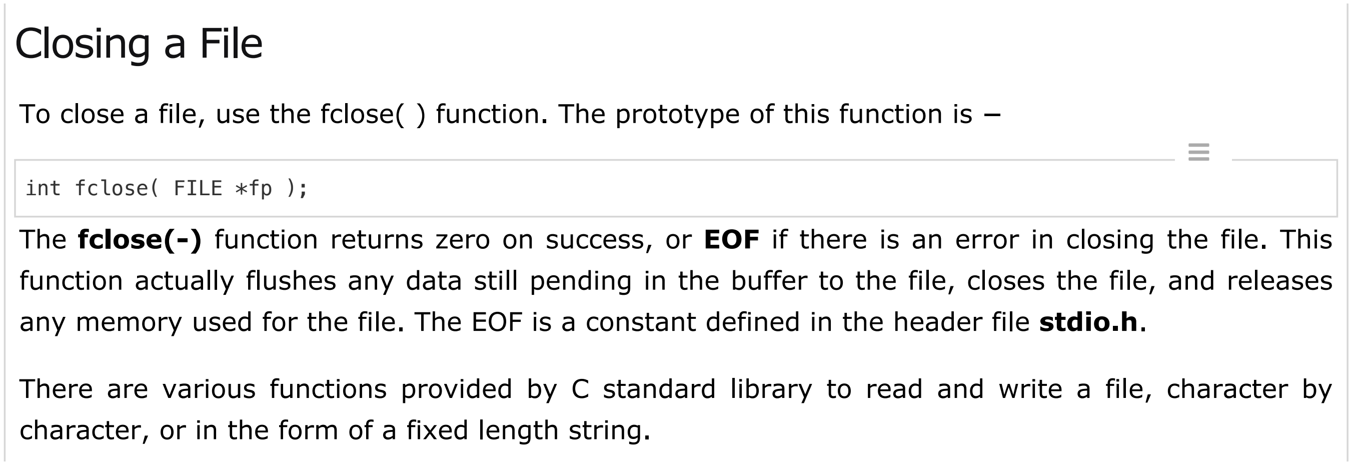
Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values −

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones −

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Scrittura/Lettura file (C)



Closing a File

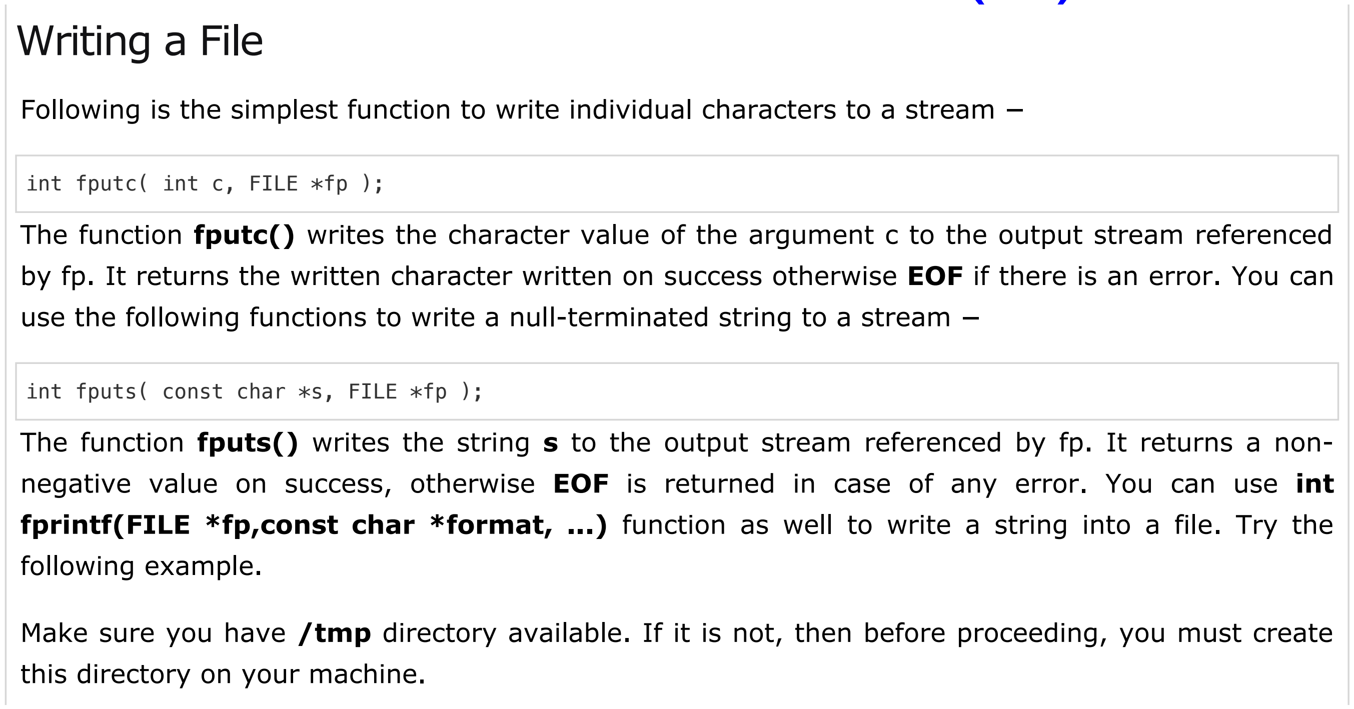
To close a file, use the fclose( ) function. The prototype of this function is −

```
int fclose( FILE *fp );
```

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Scrittura/Lettura file (C)



Writing a File

Following is the simplest function to write individual characters to a stream −

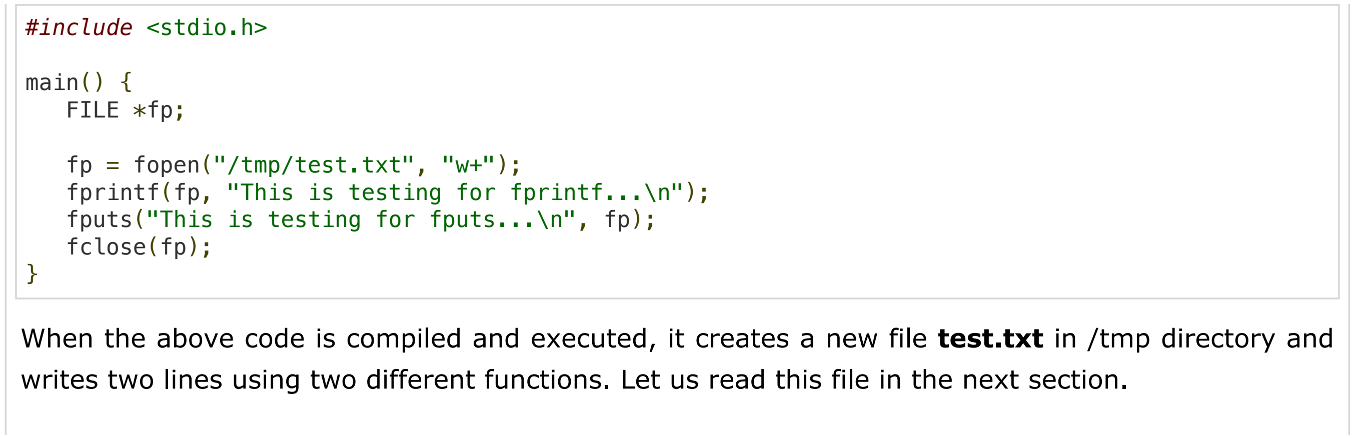
```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream −

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

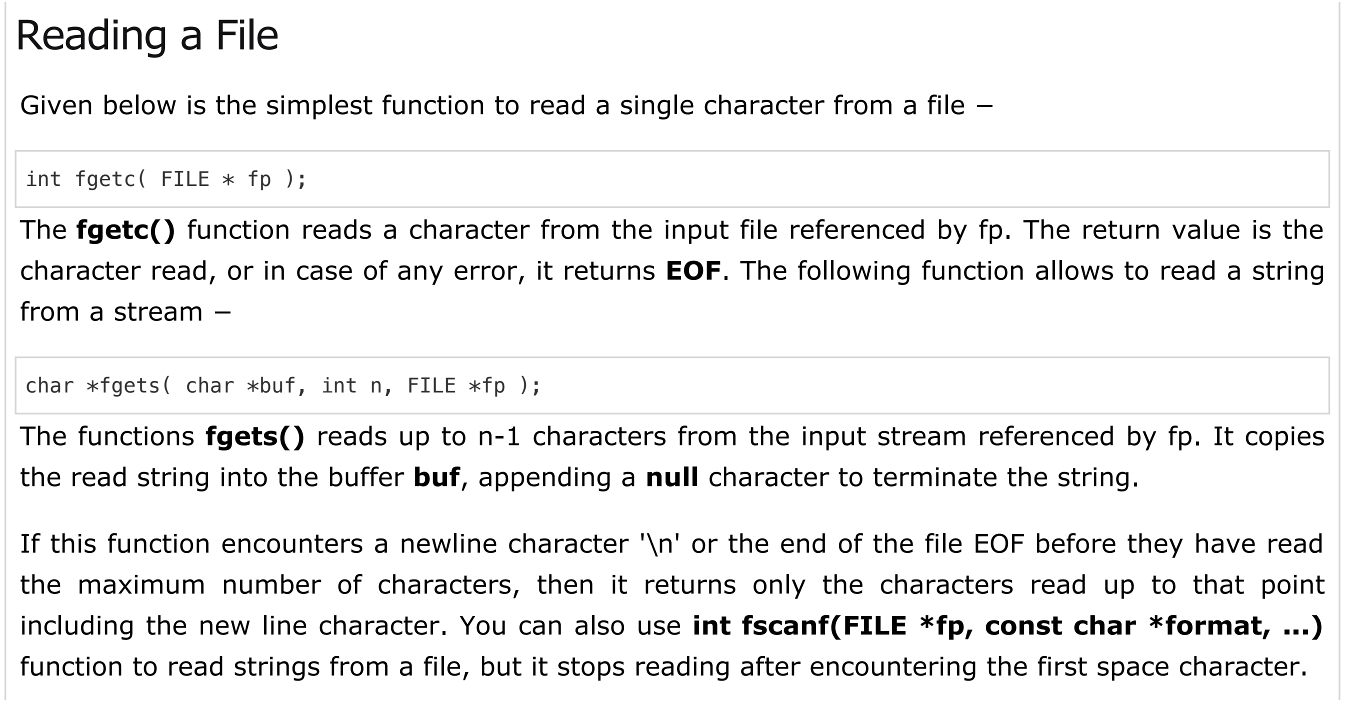
Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.



```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

Scrittura/Lettura file (C)



Reading a File

Given below is the simplest function to read a single character from a file −

```
int fgetc( FILE * fp );
```

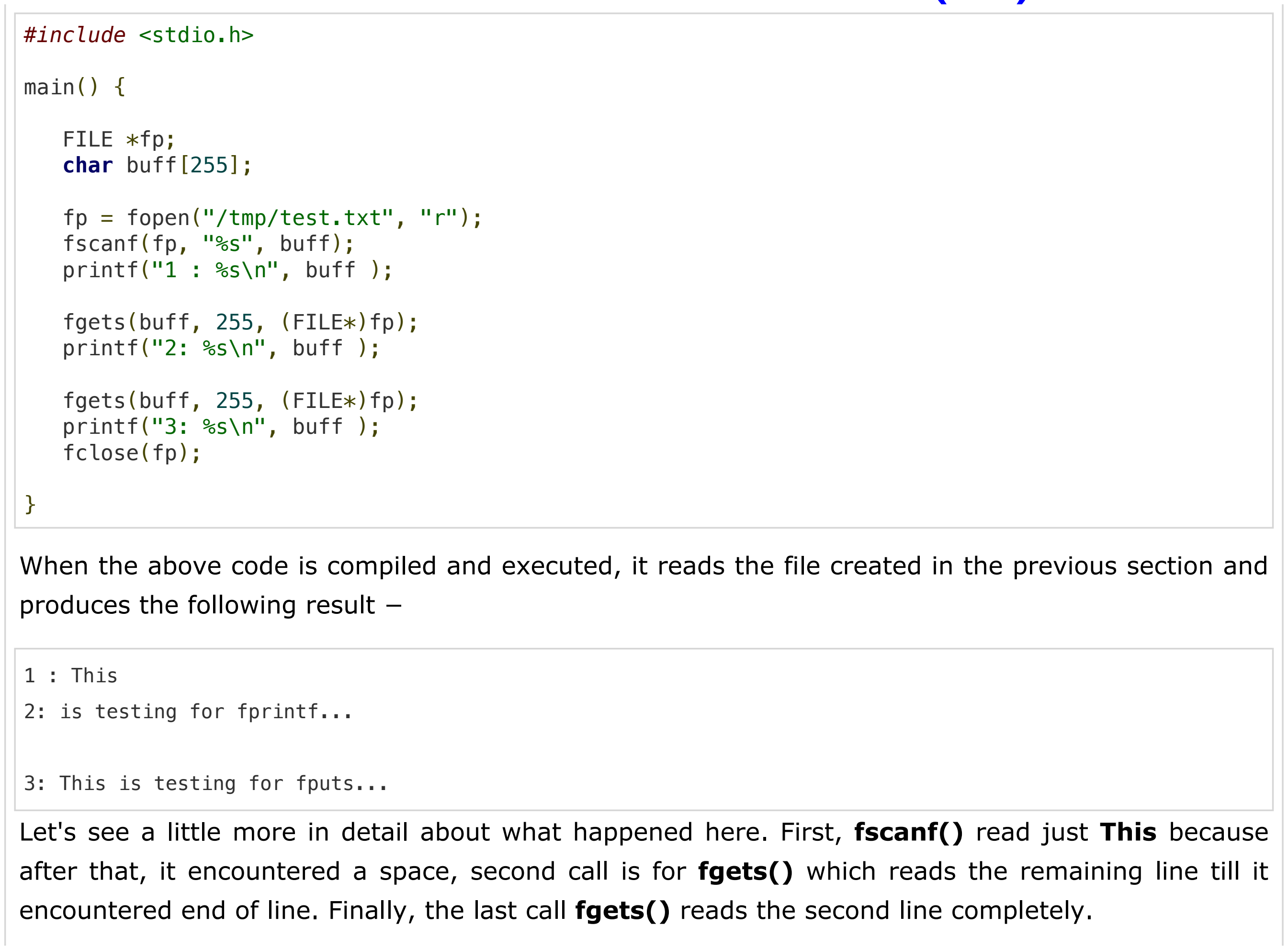
The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream −

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

Scrittura/Lettura file (C)



```c
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

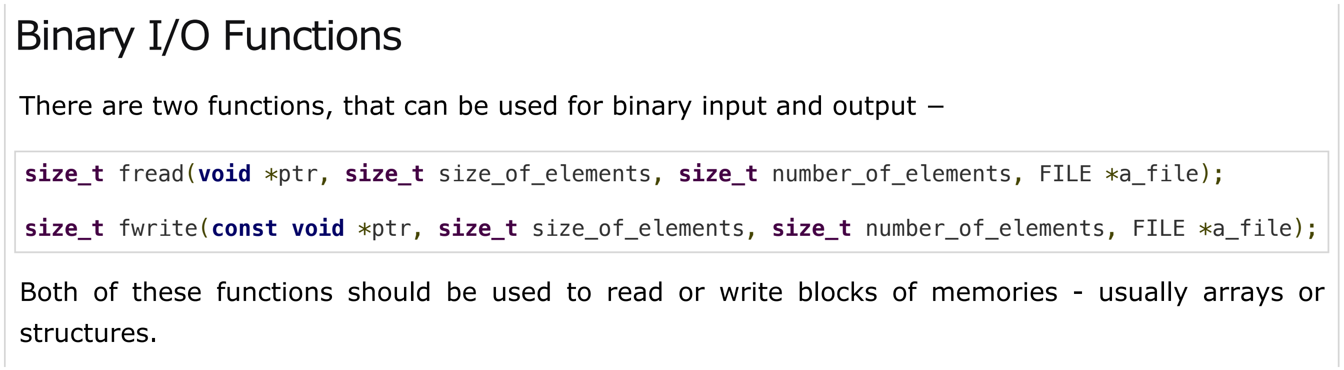
```
1 : This
2: is testing for fprintf...


3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

Scrittura/Lettura file (C)

```c
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   if(fp == NULL) {
      perror("Error in opening file");
      return(-1);
   }

   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);

   return(0);
}
```

Scrittura/Lettura file (C)



Binary I/O Functions

There are two functions, that can be used for binary input and output −

```c
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

Scrittura/Lettura file (C++)

Input/output with files

C++ provides the following classes to perform output and input of characters to/from files:

- **`ofstream`:** Stream class to write on files
- **`ifstream`:** Stream class to read from files
- **`fstream`:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream` and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

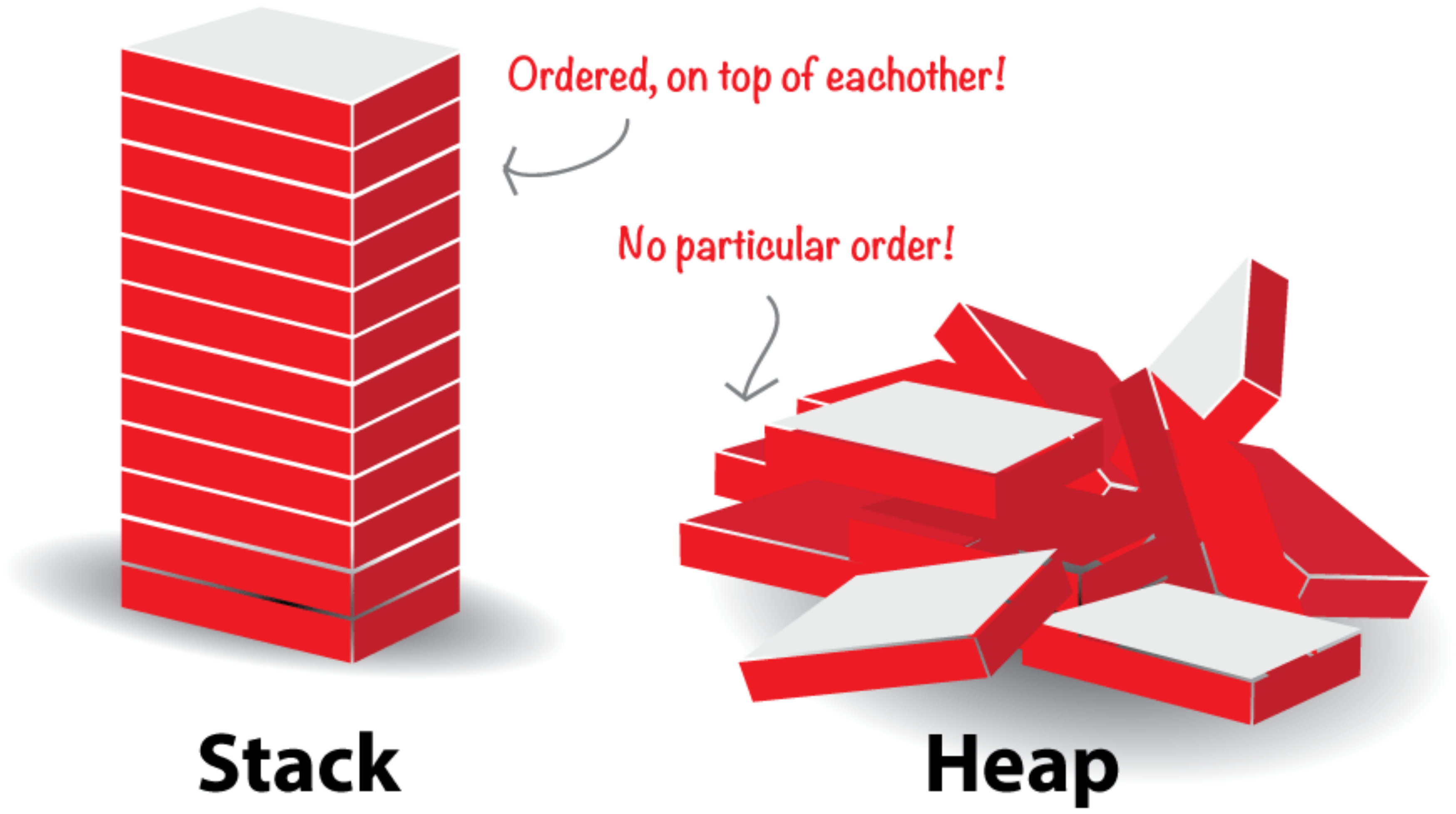
```cpp
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file.
```

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

But let's go step by step:

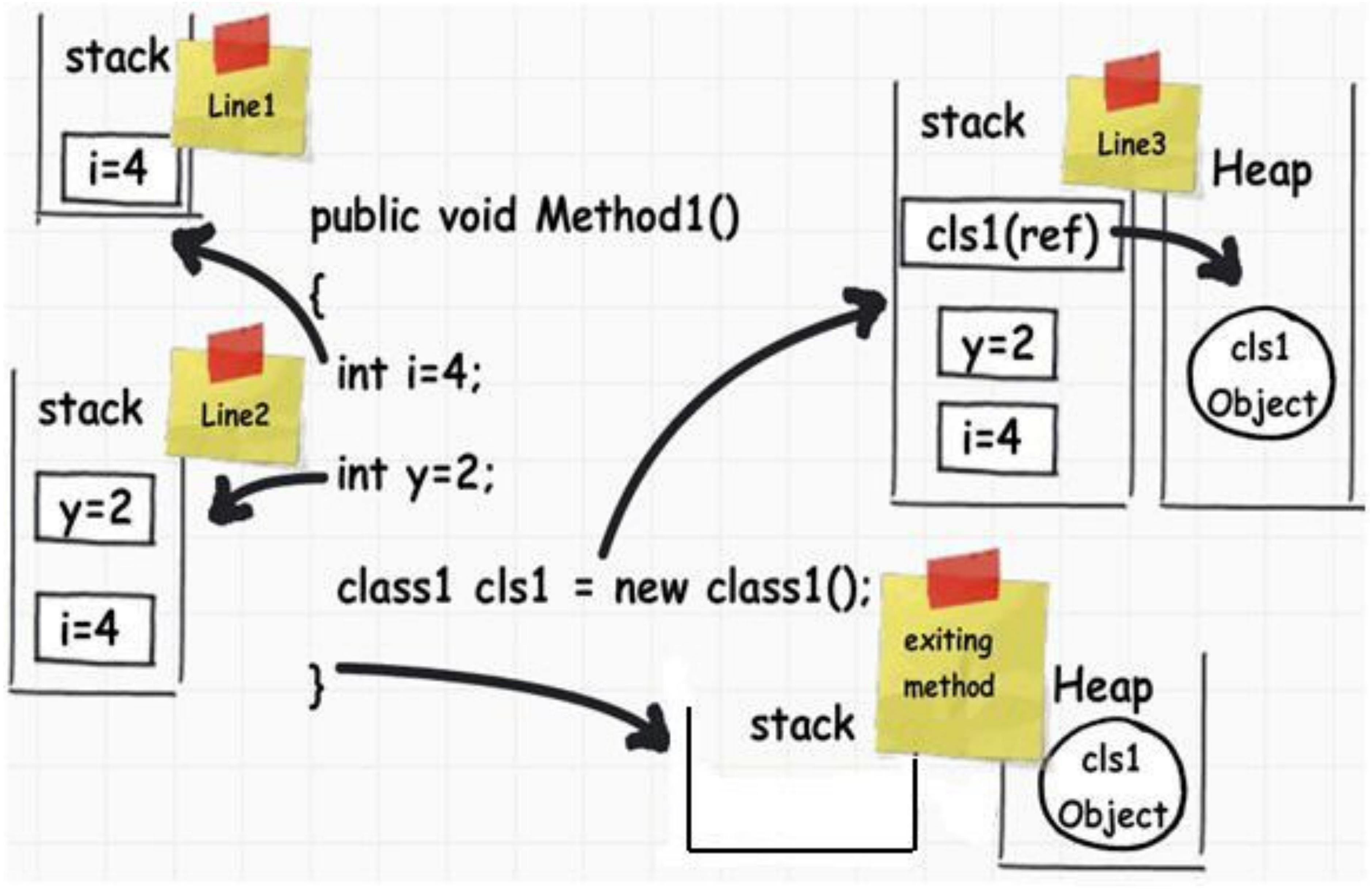
Memoria



Memoria

	stack	**heap**
access speed	fast	slow
memory management	CPU - automatic	manual allocation and deallocation
efficient use of memory	yes	no - may become fragmented, or leak
variable scope	local	global
size	limited (OS dependent)	no limit (or rather the limit of physical memory)
variable resizing	no	yes
need pointers?	no	yes

Memoria


stack
Line1
i=4
public void Method1()
{
int i=4;
stack
Line2
y=2
i=4
int y=2;
class1 cls1 = new class1();
}
stack
Line3
Heap
cls1(ref)
y=2
i=4
cls1 Object
exiting method
stack
Heap
cls1 Object

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
     double base; double height;          //members of the class

     double getArea(void);                //methods of the class
     void setBaseHeight(double,double);   //methods of the class
};

double Rectangle::getArea(void){
     return base*height;
}

void Rectangle::setBaseHeight(double b, double h){
     base=b;
     height=h;
}




int main(){
   Rectangle rec;
   rec.setBaseHeight(2, 5.5);  // the operator "." is used to access methods
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 11
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double base; double height;          //members of the class

      double getArea(void);                //methods of the class
      void setBaseHeight(double,double);   //methods of the class
};

double Rectangle::getArea(void){
     return base*height;
}

void Rectangle::setBaseHeight(double b, double h){
     base=b;
     height=h;
}







int main(){
   Rectangle* prec = new Rectangle();
   prec->setBaseHeight(2, 5.5);  // the operator "->" is used to access methods
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 11
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      Rectangle();   //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();   //destructor
};

Rectangle::Rectangle(){
           printf("Creating-Rectangle\n");
           base=1; height=1;
           return;
}

Rectangle::~Rectangle(){
           printf("Destroying-Rectangle\n");
           return;
}



int main(){
   Rectangle rec;
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   rec.setBaseHeight(2, 4);
   area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8

Destroying-Rectangle
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
           printf(“Creating-Rectangle\n”);
           base=1; height=1;
           return;
}

Rectangle::~Rectangle(){
           printf(“Destroying-Rectangle\n”);
           return;
}



int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<“Area “<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1

                                ?

Area 8

Destroying-Rectangle
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
           printf(“Creating-Rectangle\n”);
           base=1; height=1;
           return;
}

Rectangle::~Rectangle(){
           printf(“Destroying-Rectangle\n”);
           return;
}



int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<“Area “<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
           printf(“Creating-Rectangle\n”);
           base=1; height=1;
           return;
}

Rectangle::~Rectangle(){
           printf(“Destroying-Rectangle\n”);
           return;
}


int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<“Area “<<area<<endl;
   delete prec;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
Destroying-Rectangle
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
           printf("Creating-Rectangle\n");
           base=1; height=1;
           return;
}

Rectangle::~Rectangle(){
           printf("Destroying-Rectangle\n");
           return;
}


int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
Destroying-Rectangle
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
     private:
      double base;
      double height;
};
```

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<“Area “<<area<<endl;
   delete prec;
   return 0;
}
```

Nemmeno compila…

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double base;
      double height;
};
```

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   if (prec->base=3) {
       printf("Base is 3\n");
   }
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

```
Area 1


Area 8

Base is 3

Area 12
```

!

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double getBase();
      double getHeight();
     private:
      double base;
      double height;
};

double Rectangle::getBase(){ return base; }

double Rectangle::getHeight(){ return height; }



 int main(){
    Rectangle* prec = new Rectangle();
    double area = prec->getArea();
    cout<<"Area "<<area<<endl;
    prec->base=2; prec->height=4;
    area = prec->getArea();
    cout<<"Area "<<area<<endl;
    if (prec->getBase()=3) {
        printf("Base is 3\n");
    }
    cout<<"Area "<<area<<endl;
    delete prec;
    return 0;
 }
```

```
Area 1


Area 8

Base is 3

Area 8   !
```

(ovviamente l'if è comunque sbagliato…)

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      inline double getBase(){ return base; };
      inline double getHeight() { return height; };
     private:
      double base;
      double height;
};
```

Questo (*inline*) dice al compilatore di sostituire la chiamata a funzione con tutte le istruzioni che essa contiene:
✔ evita all'esecuzione del programma di muovere il puntatore sullo stack (fino a dove c'è "il pezzo di eseguibile che esegue la funzione) e di continuare la sua esecuzione sequenziale
✖ l'impronta in memoria (stack) dell'eseguibile è in generale più grande. Va utilizzato solo se la funzione è "piccola"

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   if (prec->getBase()=3) {
       printf("Base is 3\n");
   }
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

C++ Classes

```
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      static void whatAmI();
      double base;
      double height;
};

void Rectangle::whatAmI(){
     printf("I am a Rectangle!\n");
}
```

```
int main(){
   Rectangle::whatAmI();
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->whatAmI();
   delete prec;
   return 0;
}
```

```
I am a Rectangle!


Area 1


Area 8
I am a Rectangle!
```

C++ Classes

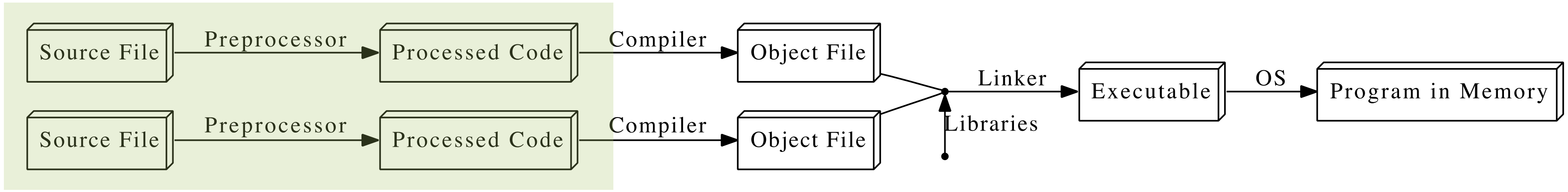
```cpp
#include <iostream>
using namespace std;

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double getDiffArea(Rectangle* prec2);
     private:
      double base;
      double height;
};

double Rectangle::getDiffArea(Rectangle* prec2){
     double area  = getArea();
     double area2 = prec2->getArea();
     return area-area2;
}
```

```cpp
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   Rectangle* prec2 = new Rectangle();
   prec2->setBaseHeight(2, 4);
   area = prec2->getArea();
   cout<<“Area “<<area<<endl;
   area = prec->getDiffArea(prec2);
   cout<<“Area “<<area<<endl;
   return 0;
}
```

```
Area 1


Area 8

Area -7
```

Preprocessor directives

- Steps from source code to machine level instructions



- The steps are performed before the program starts to run
  - Some languages follow (more or less) the same procedure BUT during the execution process. This slows down the program execution.
    - This is one of the reasons why C++ code runs far faster than code in many more recent languages.
- The Preprocessor modifies the source code according to user directives

```
#include “external_header.h” //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int main(){
  #ifdef _DEBUG_
  cout<<“Do This”<<endl;
  #else
  cout<<“Do That”<<endl;
  #endif
  return 0;
}
```

Preprocessor directives

```
vvagelli@Firefly~/test $ g++ -g -O -Wall test.C -o exe/test -D_DEBUG_ -I/
Users/vvagelli/root/root5.34/include  -L/Users/vvagelli/root/root5.34/lib -
lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-rpath,/
Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm –ldl
```

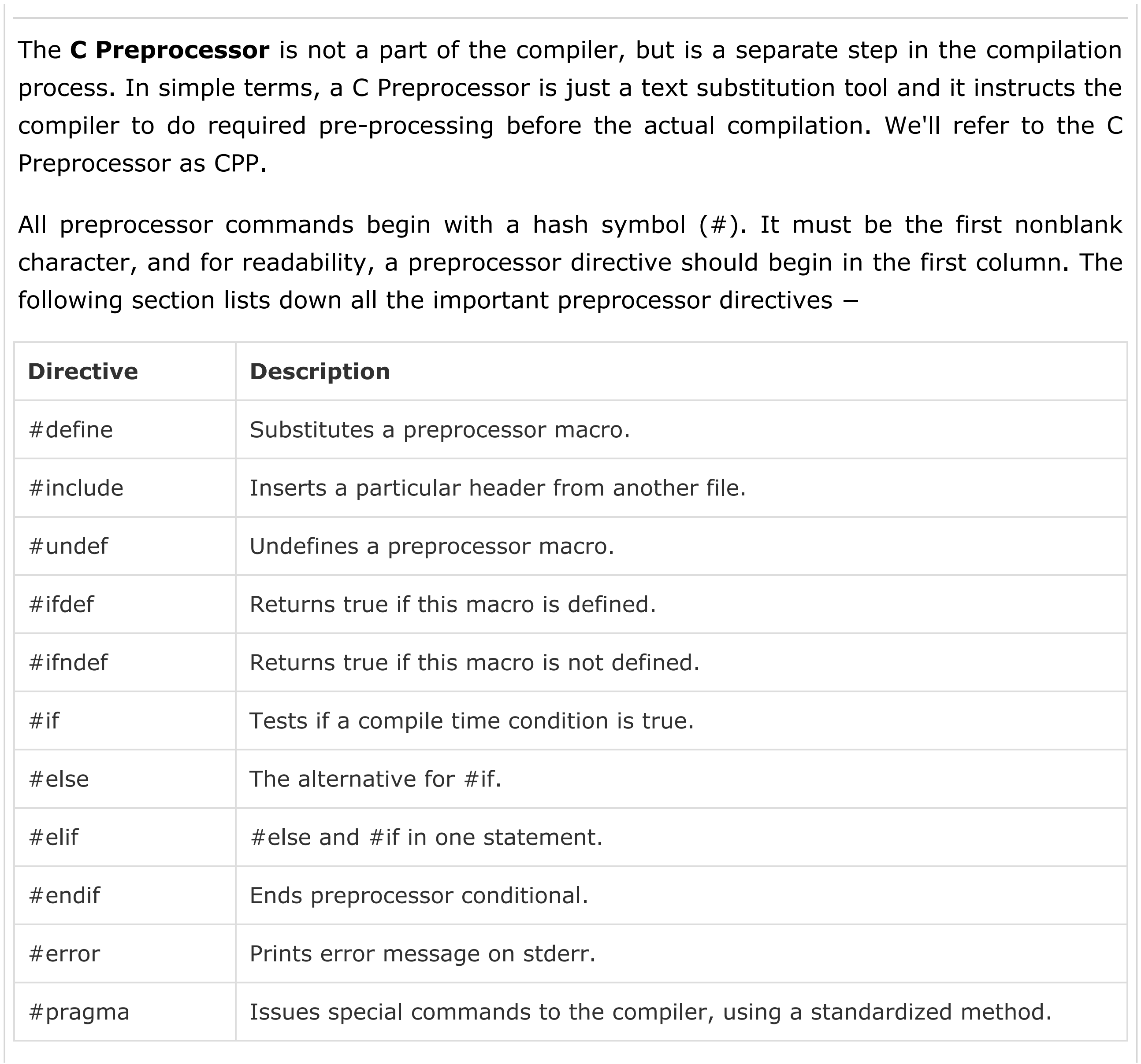
```
#include “external_header.h” //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int main(){
  #ifdef _DEBUG_
  cout<<“Do This”<<endl;
  #else
  cout<<“Do That”<<endl;
  #endif
  return 0;
}
```

Preprocessor directives

```
vvagelli@Firefly~/test $ g++ -g -O -Wall test.C -o exe/test -D_DEBUG_ -I/
Users/vvagelli/root/root5.34/include  -L/Users/vvagelli/root/root5.34/lib -
lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-rpath,/
Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm –ldl
```

```
#include “external_header.h” //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int array_sum[CONSTANT];

int main(){
  int sum=0;
  for (int ii=0; ii<CONSTANT; ii++) {
    sum+=ii;
    array_sum[ii]=sum;
    #ifdef _DEBUG_
    printf(“ii=%d) sum=%d\n”, ii, sum);
    #endif
  }
  return 0;
}
```

Direttive preprocessore

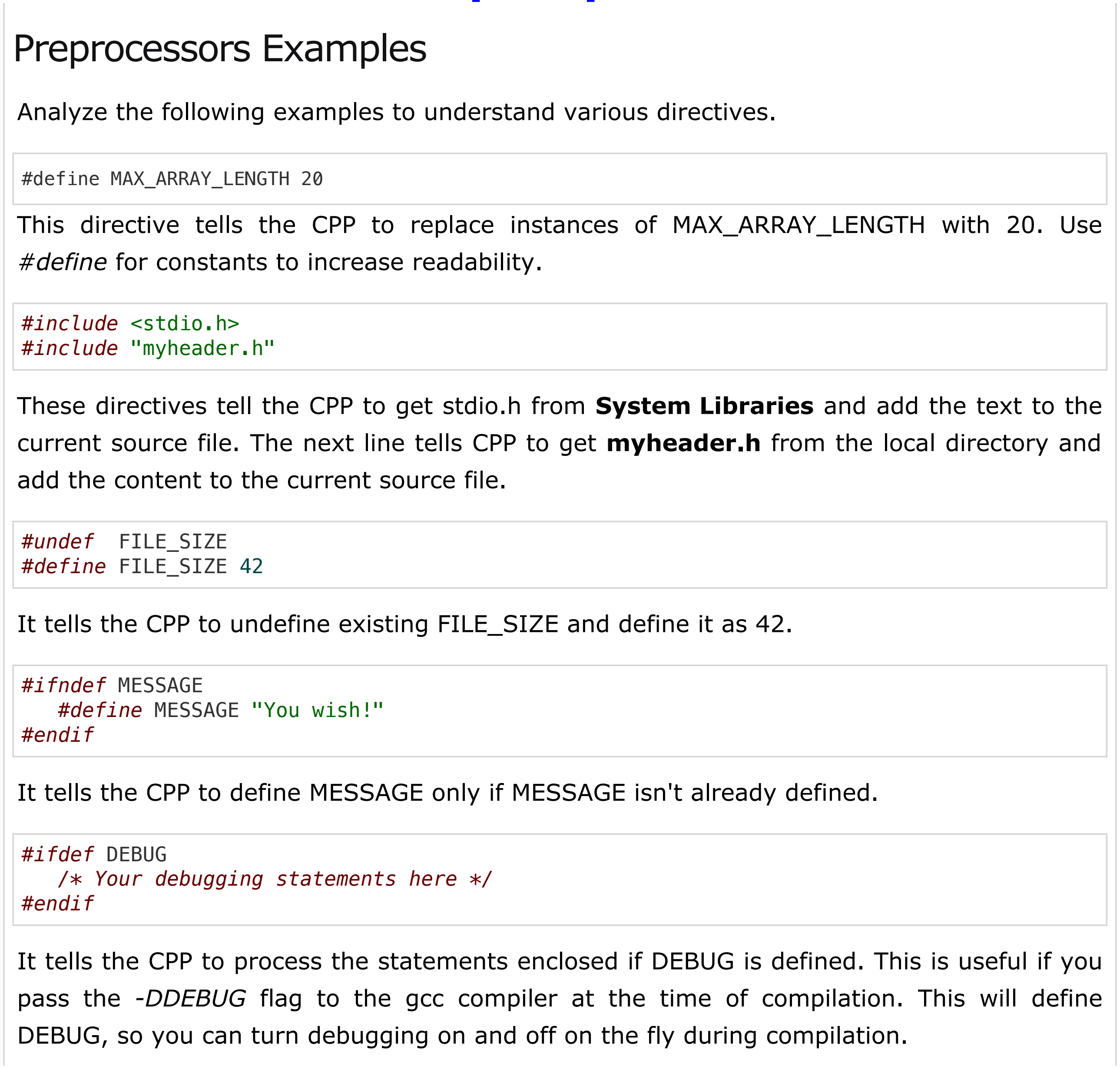


The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Directive	**Description**
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#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.
#pragma	Issues special commands to the compiler, using a standardized method.

Direttive preprocessore



Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

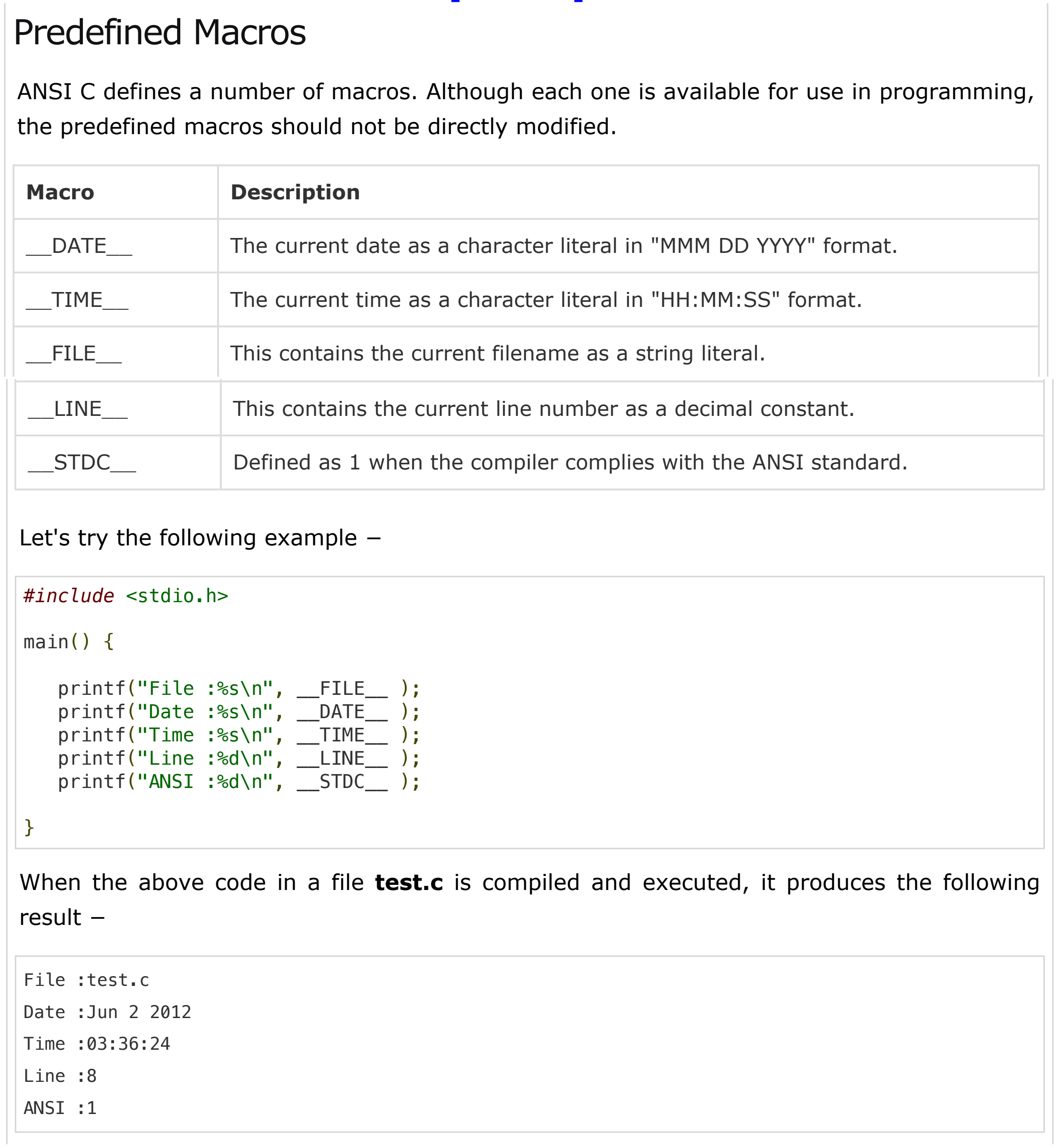
```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Direttive preprocessore



Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

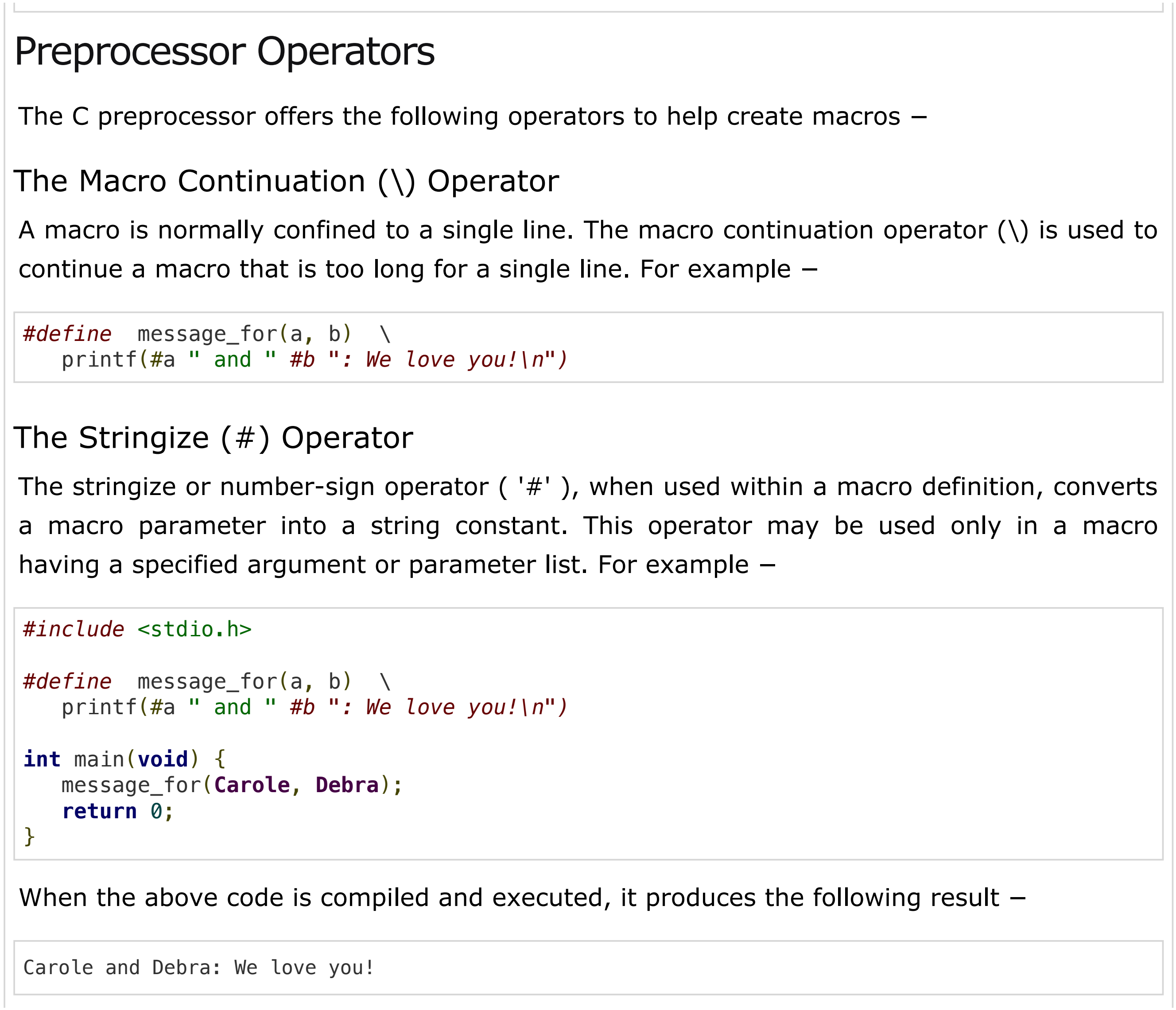
Let's try the following example −

```
#include <stdio.h>

main() {

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

}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result −

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Direttive preprocessore



Preprocessor Operators

The C preprocessor offers the following operators to help create macros −

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example −

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")
```

The Stringize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example −

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")

int main(void) {
   message_for(Carole, Debra);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Carole and Debra: We love you!
```

Direttive preprocessore

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example −

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
   int token34 = 40;
   tokenpaster(34);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

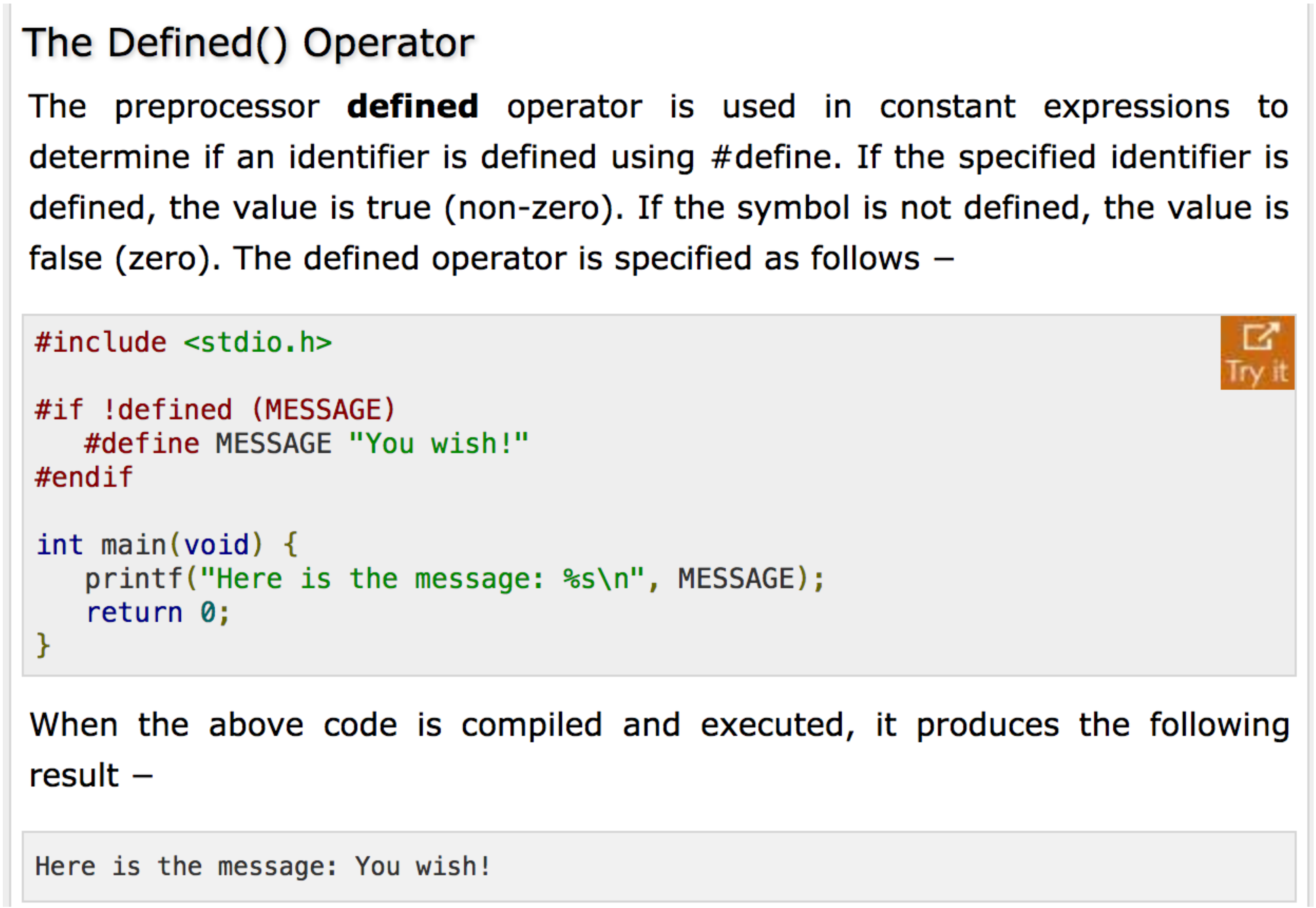
```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor −

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

Direttive preprocessore



The Defined() Operator

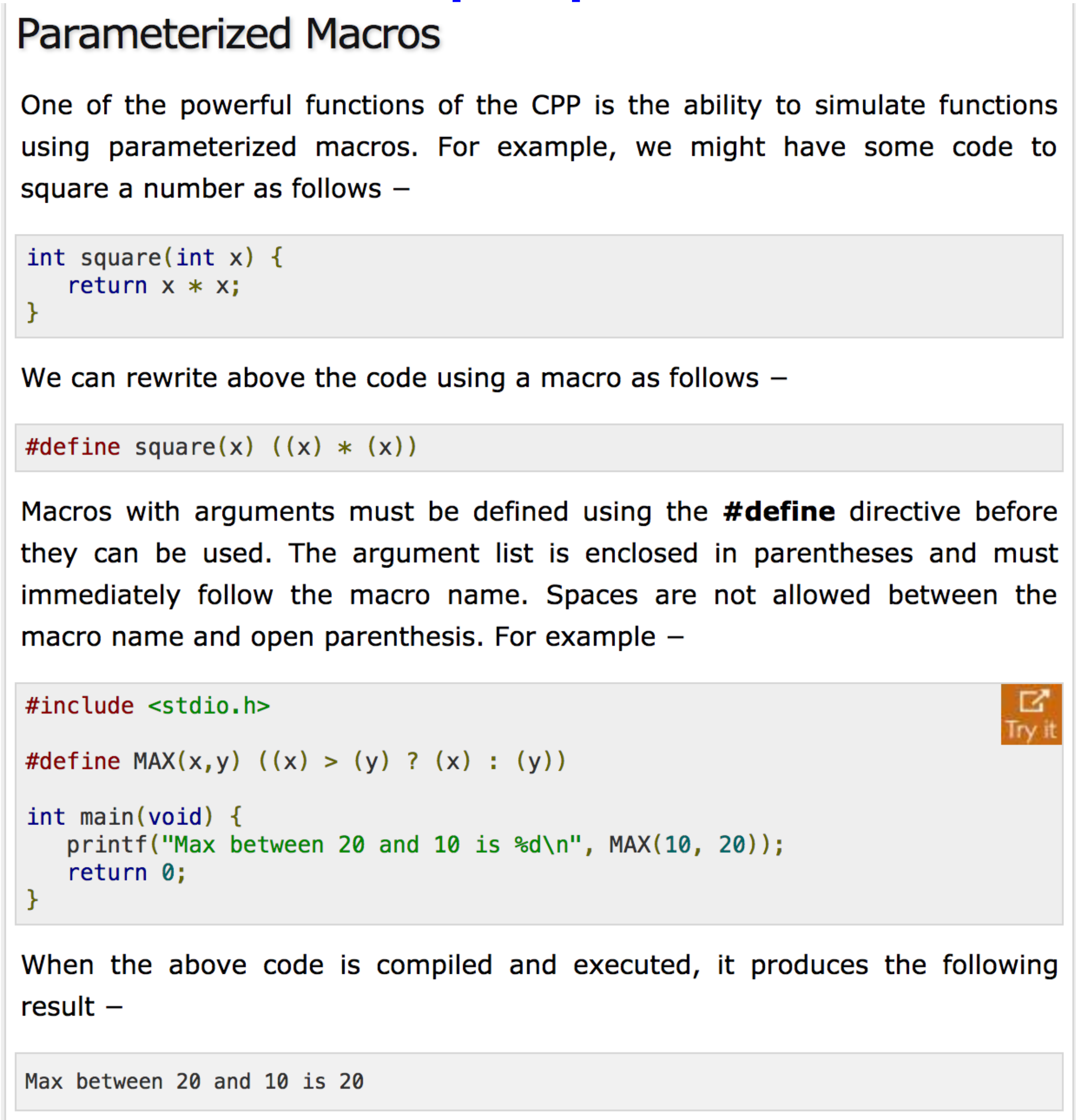
The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows −

```c
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void) {
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Here is the message: You wish!
```

Direttive preprocessore



Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows –

```
int square(int x) {
   return x * x;
}
```

We can rewrite above the code using a macro as follows –

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example –

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Max between 20 and 10 is 20
```