http://uranchimeg.com/

Object oriented programming with C++

T.Uranchimeg
Prof. Dr.

Email
uranchimeg@must.edu.mn

Power Engineering School

Subjects

- *Preprocessor directives*
- *#define*
- *#undef*
- *#ifdef, #ifndef, #if, #endif, #else and #elif*
- *Input/Output with files*

Preprocessor directives

- Preprocessor directives are orders that we include within the code of our programs that are not instructions for the program itself but for the preprocessor. The preprocessor is executed automatically by the compiler when we compile a program in C++ and is in charge of making the first verifications and digestions of the program's code.
- All these directives must be specified in a single line of code and they do not have to include an ending semicolon **;**.

#define

- At the beginning of this tutorial we have already spoken about a preprocessor directive: **#define**, that serves to generate what we called *defined constantants* or *macros* and whose form is the following:
- **#define** *name value*

Continue

- Its function is to define a macro called *name* that whenever it is found in some point of the code is replaced by *value*. For example:
- #define MAX_WIDTH 100
  char str1[MAX_WIDTH];
  char str2[MAX_WIDTH];

Continue

- It defines two strings to store up to 100 characters.
- #define can also be used to generate macro functions:
- #define getmax(a,b) a>b?a:b
  int x=5, y;
  y = getmax(x,2);
- after the execution of this code **y** would contain 5.

#undef

- **#undef** fulfills the inverse functionality of **#define**. It eliminates from the list of defined constants the one that has the name passed as a parameter to **#undef**:
- #define MAX_WIDTH 100
  char str1[MAX_WIDTH];
  #undef MAX_WIDTH
  #define MAX_WIDTH 200
  char str2[MAX_WIDTH];

#ifdef, #ifndef, #if, #endif, #else and #elif

- These directives allow to discard part of the code of a program if a certain condition is not fulfilled.
- **#ifdef** allows that a section of a program is compiled only if the *defined constant* that is specified as the parameter has been defined, independently of its value. Its operation is:
- **#ifdef** *name*
  // code here
  #endif

Example

- #ifdef MAX_WIDTH
  char str[MAX_WIDTH];
  #endif
- In this case, the line **char str[MAX_WIDTH];** is only considered by the compiler if the *defined constant* **MAX_WIDTH** has been previously defined, independently of its value. If it has not been defined, that line will not be included in the program.

Continue

- **#ifndef** serves for the opposite: the code between the **#ifndef** directive and the **#endif** directive is only compiled if the constant name that is specified has not been defined previously. For example:
- #ifndef MAX_WIDTH
  #define MAX_WIDTH 100
  #endif
  char str[MAX_WIDTH];

Continue

- In this case, if when arriving at this piece of code the *defined constant* **MAX_WIDTH** has not yet been defined it would be defined with a value of 100. If it already existed it would maintain the value that it had (because the #define statement won't be executed).

Continue

The **#if**, **#else** and **#elif** (*elif* = *else if*) directives serve so that the portion of code that follows is compiled only if the specified condition is met. The condition can only serve to evaluate constant expressions.

Example

```
#if MAX_WIDTH>200
#undef MAX_WIDTH
#define MAX_WIDTH 200

#elsif MAX_WIDTH<50
#undef MAX_WIDTH
#define MAX_WIDTH 50

#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif

char str[MAX_WIDTH];
```

#line

- When we compile a program and errors happen during the compiling process, the compiler shows the error that happened preceded by the name of the file and the line within the file where it has taken place.
- The **#line** directive allows us to control both things, the line numbers within the code files as well as the file name that we want to appear when an error takes place. Its form is the following one:
- **#line** *number* "*filename*"

Continue

- Where *number* is the new line number that will be assigned to the next code line. The line number of successive lines will be increased one by one from this.
- *filename* is an optional parameter that serves to replace the file name that will be shown in case of error from this directive until another one changes it again or the end of the file is reached. For example:
- #line 1 "assigning variable"
  int a?;
- This code will generate an error that will be shown as error in file "assigning variable", line 1.

#error

- This directive aborts the compilation process when it is found returning the error that is specified as the parameter:
- #ifndef __cplusplus
  #error A C++ compiler is required
  #endif
- This example aborts the compilation process if the *defined constant* **__cplusplus** is not defined.

#include

- This directive has also been used assiduously in other sections of this tutorial. When the preprocessor finds an **#include** directive it replaces it by the whole content of the specified file. There are two ways to specify a file to be included:
- **#include** "*file*"
  #include <*file*>

Continue

- The only difference between both expressions is the directories in which the compiler is going to look for the file. In the first case where the file is specified between quotes, the file is looked for in the same directory that includes the file containing the directive. In case that it is not there, the compiler looks for the file in the default directories where it is configured to look for the standard header files.
- If the file name is enclosed between angle-brackets <> the file is looked for directly where the compiler is configured to look for the standard header files.

#pragma

- This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with **#pragma**.

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.

Continue

- 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:

Continue

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

Continue

- 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:

Continue

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

Continue

- 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.

Continue

- 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

Continue

- **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 ();**

Continue

- 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 <<:

Continue

```
// 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.
```

Continue

- 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>
```

Continue

```
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.
```

Continue

- 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.

Continue

- **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.

Continue

- **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*

Continue

- 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).

Continue

- **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**.

Continue

- **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:

Continue

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.

Continue

- 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:

Continue

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

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

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.

Continue

- 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*.

Continue

```
// 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);
```

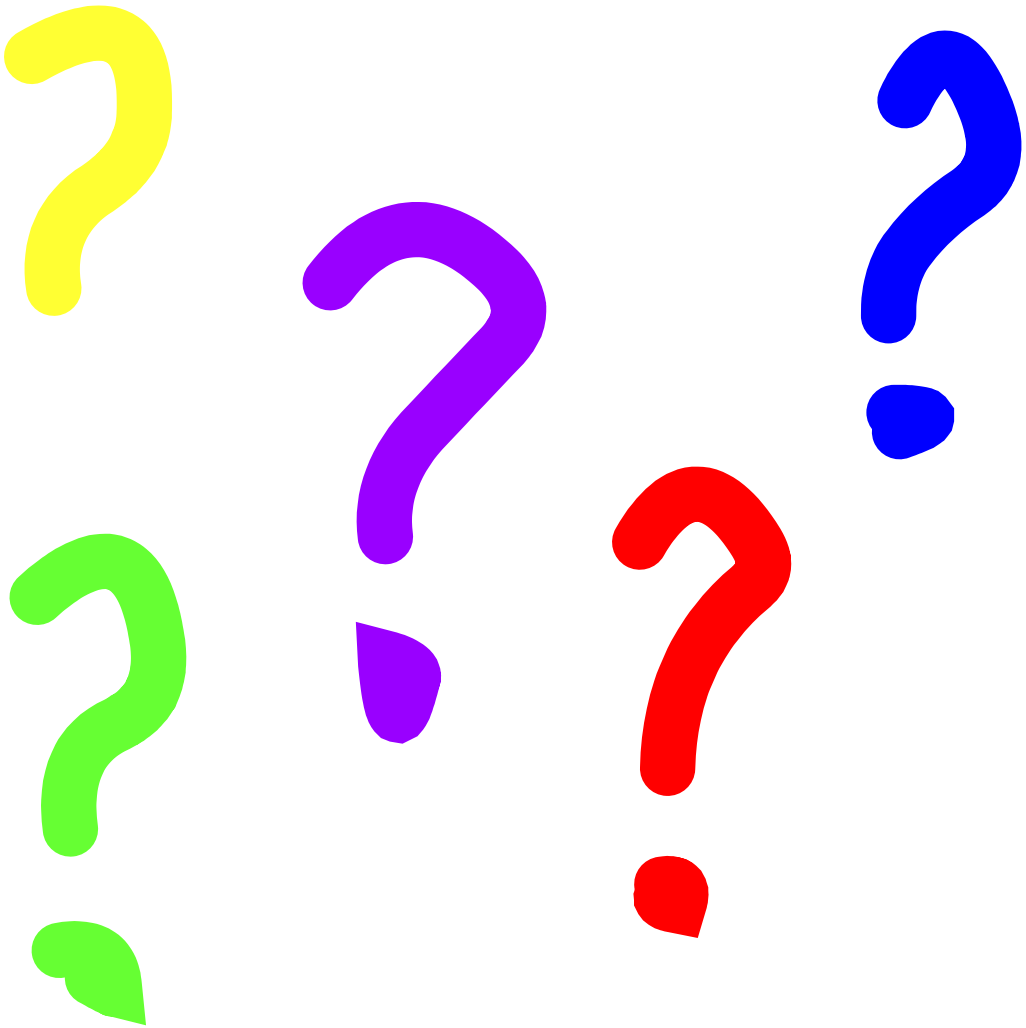
Continue

```
 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
```

Have you questions?



Summary

- *Preprocessor directives*
- *#define*
- *#undef*
- *#ifdef, #ifndef, #if, #endif, #else and #elif*
- *Input/Output with files*

End of…

Thank you for

ATTENTION