COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Input and output

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Input and Output

For input and output we shall use the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs.

A text stream consists of a sequence of lines; each line ends with a newline character.

The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with **getchar**:

int getchar(void)

The function

int putchar(int)

is used for output: **putchar**(c) puts the character c on the standard output.

Input and Output

Input and/or output can usually be directed from / to a file with *<filename* or *>filename*:

program <infile causes program to read characters from infile instead of standard input

program >outfile will write the standard output to outfile instead of standard output.

If pipes are supported,

program | anotherprogram

puts the standard output of program into the standard input of anotherprogram.

Each source file that refers to an input/output library function must contain the line *#include <stdio.h>* before the first reference.

Input and Output

- Formatted Output – printf
- Formatted Input – scanf
- File access
- Line Input and Output
- Error Handling - Stderr and Exit
- Variable-length Argument Lists

Input and Output: formatted output

The output function **printf** (stands for "print formatted“) translates internal values to characters.

The declaration is

*int printf(char *format, arg1, arg2, ...);*

printf allows you to print data of various types, such as integers, floating-point numbers, characters, and strings, with precise formatting, it converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf.

Input and Output

Each conversion specification begins with a **%** and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An *h* if the integer is to be printed as a short, or *l* (letter ell) if as a long.

If the character after the % is not a conversion specification, the behavior is undefined.

Input and Output – basic printf conversion

Character	Argument type; Printed As
d, i	***int***; decimal number
o	***int***; unsigned octal number (without a leading zero)
x, X	***int***; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	***int***; unsigned decimal number
c	***int***; single character
s	***char ****; print characters from the string until a '\0' or the number of characters given by the precision.
f	***double***; *[-]m.dddddd*, where the number of d's is given by the precision (default 6)
e, E	***double***; *[-]m.dddddde+/-xx* or *[-]m.ddddddE+/-xx*, where the number of d's is given by the precision (default 6)
g, G	***double***; use *%e* or *%E* if the exponent is less than -4 or greater than or equal to the precision; otherwise use *%f*. Trailing zeros and a trailing decimal point are not printed
p	***void ****; pointer (implementation-dependent representation)
%	no argument is converted; print a %

Input and Output – basic printf conversion

There is the possibility to specify a width or precision by using an argument by means of the character *, in which case the value is computed by converting the next argument (which must be an int).

For example

- to print at most max characters from a string s,
  *printf("%.*s", max, s);*
- to print characters from a string s in a minimum field width of max length,
  *printf("%*s", max, s);*

Input and Output – basic printf conversion

Let’s consider the precision that relates to string arguments. The following rows shows the effect of a variety of specifications in printing ``hello, world'' (12 characters). We have put colons around each field.

:%s: :hello, world:
:%10s: :hello, world:
:%.10s: :hello, wor:
:%-10s: :hello, world:
:%.15s: :hello, world:
:%-15s: :hello, world   :
:%15.10s: :   hello, wor:
:%-15.10s: :hello, wor   :

Input and Output

Common Escape Sequences:

Escape sequences are special characters used within format specifiers to control the formatting of the output. Common escape sequences include:

\n: Newline (line feed)

\t: Tab

\": Double quotation mark

\': Single quotation mark

****: Backslash

Error Handling:

It's essential to ensure that the number and types of arguments provided match the format specifiers in the printf statement. Mismatches can lead to undefined behavior or errors in your program.

Input and Output

If s is an array of char the invocation

printf("%s", s);

is safe, while the invocation

printf(s);

fails if in the string s there are characters %

The function **sprintf** performs the same conversions as printf does, but stores the output in a string:

*int sprintf(char *buffer, char *format, arg1, arg2, ...);*

sprintf formats the arguments in arg1, arg2, etc., according to format as before, but places the result in the array of char buffer instead of the standard output; buffer must be big enough to receive the result.

Input and Output – std::cout

In C++, **std::cout** is an object of the class std::ostream and is used for output to the standard output stream, which is usually the terminal or console. It is part of the C++ Standard Library and is defined in the <iostream> header file.

The syntax is *std::cout << expression;*

The **<<** operator is called the insertion operator and is used to send data to the output stream (std::cout in this case). It displays the result of the expression or variable on the screen.

Input and Output – std::cout


```
#include <iostream>

int main() {
    int age = 25;
    std::cout << "I am " << age << " years old." << std::endl;
    return 0;
}
```

I am 25 years old.

- The << operator directs data (e.g., "I am " and age and “ years old.") to the output stream (std::cout).
- std::endl is used to insert a newline character and to flush the output buffer (though '\n' can be used as an alternative without flushing the buffer).

Input and Output – std::cout

1. **Multiple Outputs in One Line**: You can chain multiple insertions with << to output different types of data in a single statement.
2. **Types of Data**: You can output various data types using std::cout, such as integers, floating-point numbers, characters, strings, etc.:

```
int num = 42;
double pi = 3.14159;
char grade = 'A';
std::cout << "Number: " << num << ", Pi: " << pi << ", Grade: " << grade << std::endl;
```

Number: 42, Pi: 3.14159, Grade: A

3. **Using std::endl vs. \n**: std::endl: Inserts a newline character and flushes the output buffer. \n: Inserts a newline character not flushing the buffer.
4. **Flushing the Output**: Flushing ensures that all data in the buffer is sent to the output. It's typically done automatically when the program finishes, but std::endl forces it immediately.

Input and Output – std::cout

5. **Namespaces (std::)**: The std:: before cout refers to the namespace std, which is part of the C++ Standard Library. You can omit std:: by using the statement using *namespace std;*

   It's considered better practice to explicitly use std::cout to avoid potential naming conflicts in larger programs.

6. **Outputting Special Characters**: You can use escape sequences (like in C) to print special characters. For example, \n for a newline, \t for a tab, etc.

   std::cout << "Line1\nLine2\tTabbed" << std::endl;

   Line1

   Line2 Tabbed

7. **Formatting Output**: C++ offers powerful formatting options for controlling how data is outputted. These options are available through manipulators (defined in <iomanip>):

   std::setw(): Set the width for the next output.

   std::setprecision(): Set the number of digits of precision for floating-point numbers.

   std::fixed: Force floating-point numbers to be displayed in fixed-point notation.

Input and Output – std::cout


```
#include <iostream>
#include <iomanip>  // For setw, setprecision

int main() {
    double pi = 3.14159;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Pi (2 decimal places): " << pi << std::endl;
    std::cout << std::setw(10) << 42 << std::endl;  // Output 42 with a width of 10 spaces
    return 0;
}
```

```
Pi (2 decimal places): 3.14
        42
```

Input and Output – std::cout


```
#include <iostream>
#include <iomanip>

int main() {
    int a = 42;
    double b = 3.14159;

    // Basic output
    std::cout << "a: " << a << ", b: " << b << std::endl;

    // Formatting with manipulators
    std::cout << std::setw(10) << a << std::endl;
    std::cout << std::fixed << std::setprecision(3) << b << std::endl;

    return 0;
}
```

This will print

- a right-aligned with 10 spaces
- b with 3 decimal places.

Input and Output: formatted input

The I/O so far has been character oriented, reading one character at a time.

It is hard to interpret two values on the same line

printf("enter your age and weight ");

A name and a number

printf("enter your name and age ");

Other combinations

These problems could be solved by asking the user for one piece of input per line.

- This might be unnatural.
- It is also awkward when there are many fields.

Input and Output: formatted input

The function **scanf** is the input analog of printf, providing many of the same conversion facilities in the opposite direction. The scanf function is part of the stdio.h library, and its syntax is as follows

*int scanf(char *format, ...)*

scanf reads characters from the standard input, interprets them according to the specification provided in format string, and stores the results through the remaining arguments. These remaining arguments, that ***must be pointers***, indicate where the corresponding converted input shall be stored.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns the number of successfully matched and assigned input items. This can be used to decide how many items were found.

Input and Output: formatted input

On the end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- blanks or tabs, which are not ignored
- ordinary characters (not %), which are expected to match the next non-white space character of the input stream
- conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

Input and Output: formatted input

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made.

An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. White space characters are blank, tab, newline, carriage return, vertical tab, and form feed.

The conversion character indicates the interpretation of the input field.

Input and Output – basic scanf conversion

Character	Input data; Argument type
d	decimal integer; ***int ****
i	integer; ***int ****. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); ***int ****
x	hexadecimal integer (with or without leading 0x or 0X); ***int ****
u	unsigned decimal integer; ***unsigned int ****
c	characters; ***char ****. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use ***%1s***
s	character string (not quoted); ***char ****, pointing to an array of characters long enough for the string and a terminating ***'\0'*** that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; ***float ****
%	literal %; no assignment is made.

Input and Output: formatted input

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list.

```
#include <stdio.h>
main() /* rudimentary calculator */
{
    double sum, v;
    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Input and Output: formatted input

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with the scanf statement:

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

to read input lines that contain dates of the form
25 Dec 1988
The scanf statement is

```
int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);
```

Input and Output: formatted input

scanf ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with string variant input management of scanf.

Warning: the arguments to scanf must be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.
Some examples in the next slides.

Input and Output: formatted input

Reading an integer:

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);  // The address of `num` is passed using &
    printf("You entered: %d\n", num);
    return 0;
}
```

The format specifier %d tells scanf to expect an integer. The &num passes the address of the variable num, where the value will be stored.

Input and Output: formatted input

Reading Multiple Inputs:

```
#include <stdio.h>
int main() {
    int age;
    float salary;
    printf("Enter your age and salary: ");
    scanf("%d %f", &age, &salary);  // Reading two values: integer and float
    printf("Age: %d, Salary: %.2f\n", age, salary);
    return 0;
}
```

The format string %d %f tells scanf to expect an integer followed by a floating-point number. **The inputs should be separated by whitespace (space, tab, or newline).**

Input and Output: formatted input

Reading a String:

```
#include <stdio.h>
int main() {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);  // No need for `&` with arrays
    printf("Hello, %s!\n", name);
    return 0;
}
```

The format specifier %s reads a string. Unlike other data types, when reading a string, you don't need to use & since arrays are already pointers.

Input and Output: formatted input

Reading a single character:

```
#include <stdio.h>
int main() {
    char ch;
    printf("Enter a character: ");
    scanf("%c", &ch);  // Reading a single character
    printf("You entered: %c\n", ch);
    return 0;
}
```

The format specifier %c reads a single character, including spaces or newlines.

Input and Output: formatted input

Handling Input with Spaces.
For strings with spaces, scanf will stop reading at the first space. If you want to input a full line of text including spaces, it's better to use fgets():

```
char str[100];
printf("Enter a sentence: ");
fgets(str, sizeof(str), stdin);
printf("You entered: %s\n", str);
```

Input and Output: formatted input

It's crucial to validate the input when using scanf to avoid unexpected behavior or errors. For example, you should check the return value to see if the input was successfully read, and you can use conditional statements to prompt the user for input again if needed.

There is a function sscanf that reads from a string instead of the standard input:

int sscanf(char *string, char *format, arg1, arg2, ...)

It scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

Input and Output: formatted input

In C++, the main tool for handling input is the **std::cin** object, which is part of the C++ Standard Library and defined in the <iostream> header.

1. **Basic Input with std::cin**: std::cin (C++ input) is used to read data from standard input (typically the keyboard). It uses the extraction operator **>>** to read values.

```
#include <iostream>

int main() {
    int x, y;
    std::cout << "Enter two numbers: ";
    std::cin >> x >> y;  // Reading two integers
    std::cout << "You entered: " << x << " and " << y << std::endl;
    return 0;
}
```

std::cin >> age; reads an integer input and stores it in the variable age. The extraction operator (>>) ignores leading whitespace and reads the input until the first whitespace (space, tab, or newline).

Input and Output: formatted input

2. **Multiple Inputs with std::cin**: You can read multiple values in a single line using std::cin:

```
#include <iostream>

int main() {
   int x, y;
   std::cout << "Enter two numbers: ";
   std::cin >> x >> y;  // Reading two integers
   std::cout << "You entered: " << x << " and " << y << std::endl;
   return 0;
}
```

3. **Reading Strings:** C++ offers several ways to handle string input.
   a. Using std::cin >>
   b. Using std::getline() for Full Line Input
      to handle strings with spaces (i.e., full sentences or multiple words), use std::getline()
      std::getline(std::cin, fullName); reads the entire line, including spaces, until a newline (\n) is encountered.

Input and Output: formatted input


```
#include <iostream>

int main() {
    std::string name;
    std::cout << "Enter your name: ";
    std::cin >> name;  // Reads a single word (stops at the first space)
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
#include <iostream>
#include <string>

int main() {
    std::string fullName;
    std::cout << "Enter your full name: ";
    std::getline(std::cin, fullName);  // Reads the entire line including spaces
    std::cout << "Hello, " << fullName << "!" << std::endl;
    return 0;
}
```

Input and Output: formatted input

4. **Handling Different Data Types Together**: You can read different types of data together by chaining the input with the >> operator:

```
#include <iostream>

int main() {
   int age;
   std::string name;
      std::cout << "Enter your age and name: ";
   std::cin >> age;       // Reading an integer
   std::cin.ignore();     // Ignore the newline character left in the input buffer
   std::getline(std::cin, name);  // Read the full name

   std::cout << "Age: " << age << ", Name: " << name << std::endl;
   return 0;
}
```

After reading age using std::cin >>, there may be a leftover newline character in the input buffer. To prevent std::getline() from reading an empty string, we use std::cin.ignore() to discard the newline character.

Input and Output: formatted input

5. **Input Validation and Error Handling**: C++ does not automatically validate input types. If the user enters incorrect data, std::cin can enter a fail state, meaning it stops functioning properly until the state is cleared.:

```
#include <iostream>

int main() {
    int number;
    std::cout << "Enter a number: ";

    while (!(std::cin >> number)) {  // Check if input is invalid
        std::cin.clear();  // Clear the error flag
        std::cin.ignore(1000, '\n');  // Discard invalid input
        std::cout << "Invalid input. Please enter a valid number: ";
    }

    std::cout << "You entered: " << number << std::endl;
    return 0;
}
```

- If std::cin receives non-numeric input (e.g., letters), it fails.
- std::cin.clear(); clears the error flag, and std::cin.ignore(1000, '\n'); discards the invalid input (up to 1000 characters or until a newline is found).
- This loop continues until the user provides valid input.

Input and Output: formatted input

6. **Using std::cin.ignore()**: std::cin.ignore() is used to skip or discard unwanted characters from the input buffer. It is useful when dealing with situations where leftover characters might interfere with the next input operation.

Syntax: *std::cin.ignore(n, delimiter);*
n: Number of characters to ignore (optional).
delimiter: Character that stops the ignoring process (optional, defaults to newline).

```
#include <iostream>
int main() {
    int age;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cin.ignore();  // Discards the newline character
    std::string name;
    std::cout << "Enter your name: ";
    std::getline(std::cin, name);  // Now it works correctly
    std::cout << "Age: " << age << ", Name: " << name << std::endl;
    return 0;
}
```

Input and Output: formatted input

7. **Reading Characters with std::cin.get()**: To read single characters, you can use the std::cin.get() function.

```
#include <iostream>

int main() {
    char ch;
    std::cout << "Enter a character: ";
    std::cin.get(ch);  // Reads a single character
    std::cout << "You entered: " << ch << std::endl;
    return 0;
}
```

Note that std::cin.get() can read whitespace characters (like spaces or newlines), unlike std::cin >>.

8. **Handling Input Using Streams**: C++ provides input streams for more complex input handling. These streams include **std::ifstream** to read data from files and **std::istringstream** to read from strings in memory.

Input and Output: formatted input


```
#include <iostream>
#include <sstream>

int main() {
    std::string input = "42 3.14 hello";
    std::istringstream iss(input);  // Stream that reads from the string

    int i;
    float f;
    std::string s;

    iss >> i >> f >> s;
    std::cout << "Integer: " << i << ", Float: " << f << ", String: " << s << std::endl;

    return 0;
}
```

File access

In C programming, file access functions are part of the standard C library (<stdio.h>) and are used to interact with files on a computer's file system. These functions allow you to perform various file-related operations, such as opening, reading, writing, closing, and manipulating files.

So far, we have seen how to read from the standard input and to write to the standard output, which are automatically defined for a program by the local operating system.

Now we are going to see how to access a file that is not already connected to the program. How can we arrange for the named files to be read - that is, how to connect the external names to the statements that read the data?

File access

The rules are simple. Before a file can be read or written, it has to be opened by the library function **fopen**. fopen takes an external filename, and after some negotiations with the operating system returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the file pointer, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. The definitions in <stdio.h> include a structure declaration called FILE.

File access: fopen

The only declarations needed for a file pointer are:

*FILE *fp;*

*FILE *fopen(char *name, char *mode);*

These statements mean that *fp* is a pointer to a *FILE*, and *fopen* returns a pointer to a *FILE*. *FILE* is a type name, like *int*, not a structure tag; therefore, it is defined with a typedef.

The call to fopen in a program is

fp = fopen(name, mode);

The first argument of fopen is a character string containing the **name of the file**. The second argument is the mode, also a character string, which indicates **how one intends to use the file**.

File access: fopen

Allowable modes include read ("**r**"), write ("***w***"), and append ("***a***"). Some systems distinguish between text and binary files; for the latter, a "***b***" must be appended to the mode string.

*FILE *file = fopen("example.txt", "r"); // Open for reading*

- If a file that does not exist is opened for writing or appending, it is created if possible.
- Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them.
- Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, *fopen* will return **NULL**.

File access: getc and putc

How to read or write the file once it is open?

getc returns the next character from a file; it needs the file pointer to tell it which file.

*int getc(FILE *fp)*

getc returns the next character from the stream referred to by fp; it returns EOF for end of file or error.

putc is an output function:

*int putc(int c, FILE *fp)*

putc writes the character c to the file fp and returns the character written, or EOF if an error occurs.

Like getchar and putchar, getc and putc may be macros instead of functions.

File access: fprintf

For formatted input or output of files, the functions fscanf and fprintf may be used. These are identical to scanf and printf, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.
The fprintf function allows you to write formatted data to a file, similar to the printf function for standard output. It takes a file stream and a format string, along with additional arguments for variable data.

Syntax:

*int fprintf(FILE *stream, const char *format, ...);*

Example:

fprintf(file, "This is an integer: %d\n", 42);

File access: fscanf

The fscanf function is used to read formatted data from a file, similar to the scanf function for standard input. It takes a file stream and a format string, along with pointers to variables where the data will be stored.

Syntax:

```
int fscanf(FILE *stream, const char *format, ...);
```

Example:

```
int number;
fscanf(file, "This is an integer: %d", &number);
```

File access: fread

The *fread* function is used to read data from a file into a specified buffer. You provide the buffer, the size of each element to be read, the number of elements to be read, and the file stream.

Syntax:

*size_t fread(void *ptr, size_t size, size_t count, FILE *stream);*

Example:

```
char buffer[100];
fread(buffer, sizeof(char), 100, file);
```

File access: fwrite

The *fwrite* function is used to write data from a buffer to a file. You provide the buffer, the size of each element to be written, the number of elements to be written, and the file stream.

Syntax:

*size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);*

Example:

```
char data[] = "This is a sample text.";
fwrite(data, sizeof(char), strlen(data), file);
```

File access: fseek and ftell

fseek allows you to set the file position indicator within a file stream. *ftell* is used to determine the current file position.

Syntax:

*int fseek(FILE *stream, long int offset, int origin); (for fseek)*

Syntax:

*long int ftell(FILE *stream); (for ftell)*

Example:

```
fseek(file, 0, SEEK_SET); // Move to the beginning of the file
long position = ftell(file); // Get the current position
```

File access: rewind and feof

The *rewind* function is used to reset the file position indicator to the beginning of the file.

Syntax:

```
void rewind(FILE *stream);
```

Example:

```
rewind(file);
```

The *feof* function is used to check if the end-of-file has been reached in a file stream. It returns a non-zero value if the end-of-file indicator is set.

Syntax:

```
int feof(FILE *stream);
```

Example:

```
if (feof(file)) {
    // End of file reached
}
```

File access: fclose

The *fclose* function is used to close an open file, releasing any resources associated with it. It should be called when you're done working with a file to ensure proper file management. *fclose* flushes the stream pointed to by stream (writing any buffered output data) and closes the underlying file descriptor, freeing the file pointer for another file. *fclose* is called automatically for each open file when a program terminates normally.

Syntax:

*int fclose(FILE *stream);*

Upon successful completion, 0 is returned. Otherwise, EOF is returned. In either case, any further access (including another call to fclose()) to the stream results in undefined behavior.

Example:

fclose(file);

File access: remove and rename

The *remove* function is used to delete a file, and the *rename* function is used to change the name of a file.

Syntax:

*int remove(const char *filename); (for remove)*

Syntax:

*int rename(const char *old_filename, const char *new_filename); (for rename)*

Example:

```
remove("oldfile.txt");
rename("newfile.txt", "renamedfile.txt");
```

File access: stdout, stdin, stderr

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them.
These files are

- the standard input
- the standard output
- the standard error;

the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>. Normally stdin is connected to the keyboard and stdout and stderr are connected to the screen, but stdin and stdout may be redirected to files or pipes.

File access: example


```
#include<stdio.h>
#include<conio.h>
main()
{
      FILE *fp;
      char ch;
      fp = fopen("hello.txt", "w");
      printf("Inserisci il dato: ");
      while( (ch = getchar()) != EOF) {
        putc(ch,fp);
      }
      fclose(fp);
      fp = fopen("hello.txt", "r");

      while( (ch = getc(fp)! = EOF)
        printf("%c",ch);

      fclose(fp);
}
```

File access: example


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

int main() {
   // create file pointers.
   FILE *names = fopen("names.txt", "r");
   FILE *greet = fopen("greet.txt", "w");

   // check if all is fine
   if (!names || !greet) {
      fprintf(stderr, "Apertura del file fallita!\n");
      return EXIT_FAILURE;
   }

   // Time of greetings
   char nome[20];
   // keep on reading until ....
   while (fscanf(names, "%s\n", nome) > 0) {
      fprintf(greet, "Ciao, %s!\n", nome);
   }

   // When the feof is reached prints a message on stdout
   if (feof(names)) {
      printf("The greetings are over!\n");
   }

   return EXIT_SUCCESS;
}
```

File access

In C++ programming, reading from and writing to files is done using the file stream classes provided by the <fstream> library, which is part of the C++ Standard Library.

The three main classes used for file handling are:

- std::ifstream (input file stream): For reading data from files.
- std::ofstream (output file stream): For writing data to files.
- std::fstream (file stream): For both reading from and writing to files.

These classes operate similarly to the std::cin and std::cout streams that handle input from and output to the console, but they are used to manage file I/O instead.

File access

To work with file streams, you must include the <fstream> header

#include <fstream>

File streams (ifstream, ofstream, fstream) need to be opened before they can be used. You can open a file using the constructor or the .open() method.

```
std::ifstream inputFile;
inputFile.open("filename.txt");  // Open file for reading

std::ofstream outputFile;
outputFile.open("filename.txt"); // Open file for writing
```

File access

You can read data from a file using the std::ifstream class. The common input methods are:

- Using the extraction operator >> for formatted input (e.g., reading numbers, strings, etc.).
- Using std::getline() to read entire lines of text.

You can write data to a file using the std::ofstream class and the insertion operator (<<), just like std::cout.

- The << operator is used to write data to the file, just like it’s used with std::cout to write data to the console.
- Make sure to call .close() when you're done writing to the file.

File access

- To append data to an existing file instead of overwriting it, open the file in append mode using the std::ios::app flag.
- std::fstream can be used to both read from and write to a file. It needs to be opened in a specific mode to allow for both reading and writing.

When opening files, you can specify different modes by combining flags from the std::ios namespace.

Mode	Description
std::ios::in	Open the file for reading (default for ifstream)
std::ios::out	Open the file for writing (default for ofstream)
std::ios::app	Append to the end of the file
std::ios::ate	Open the file and move the cursor to the end (can write anywhere)
std::ios::trunc	Truncate the file (remove its contents) if it exists
std::ios::binary	Open the file in binary mode (for non-text files)

File access: reading


```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream inputFile("input.txt");  // Open the file for reading

    if (!inputFile) {  // Check if the file opened successfully
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }

    std::string line;
    while (std::getline(inputFile, line)) {  // Read the file line by line
        std::cout << line << std::endl;  // Print each line
    }

    inputFile.close();  // Close the file
    return 0;
}
```

File access: reading


```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream inputFile("data.txt");  // Open file

    if (!inputFile) {
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }

    int number;
    while (inputFile >> number) {  // Read integers from the file
        std::cout << number << std::endl;  // Output each integer
    }

    inputFile.close();  // Close the file
    return 0;
}
```

File access: writing


```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outputFile("output.txt");  // Open file for writing

    if (!outputFile) {
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }

    outputFile << "Hello, file!" << std::endl;
    outputFile << "This is some text written to the file." << std::endl;

    outputFile.close();  // Close the file
    return 0;
}
```

File access: appending


```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outputFile("output.txt", std::ios::app);  // Open file in append mode

    if (!outputFile) {
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }

    outputFile << "This text will be appended to the file." << std::endl;

    outputFile.close();  // Close the file
    return 0;
}
```

File access: reading and writing


```
#include <iostream>
#include <fstream>

int main() {
    std::fstream file("data.txt", std::ios::in | std::ios::out);  // Open file for both reading and writing

    if (!file) {
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }
    std::string line;
    while (std::getline(file, line)) {  // Read each line
        std::cout << line << std::endl;  // Output it to the console
    }

    file.clear();  // Clear the end-of-file flag to perform writing after reading
    file << "\nNew line added!" << std::endl;  // Write to the file

    file.close();  // Close the file
    return 0;
}
```

File access

- It’s essential to check if a file has been successfully opened, and to handle potential I/O errors.

- You can manually move the file pointer (seek) to different positions in a file using functions like .seekg() (for input files) and .seekp() (for output files). This is useful for random file access.

- For non-text (binary) data, such as images or serialized objects, you can use binary mode by adding the std::ios::binary flag. Instead of using formatted input/output operations, you can use the .read() and .write() functions.

File access: checking for errors


```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream inputFile("nonexistent.txt");

    if (!inputFile) {  // Check if the file opened successfully
        std::cerr << "Error: File could not be opened!" << std::endl;
        return 1;
    }

    // Further processing...

    inputFile.close();
    return 0;
}
```

File access: file pointer operations


```
#include <iostream>
#include <fstream>

int main() {
    std::fstream file("data.txt", std::ios::in | std::ios::out);

    if (!file) {
        std::cerr << "Error opening file!" << std::endl;
        return 1;
    }

    file.seekg(10);  // Move the input pointer to the 10th byte
    std::string data;
    file >> data;  // Read from the new position
    std::cout << "Data from position 10: " << data << std::endl;

    file.seekp(0, std::ios::end);  // Move the output pointer to the end of the file
    file << "\nAppended text";     // Append data at the end
    file.close();
    return 0;
}
```

File access: read/write binary files


```
#include <iostream>
#include <fstream>
int main() {
    int numbers[5] = {1, 2, 3, 4, 5};

    // Writing binary data
    std::ofstream outputFile("data.bin", std::ios::binary);
    outputFile.write(reinterpret_cast<char*>(numbers), sizeof(numbers));
    outputFile.close();
    // Reading binary data
    int readNumbers[5];
    std::ifstream inputFile("data.bin", std::ios::binary);
    inputFile.read(reinterpret_cast<char*>(readNumbers), sizeof(readNumbers));
    inputFile.close();

    for (int n : readNumbers) {
        std::cout << n << " ";
    }

    return 0;
}
```