*main.cpp [files] - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Debug

<global> main() : int

Management

Projects Symbols Files

Workspace
files
Sources
main.cpp

*main.cpp

```
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main()
7  {
8      ofstream fail;
9      fail.open("hello2.txt", ios::app);//will create file if none is found
10     fail << "c++"<<endl;
11     fail.close();
12
13     char data[100];
14     ifstream fail2;
15     fail2.open("hello2.txt");
16     fail2>>data;//will read the first line
17     cout<<data;
18
19     fail.close();
20     //remmber #include <fstream>
21     return 0;
22 }
23
```

Logs & others

Code::Blocks | Build messages | Cscope | Debugger | Build log | Closed files list | Fortran info

```
-------------- Run: Debug in files (compiler: GNU GCC Compiler)---------------
Checking for existence: C:\Users\Martin Iliev\Desktop\beginning\files\bin\Debug\files.exe
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Users\Martin Iliev\Desktop\beginning\files\bin\Debug\files.exe"  (in C:\Users
\Martin Iliev\Desktop\beginning\files\.)
Process terminated with status 0 (0 minute(s), 1 second(s))
```

C:\Users\Martin Iliev\Desktop\beginning\files\main.cpp | Windows (CR+LF) | WINDOWS-1252 | Line 13, Column 20 | Insert | Modified | Read/Write | default

ENG 8:10 AM 7/11/2018

Section 6.1

Input/Output with files

C++ has support both for input and output with files through the following classes:

- `ofstream`: File class for writing operations (derived from `ostream`)
- `ifstream`: File class for reading operations (derived from `istream`)
- `fstream`: File class for both reading and writing operations (derived from `iostream`)

Open a file

The first operation generally done on an object of one of these classes is to associate it to a real file, that is to say, to open a file. The open file is represented within the program by a stream object (an instantiation of one of these classes) and any input or output performed on this stream object will be applied to the physical file.

In order to open a file with a stream object we use its member function `open()`:

```
void open (const char * filename, openmode mode);
```

where *filename* is a string of characters representing the name of the file to be opened and *mode* is a combination of the following flags:

`ios::in`	Open file for reading
`ios::out`	Open file for writing
`ios::ate`	Initial position: end of file
`ios::app`	Every output is appended at the end of file
`ios::trunc`	If the file already existed it is erased
`ios::binary`	Binary mode

These flags can be combined using bitwise operator OR: `|`. For example, if we want to open the file "example.bin" in binary mode to add data we could do it by the following call to function-member `open`:

```
ofstream file;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

All of the member functions `open` of classes `ofstream`, `ifstream` and `fstream` include a default mode when opening files that varies from one to the other:

class	default *mode* to parameter
`ofstream`	`ios::out | ios::trunc`
`ifstream`	`ios::in`
`fstream`	`ios::in | ios::out`

The default value is only applied if the function is called without specifying a *mode* parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined.

Since the first task that is performed on an object of classes `ofstream`, `ifstream` and `fstream` is frequently to open a file, these three classes include a constructor that directly calls the `open` member function and has the same parameters as this. This way, we could also have declared the previous object and conducted the same opening operation just by writing:

```
ofstream file ("example.bin", ios::out | ios::app | ios::binary);
```

Both forms to open a file are valid.

You can check if a file has been correctly opened by calling the member function `is_open()`:

```
bool is_open();
```

that returns a `bool` type value indicating `true` in case that indeed the object has been correctly associated with an open file or `false` otherwise.

Closing a file

When reading, writing or consulting operations on a file are complete we must close it so that it becomes available again. In order to do that we shall call the member function `close()`, that is in charge of flushing the buffers and closing the file. Its form is quite simple:

```
void close ();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close`.

Text mode files

Classes `ofstream`, `ifstream` and `fstream` are derived from `ostream`, `istream` and `iostream` respectively. That's why *fstream* objects can use the members of these parent classes to access data.

Generally, when using text files we shall use the same members of these classes that we used in communication with the console (`cin` and `cout`). As in the following example, where we use the overloaded insertion operator `<<`:

```
// writing on a text file
#include <fstream.h>

int main () {
  ofstream examplefile ("example.txt");
  if (examplefile.is_open())
  {
    examplefile << "This is a line.\n";
    examplefile << "This is another line.\n";
    examplefile.close();
  }
  return 0;
```

file example.txt

```
This is a line.
This is another line.
```

```
  }
  return 0;
}
```

Data input from file can also be performed in the same way that we did with `cin`:

```
// reading a text file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main () {
  char buffer[256];
  ifstream examplefile ("example.txt");
  if (! examplefile.is_open())
  { cout << "Error opening file"; exit (1); }

  while (! examplefile.eof() )
  {
    examplefile.getline (buffer,100);
    cout << buffer << endl;
  }
  return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `eof` that `ifstream` inherits from class `ios` and that returns `true` in case that the end of the file has been reached.

Verification of state flags

In addition to `eof()`, other member functions exist to verify the state of the stream (all of them return a `bool` value):

`bad()`
: Returns `true` if a failure occurs in a reading or writing operation. For example in case we try to write to a file that is not open for writing or if the device where we try to write has no space left.

`fail()`
: Returns `true` in the same cases as `bad()` plus in case that a format error happens, as trying to read an integer number and an alphabetical character is received.

`eof()`
: Returns `true` if a file opened for reading has reached the end.

`good()`
: It is the most generic: returns `false` in the same cases in which calling any of the previous functions would return `true`.

In order to reset the state flags checked by the previous member functions you can use member function `clear()`, with no parameters.

get and *put* stream pointers

All i/o streams objects have, at least, one stream pointer:

- `ifstream`, like `istream`, has a pointer known as *get pointer* that points to the next element to be read.
- `ofstream`, like `ostream`, has a pointer *put pointer* that points to the location where the next element has to be written.
- Finally `fstream`, like `iostream`, inherits both: *get* and *put*

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

`tellg()` and `tellp()`

These two member functions admit no parameters and return a value of type `pos_type` (according ANSI-C++ standard) that is an integer data type representing the current position of *get* stream pointer (in case of `tellg`) or *put* stream pointer (in case of `tellp`).

`seekg()` and `seekp()`

This pair of functions serve respectively to change the position of stream pointers *get* and *put*. Both functions are overloaded with two different prototypes:

```
seekg ( pos_type position );
seekp ( pos_type position );
```

Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions `tellg` and `tellp`.

```
seekg ( off_type offset, seekdir direction );
seekp ( off_type offset, seekdir direction );
```

Using this prototype, an offset from a concrete point determined by parameter *direction* can be specified. It can be:

`ios::beg`	offset specified from the beginning of the stream
`ios::cur`	offset specified from the current position of the stream pointer
`ios::end`	offset specified from the end of the stream

The values of both stream pointers *get* and *put* are counted in different ways for text files than for binary files, since in text mode files some modifications to the appearance of some special characters can occur. For that reason it is advisable to use only the first prototype of `seekg` and `seekp` with files opened in text mode and always use non-modified values returned by `tellg` or `tellp`. With binary files, you can freely use all the implementations for these functions. They should not have any unexpected behavior.

The following example uses the member functions just seen to obtain the size of a binary file:

```
// obtaining file size
#include <iostream.h>
#include <fstream.h>

const char * filename = "example.txt";

int main () {
  long l,m;
  ifstream file (filename, ios::in|ios::binary);
  l = file.tellg();
  file.seekg (0, ios::end);
  m = file.tellg();
  file.close();
```

```
size of example.txt is 40 bytes.
```

```
  m = file.tellg();
  file.close();
  cout << "size of " << filename;
  cout << " is " << (m-l) << " bytes.\n";
  return 0;
}
```

Binary files

In binary files inputting and outputting data with operators like << and >> and functions like `getline`, does not make too much sense, although they are perfectly valid.

File streams include two member functions specially designed for input and output of data sequentially: **write** and **read**. The first one (**write**) is a member function of `ostream`, also inherited by `ofstream`. And **read** is member function of `istream` and it is inherited by `ifstream`. Objects of class `fstream` have both. Their prototypes are:

```
write ( char * buffer, streamsize size );
 read ( char * buffer, streamsize size );
```

Where *buffer* is the address of a memory block where the read data are stored or from where the data to be written are taken. The *size* parameter is an integer value that specifies the number of characters to be read/written from/to *the buffer*.

```
// reading binary file
#include <iostream.h>
#include <fstream.h>

const char * filename = "example.txt";

int main () {
  char * buffer;
  long size;
  ifstream file (filename, ios::in|ios::binary|ios::ate);
  size = file.tellg();
  file.seekg (0, ios::beg);
  buffer = new char [size];
  file.read (buffer, size);
  file.close();

  cout << "the complete file is in a buffer";

  delete[] buffer;
  return 0;
}
```

```
the complete file is in a buffer
```

Buffers and Synchronization

When we operate with file streams, these are associated to a *buffer* of type `streambuf`. This *buffer* is a memory block that acts as an intermediary between the stream and the physical file. For example, with an out stream, each time the member function `put` (write a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in the *buffer* for that stream.

```
read ( char * buffer, streamsize size );
```

Where *buffer* is the address of a memory block where the read data are stored or from where the data to be written are taken. The *size* parameter is an integer value that specifies the number of characters to be read/written from/to *the buffer*.

```
// reading binary file
#include <iostream.h>
#include <fstream.h>

const char * filename = "example.txt";

int main () {
  char * buffer;
  long size;
  ifstream file (filename, ios::in|ios::binary|ios::ate);
  size = file.tellg();
  file.seekg (0, ios::beg);
  buffer = new char [size];
  file.read (buffer, size);
  file.close();

  cout << "the complete file is in a buffer";

  delete[] buffer;
  return 0;
}
```

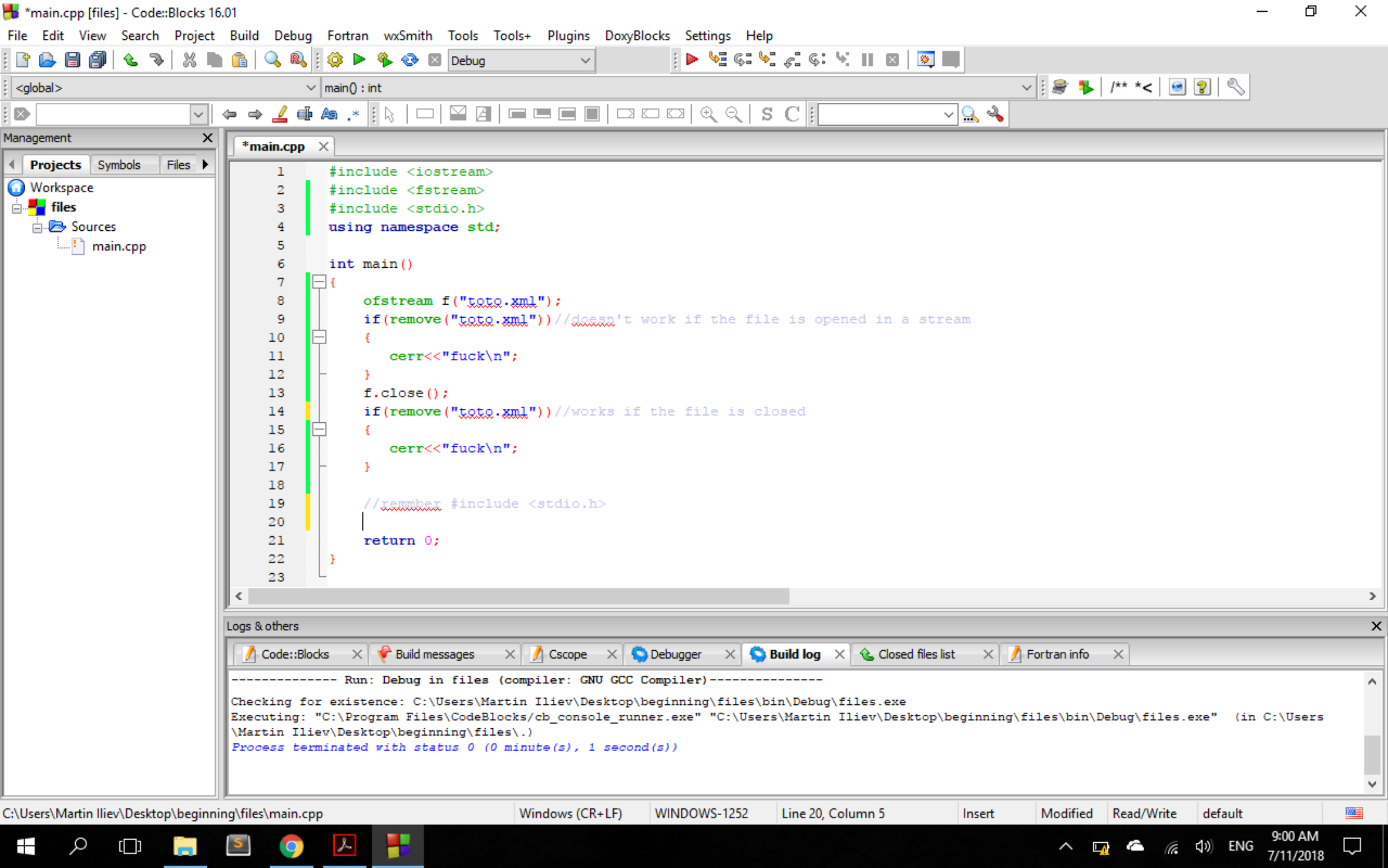
```
the complete file is in a buffer
```

Buffers and Synchronization

When we operate with file streams, these are associated to a *buffer* of type `streambuf`. This *buffer* is a memory block that acts as an intermediary between the stream and the physical file. For example, with an out stream, each time the member function `put` (write a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in the *buffer* for that stream.

When the buffer is flushed, all data that it contains is written to the physic media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:

- **When the file is closed:** before closing a file all buffers that have not yet been completely written or read are synchronized.
- **When the *buffer* is full:** *Buffers* have a certain size. When the *buffer* is full it is automatically synchronized.
- **Explicitly with manipulators:** When certain manipulators are used on streams a synchronization takes place. These manipulators are: `flush` and `endl`.
- **Explicitly with function `sync()`:** Calling member function `sync()` (no parameters) causes an immediate syncronization. This function returns an `int` value equal to `-1` if the stream has no associated *buffer* or in case of failure.


*main.cpp [files] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
main() : int
Management
Projects Symbols Files
Workspace
files
Sources
main.cpp
*main.cpp
1 #include <iostream>
2 #include <fstream>
3 #include <stdio.h>
4 using namespace std;
5
6 int main()
7 {
8 ofstream f("toto.xml");
9 if(remove("toto.xml"))//doesn't work if the file is opened in a stream
10 {
11 cerr<<"fuck\n";
12 }
13 f.close();
14 if(remove("toto.xml"))//works if the file is closed
15 {
16 cerr<<"fuck\n";
17 }
18
19 //remmber #include <stdio.h>
20
21 return 0;
22 }
23
Logs & others
Code::Blocks
Build messages
Cscope
Debugger
Build log
Closed files list
Fortran info
---------- Run: Debug in files (compiler: GNU GCC Compiler)----------
Checking for existence: C:\Users\Martin Iliev\Desktop\beginning\files\bin\Debug\files.exe
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Users\Martin Iliev\Desktop\beginning\files\bin\Debug\files.exe" (in C:\Users\Martin Iliev\Desktop\beginning\files\.)
Process terminated with status 0 (0 minute(s), 1 second(s))
C:\Users\Martin Iliev\Desktop\beginning\files\main.cpp
Windows (CR+LF)
WINDOWS-1252
Line 20, Column 5
Insert
Modified
Read/Write
default
ENG
9:00 AM
7/11/2018