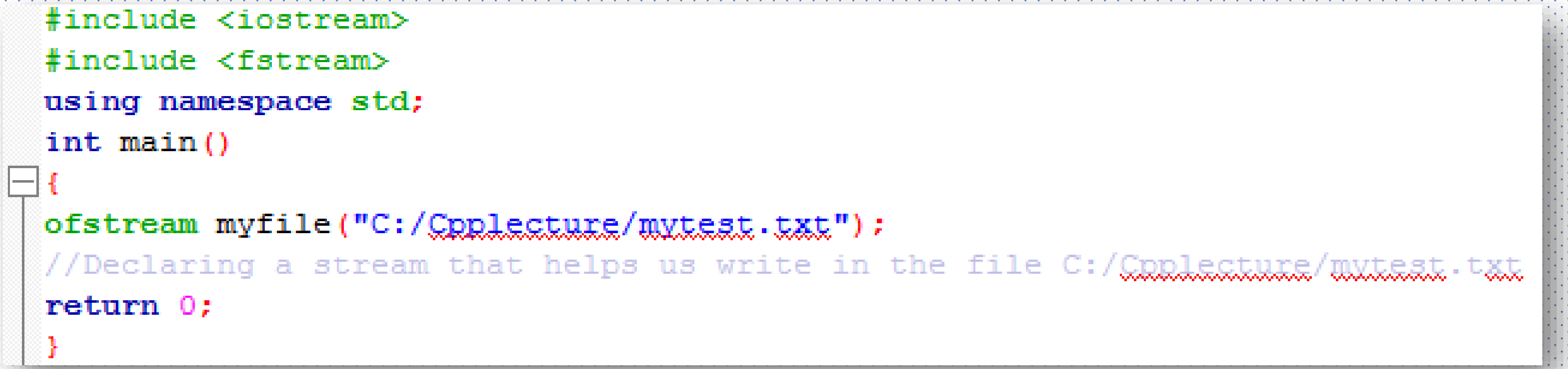
Reading and modifying files

Files

- So far, we have been writing console programs: taking in what the user typed and displaying a result in the console. This was relatively easy.
- I hope we agree that this is not enough. e.g. think of softwares like bloc-note, your IDE, …: these are programs that can read files and write in them.
- This point is very essential practically, e.g. in video games: we need backup files, game images, the kinematics, the musics, etc.
- To conclude, if your program can not deal with files, then it will be very limited.

Files: Writing in files

- In the case of files, we should insert **#include <fstream>.**
- Just as iostream means input/output stream, fstream means file stream.
- The first thing we should start with is **opening and/or creating** the file in the writing mode.
- The stream is declared just like a variable but while taking as type **ofstream** and as value the **file path**.



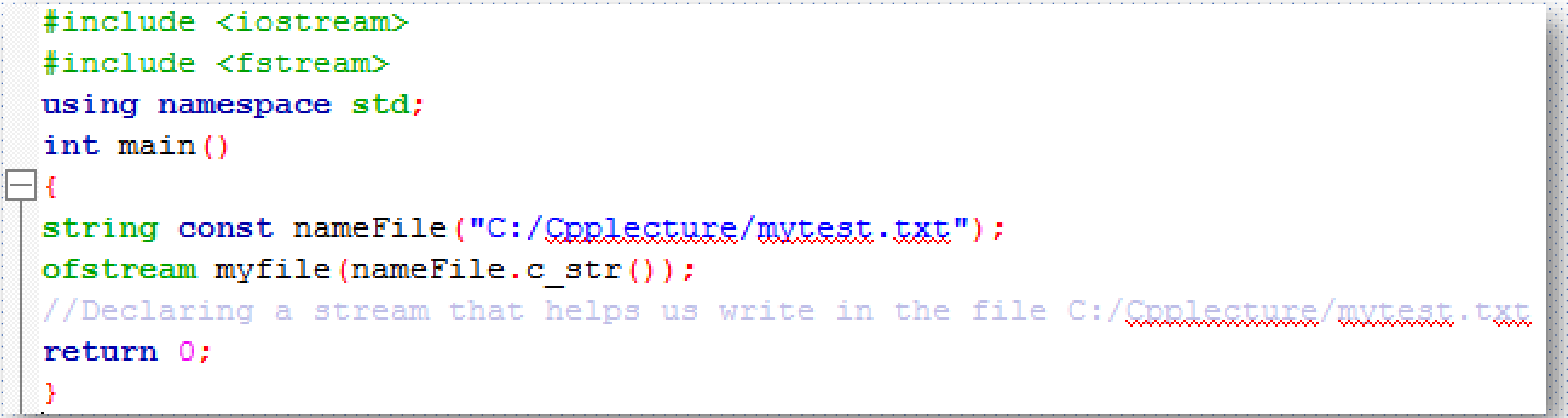
```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream myfile("C:/Cpplecture/mytest.txt");
//Declaring a stream that helps us write in the file C:/Cpplecture/mytest.txt
return 0;
}
```

If the file doesn't exist, our program will create one automatically

- The path can be written in two other ways:
  - Absolute path, i.e. the placement of the file from the drive: **C:/Cpplecture/C++/Files/mytest.txt.**
  - A relative path, i.e. placement of the file from where our program is situated in the drive: **Files/mytest.txt.** If my program is located at **C:/Cpplecture/C++/.**

Files: Writing in files

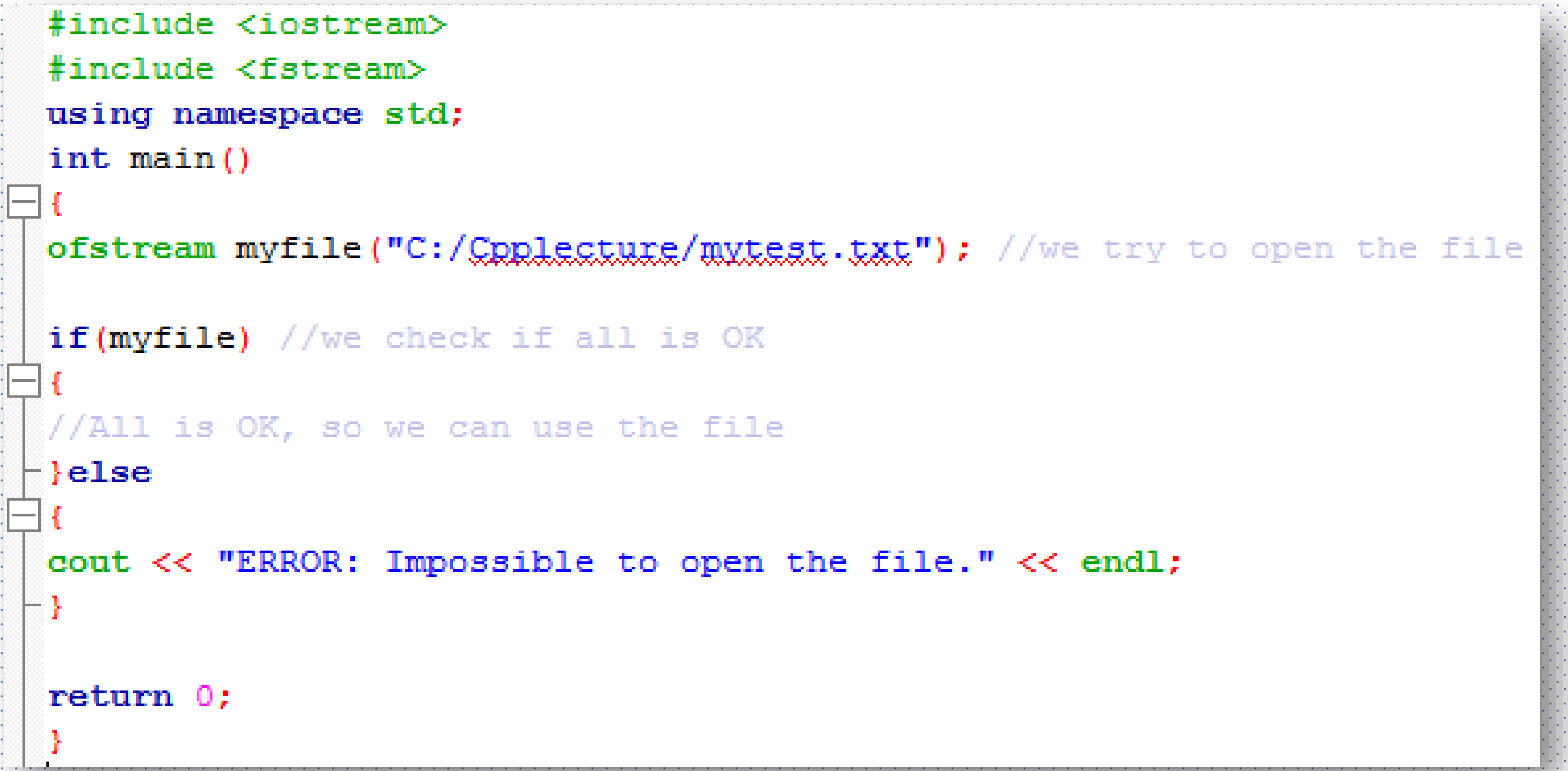
- In most cases, the name of the file is contained in a string. In this case, we should use the function **`c_str()`** when opening the file.



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
string const nameFile("C:/Cpplecture/mytest.txt");
ofstream myfile(nameFile.c_str());
//Declaring a stream that helps us write in the file C:/Cpplecture/mytest.txt
return 0;
}
```

- Problems may arise when opening the file, e.g. if the file doesn't belong to you, or if the hard drive is full, etc. That is why we should always test to check if everything went on well.
- To do so, we use the syntax: **`if (myfile).`** If this test is false, then there is a problem and we can not use the file.

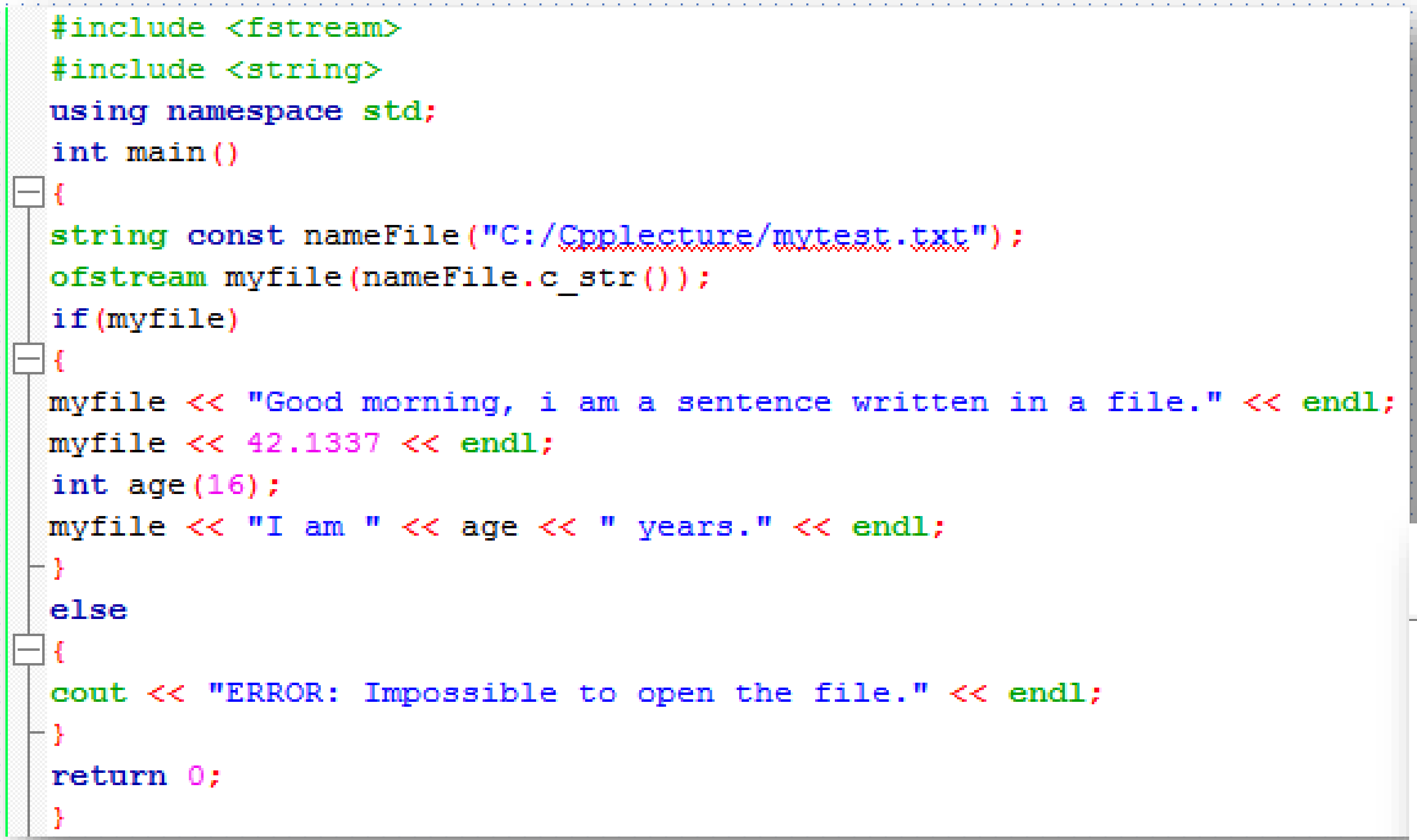
Files: Writing in files



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream myfile("C:/Cpplecture/mytest.txt"); //we try to open the file

if(myfile) //we check if all is OK
{
//All is OK, so we can use the file
}else
{
cout << "ERROR: Impossible to open the file." << endl;
}

return 0;
}
```

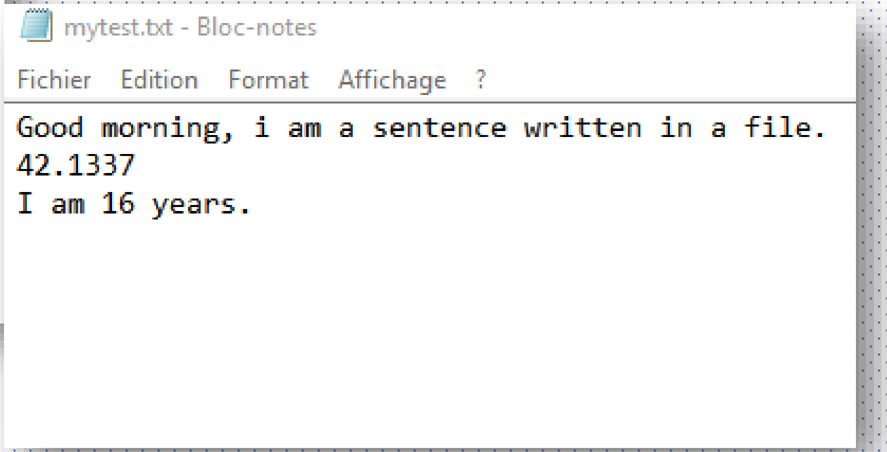
Files: Writing in files

- Now, I believe we are ready to start the writing propely.
- Everything is just like **`cout <<`**



```cpp
#include <fstream>
#include <string>
using namespace std;
int main()
{
string const nameFile("C:/Cpplecture/mytest.txt");
ofstream myfile(nameFile.c_str());
if(myfile)
{
myfile << "Good morning, i am a sentence written in a file." << endl;
myfile << 42.1337 << endl;
int age(16);
myfile << "I am " << age << " years." << endl;
}
else
{
cout << "ERROR: Impossible to open the file." << endl;
}
return 0;
}
```

Exercise: Write a program that asks a user his name and his age and that inserts this information in a file.



```
Good morning, i am a sentence written in a file.
42.1337
I am 16 years.
```

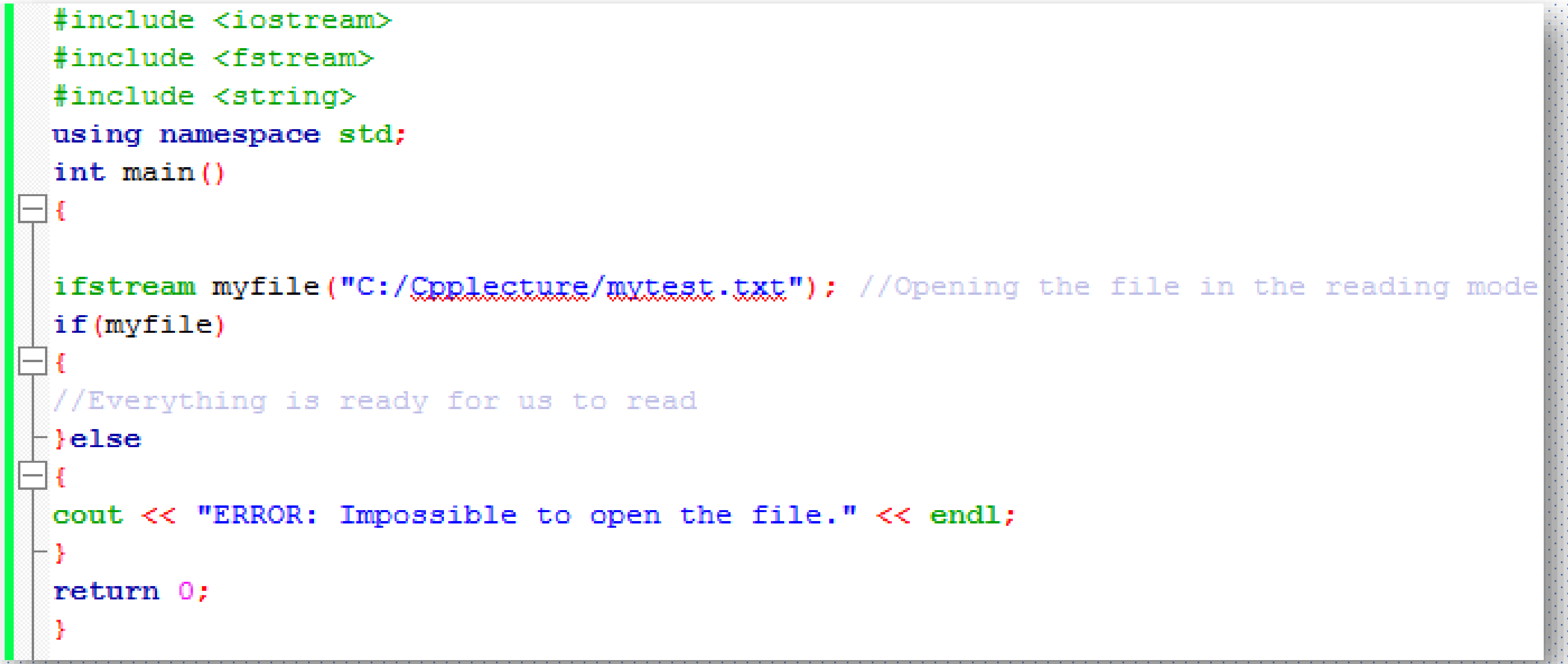
Files: Writing in files

- Remark: When inserting the information in the file, it overwrites what was existing.
- **If we instead wish to insert the new data at the end, when declaring the file, we should use the syntax:** `ofstream myfile("C:/Cpplecture/mytest.txt", ios::app);`.
- "app" here is to mean "**append**".

Exercise: Write a program that asks a user his name and his age and that inserts this information in a file. The program should do this for 5 individuals and it should append their informations.

Files: Reading files

- As usual, we should first start by opening the file. The procedure is the same as with writing but we instead use **ifstream** rather than **ofstream**. We should equally check the opening of the file in order to avoid errors.

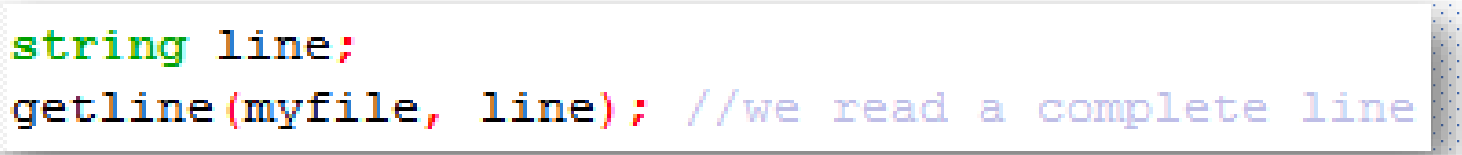


```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{

ifstream myfile("C:/Cpplecture/mytest.txt"); //Opening the file in the reading mode
if(myfile)
{
//Everything is ready for us to read
}else
{
cout << "ERROR: Impossible to open the file." << endl;
}
return 0;
}
```

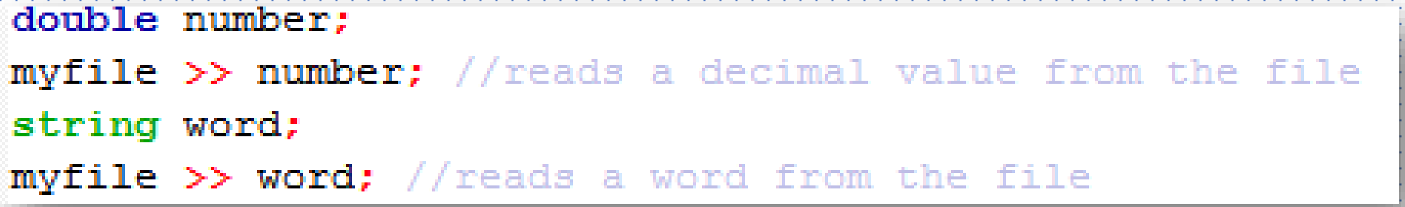
Files: Reading files

There are 3 different ways to read a file:

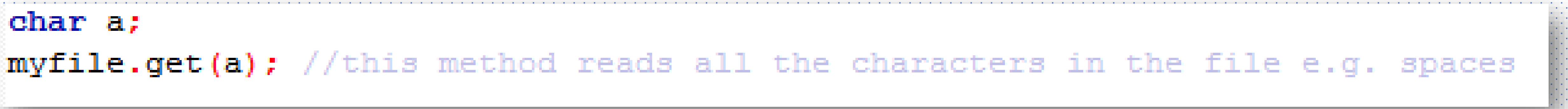
- ➢ Line by line, while using **`getline();`**
- ➢ Word by word, while using **`>>`**; (just like with cin>>)
- ➢ Character by character, while using **`get().`**



```
string line;
getline(myfile, line); //we read a complete line
```



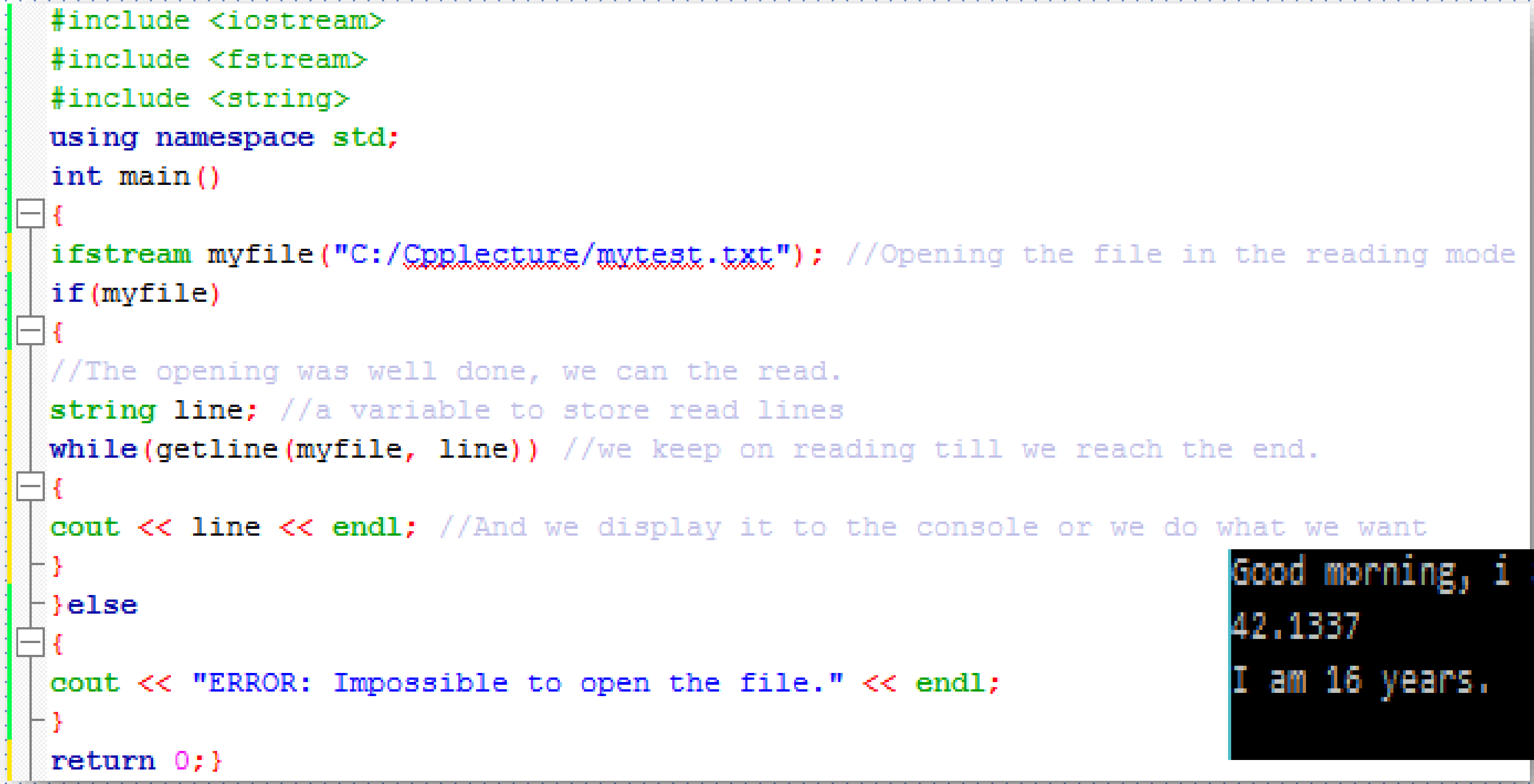
```
double number;
myfile >> number; //reads a decimal value from the file
string word;
myfile >> word; //reads a word from the file
```



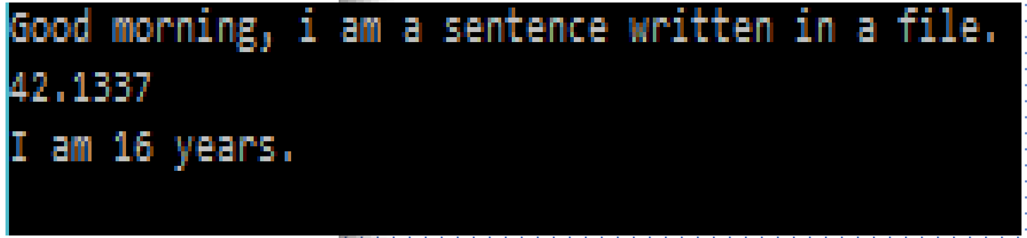
```
char a;
myfile.get(a); //this method reads all the characters in the file e.g. spaces
```

Files: Reading files

If we wish to **read a whole file**, then we need to read each line (using getline()) till there is no more line to read. The getline() function in this case will not only serve as to read. It wil equally return a bool indicating if we can continue to read. If it returns **false**, then it means we have arrived the end. To do so, we may use a **while** loop.



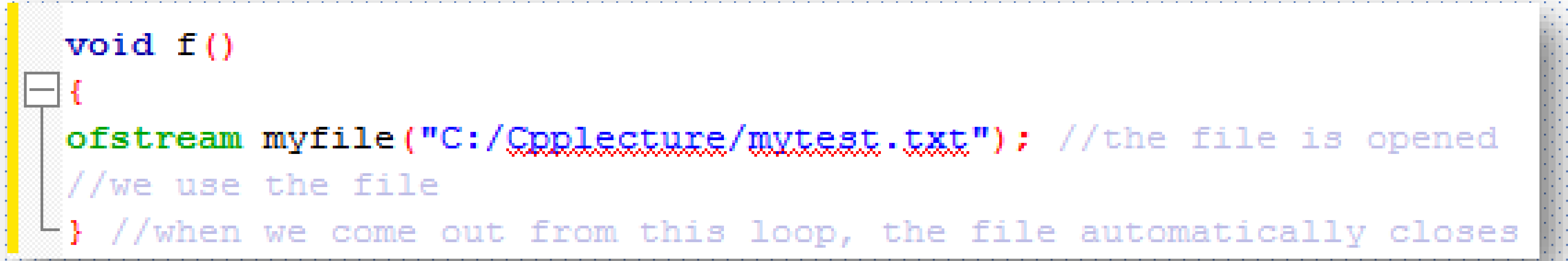
```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ifstream myfile("C:/Cpplecture/mytest.txt"); //Opening the file in the reading mode
if(myfile)
{
//The opening was well done, we can the read.
string line; //a variable to store read lines
while(getline(myfile, line)) //we keep on reading till we reach the end.
{
cout << line << endl; //And we display it to the console or we do what we want
}
}else
{
cout << "ERROR: Impossible to open the file." << endl;
}
return 0;}
```



```
Good morning, i am a sentence written in a file.
42.1337
I am 16 years.
```

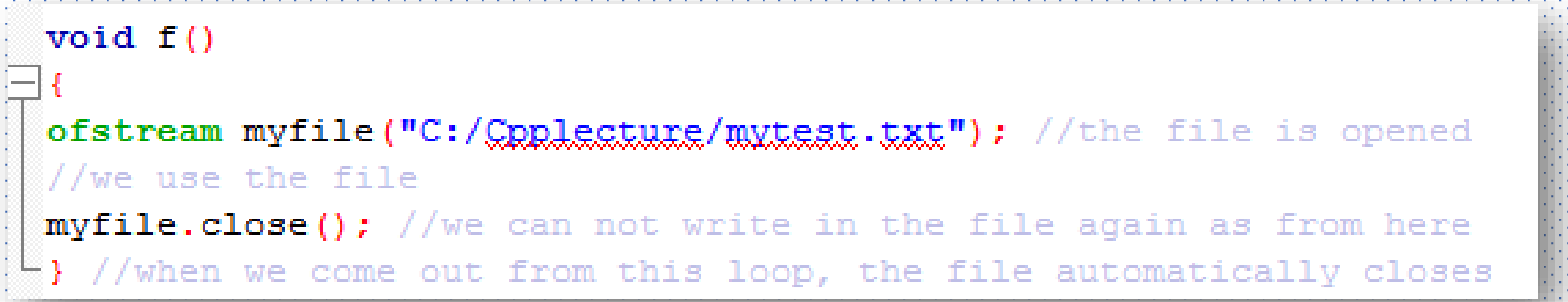
Files: Closing a file prematurely

We saw how to open a file but we didn't see how to close the file. This is because the opened file automatically closes when we come out from its stream loop. Hence, no need to close the file.



```
void f()
{
ofstream myfile("C:/Cpplecture/mytest.txt"); //the file is opened
//we use the file
} //when we come out from this loop, the file automatically closes
```

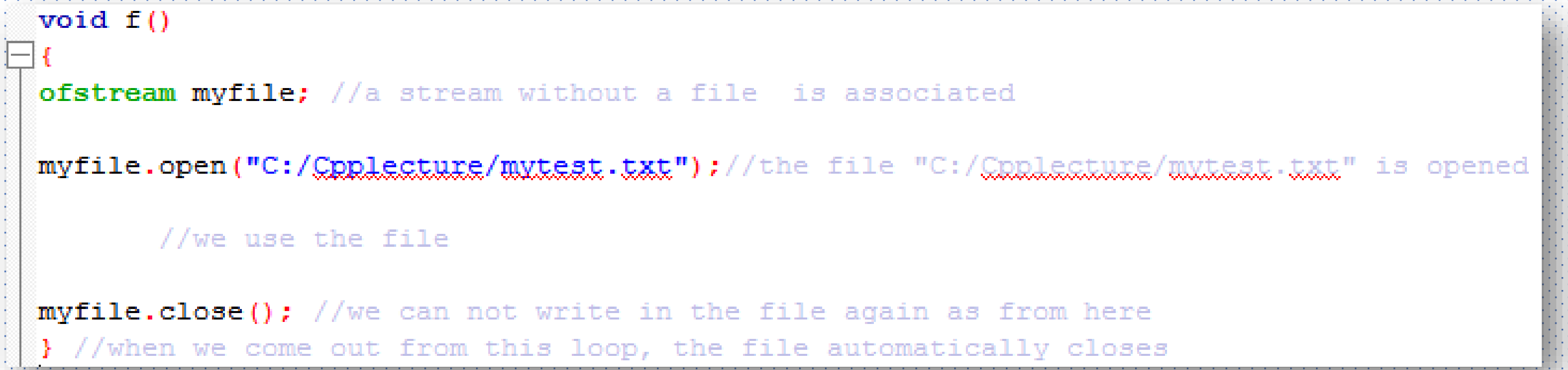
On the other hand, we may want to close a file before it is automatically closed. In this case, we should use the function **close()** .



```
void f()
{
ofstream myfile("C:/Cpplecture/mytest.txt"); //the file is opened
//we use the file
myfile.close(); //we can not write in the file again as from here
} //when we come out from this loop, the file automatically closes
```

Files: Closing a file prematurely

Similarly, we can delay the opening of a file by using `open()`.

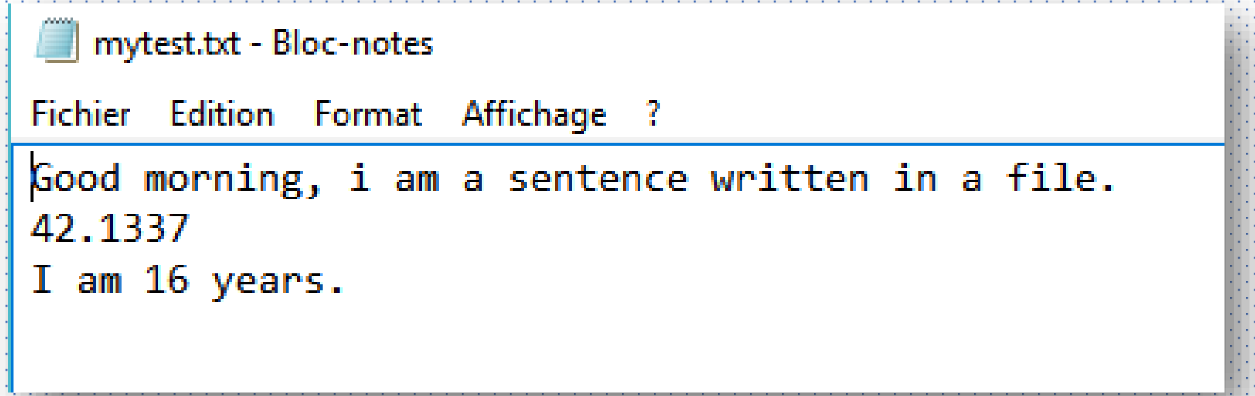


```
void f()
{
ofstream myfile; //a stream without a file  is associated

myfile.open("C:/Cpplecture/mytest.txt");//the file "C:/Cpplecture/mytest.txt" is opened

       //we use the file

myfile.close(); //we can not write in the file again as from here
} //when we come out from this loop, the file automatically closes
```

To conclude, it is simple to deal with open() and close(), but some people prefer to open the file automatically and to let it close automatically, it's a matter of choice.

Files: Cursor position

When we obtain the file C:/Cpplecture/mytest.txt, we observe that the cursor is placed at the begining of the first word. So, if e wish to read, we are uplarged to read line by line.



```
Good morning, i am a sentence written in a file.
42.1337
I am 16 years.
```

This isn’t very practical, we can’t just read the portion we want to this way.

Fortunately, there exist methods in C++ to displace the cursor. The first thing we should do is to **locate** the cursor position and then we displace ourselves.

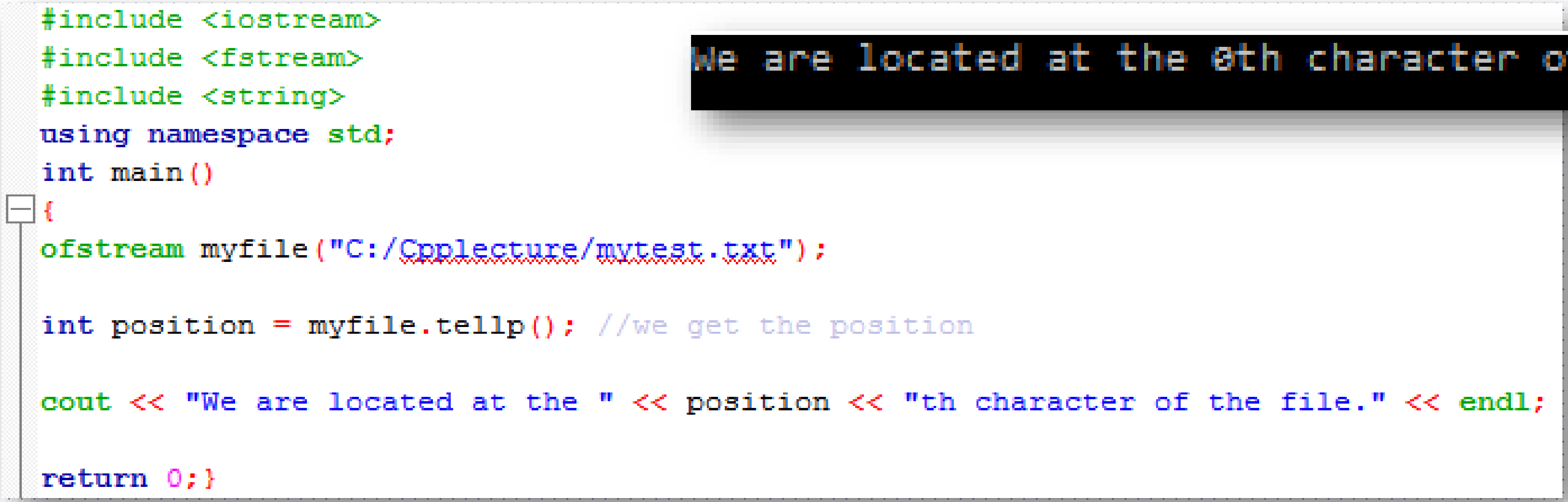
Case of ifstream	Case of ofstream
tellg	tellp

LOCATE

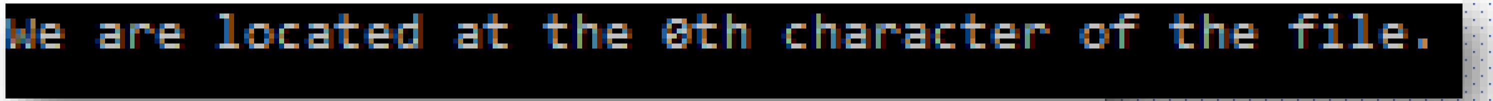
Case of ifstream	Case of ofstream
seekg	seekp

POSITION

Files: Cursor position



```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ofstream myfile("C:/Cpplecture/mytest.txt");

int position = myfile.tellp(); //we get the position

cout << "We are located at the " << position << "th character of the file." << endl;

return 0;}
```



```
We are located at the 0th character of the file.
```

Files: Cursor position

- `Seekg` and `seekp` have the same syntax.

```
nameOfFile.seekp(numberOfCharacters, position);
```

- **There exist 3 different positions:**
  - At the beginning of the file: `ios::beg`
  - At the end of the file: `ios::end`
  - At the actual position: `ios::cur`

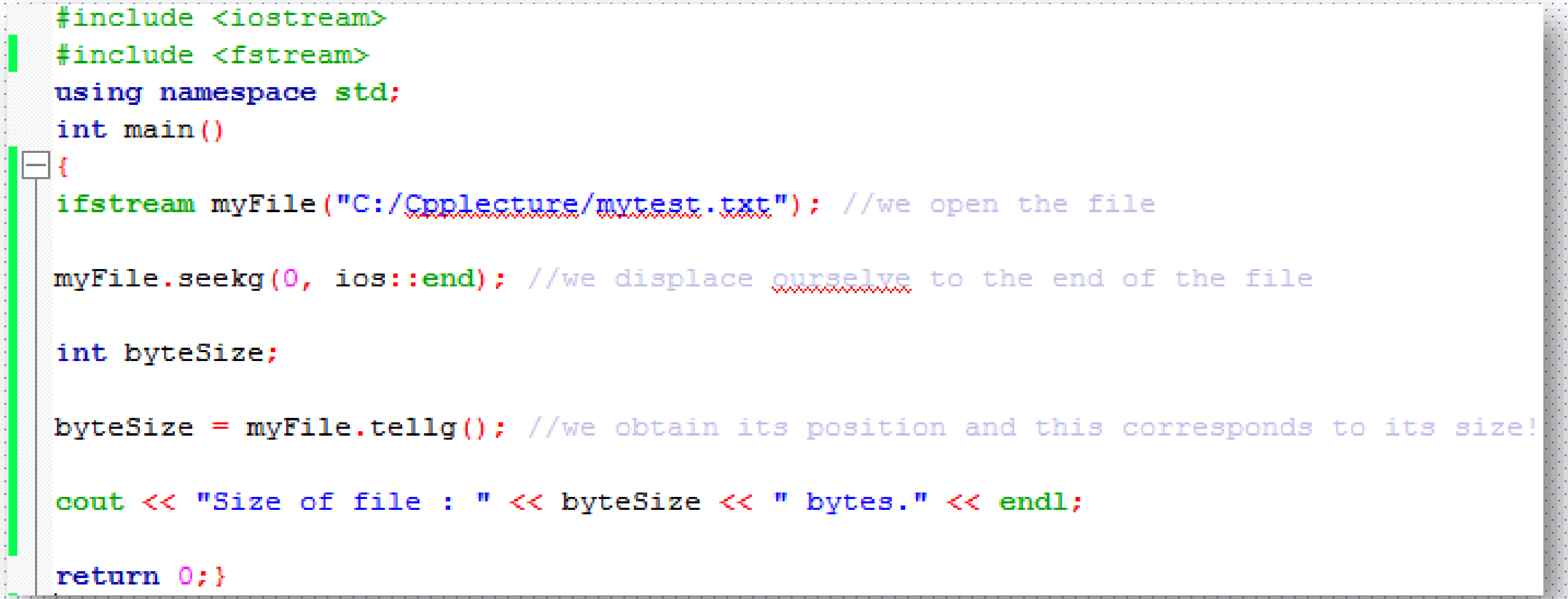
For example:

`myfile.seekp(10, ios::beg);` positions my cursor 10 characters after the beginning of the file.

`myfile.seekp(20, ios::cur);` positions my cursor 20 characters further than where it was.

Files: Size of a file

To know the size of a file, we simply displace ourselve to the end of th file and we ask th cursor to tell us his current position.



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream myFile("C:/Cpplecture/mytest.txt"); //we open the file

myFile.seekg(0, ios::end); //we displace ourselve to the end of the file

int byteSize;

byteSize = myFile.tellg(); //we obtain its position and this corresponds to its size!

cout << "Size of file : " << byteSize << " bytes." << endl;

return 0;}
```

Files: Summary

- In C++, if you wish to read or write in a file, we should include at the top `<fstream>`
- We should create a variable of type `ofstream` if we wish to open a file in the writing mode, and use `ifstream` if we wish to open the file in the reading mode.
- In order to write, we do like with cout: `myFile << " Text " ;`
- In order to read, we do like with cin: `myFile >> Variable;`
- We can read line by lin using `getline()`
- The cursor indicates at which position you are in the file during a reading or writing operation.

Exercise: Guess « word game »

SECTION A:

We wish to realise a game such that:

- ***Player 1 types a word on the keyboard;***
- ***The computer mixes up the letters of the word;***
- ***Player 2 tries to guess which word was inserted by player 1.***
- ***Your game should stop when Player 2 wins.***

```
Type a word
MAGIC

Find the word : CGAIM
MGICA
You lost !

Find the word : CGAIM
GAMIC
You lost !

Find the word : CGAIM
MAGIC !
Congratulations, you worn !
```

Exercise: Guess « word game »

SECTION B:

Do the following updates to the program you developed in section A.

- ***Make in such a way that when player 1 inserts the word, player 2 should not be capable of seeing the word. (you may use mny endl for example)***
- ***Propose to the player to play again***
- ***Limit the number of attempts to get the right word. e.g. when the player looses the first time, you remind him that 4 attempts are left.***
- ***Calculate the average score of the player. Your program should display him his score at the end of the game. You are free touse any method for score evaluation.***

Exercise: Guess « word game »

SECTION C:

Do the following updates to the program you developed in section B.

- ***Try to play the game without the use of player 1.***

To do so, we suggest you to create a text dictionary that contain a series of names (one per line) and your program should slet any of these names at random to play the game.

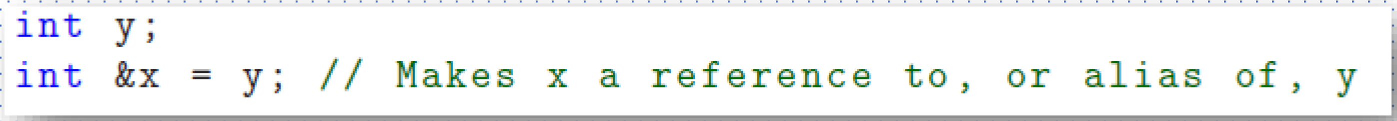
SECTION D:

Bring forth any personal update to the game

POINTERS

Pointers

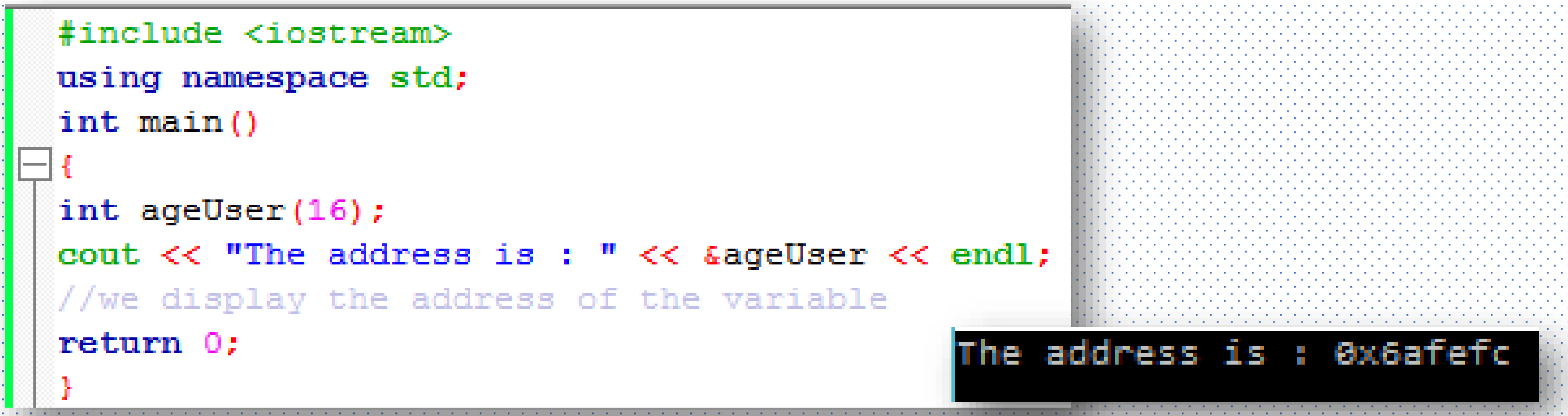
- When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.
- When you refer to the variable by name in your code, the computer must take two steps:
  - ➢ Look up the address that the variable name corresponds to
  - ➢ Go to that location in memory and retrieve or set the value it contains
- C++ allows us to perform either one of these steps independently on a variable with the **`&`** and **`*`** operators:
  - ➢ **`&x`** evaluates to the address of x in memory.
  - ➢ **`*(&x)`** takes the address of x and dereferences it – it retrieves the value at that location in memory. **`*(&x)`** thus evaluates to the same thing as **`x`**.



```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

Pointers

- Each variable possesses a single address and each address corresponds to a single variable.
- We can then conclude the following: we can access a variable using two ways
  - ➢ ***We can use its name***
  - ➢ ***Or we can use its address.***



```
#include <iostream>
using namespace std;
int main()
{
int ageUser(16);
cout << "The address is : " << &ageUser << endl;
//we display the address of the variable
return 0;
}
```

```
The address is : 0x6afefc
```

Pointers

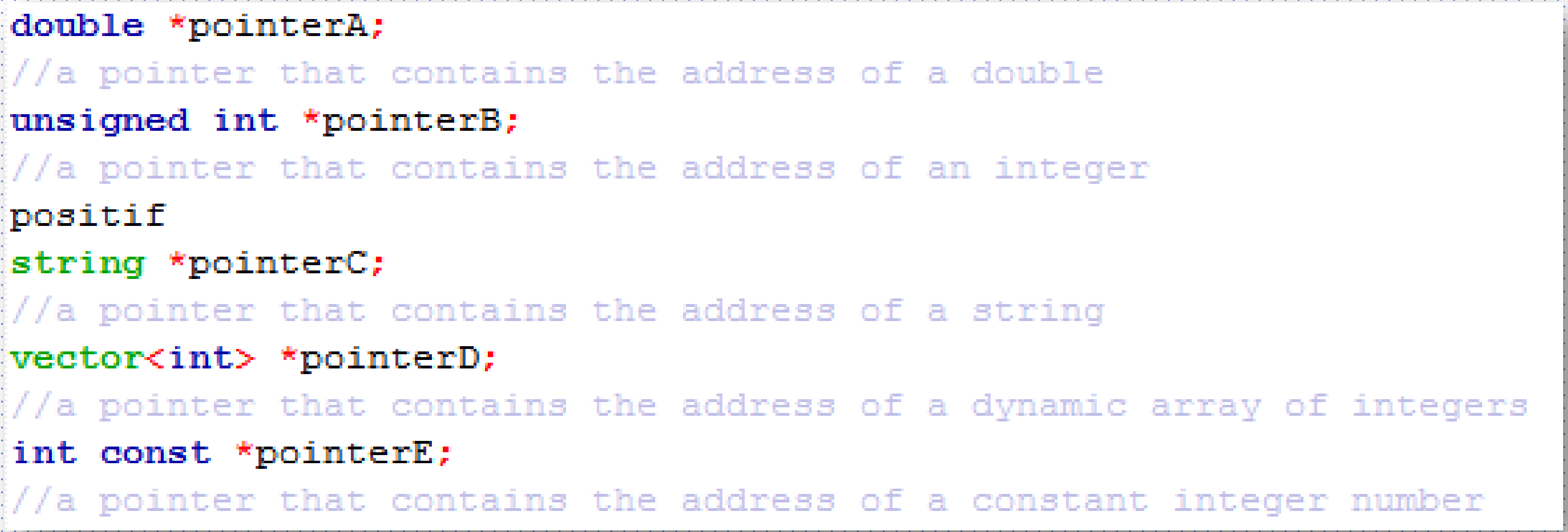
- ***A pointer is a variable that contains the address of another variable.***
- The declaration of pointers follows this format: **`type * name;`**
- The first one points to an `int`, the second one to a `char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

```
int * number;
char * character;
float * greatnumber;
```

We can equally write `int*` number (the star joined with the word `int`). This notation has as inconvenient not to allow the declaration of many pointers on the same line, like: `int* pointer1, pointer2, pointer3;` if we do so, only ponter1 will be a pointer, the two others will be standard variables of type int.

Pointers

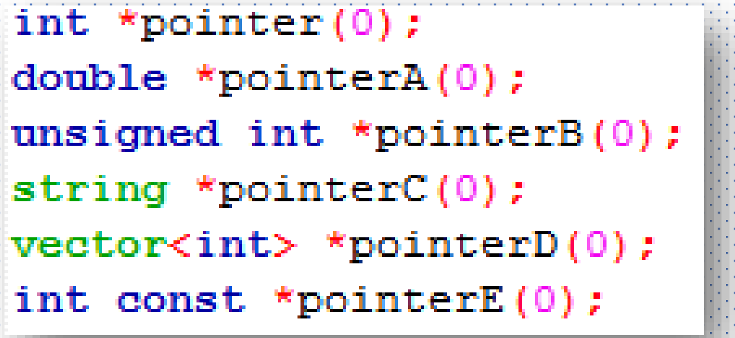
- Further declaration examples



```
double *pointerA;
//a pointer that contains the address of a double
unsigned int *pointerB;
//a pointer that contains the address of an integer
positif
string *pointerC;
//a pointer that contains the address of a string
vector<int> *pointerD;
//a pointer that contains the address of a dynamic array of integers
int const *pointerE;
//a pointer that contains the address of a constant integer number
```

Pointers

- So far, we have been declaring our pointers containing unknown addresses. **THIS IS VERY DANGEROUS!**
- This is because, by so doing, we do not know which memory cell we are dealing with, e.g. we may be playing along with the memory cell responsible for storing our OS password, our time, etc.
- **HENCE, WE SHOULD NEVER DECLARE OUR POINTERS WITHOUT ASSIGNING HIM AN ADDRESS. To do this, we shall generally declare our pointer while assigning the value 0.**

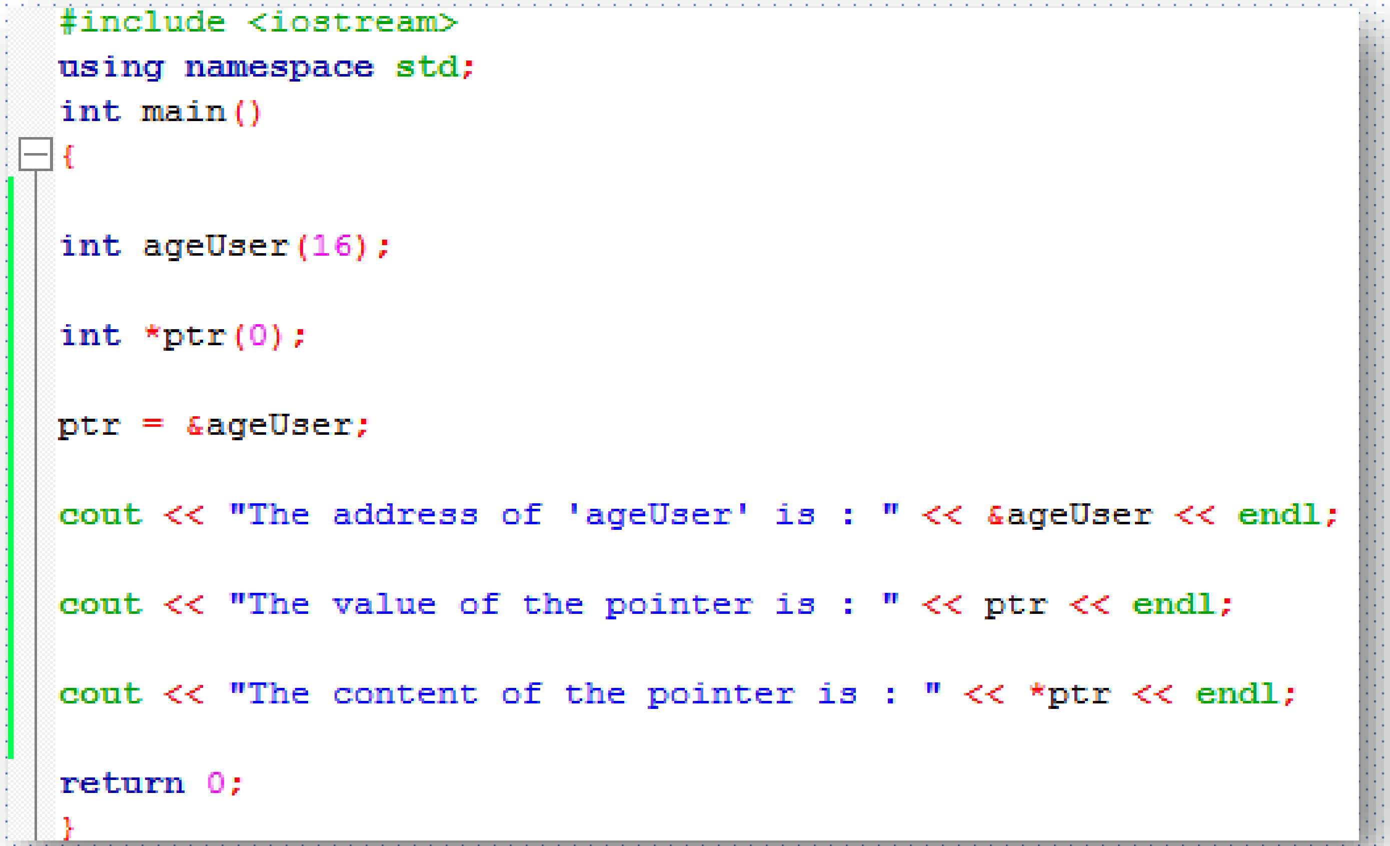


```
int *pointer(0);
double *pointerA(0);
unsigned int *pointerB(0);
string *pointerC(0);
vector<int> *pointerD(0);
int const *pointerE(0);
```

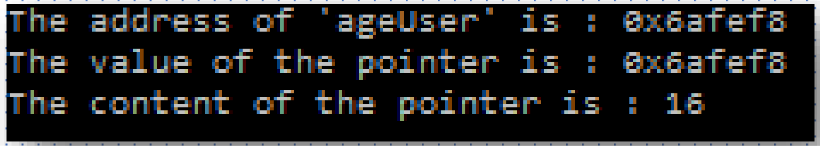
Some pointers do not point to valid data; dereferencing such a pointer is a runtime error. Any pointer set to 0 is called a null pointer, and since there is no memory location 0, it is an invalid pointer. One should generally check whether a pointer is null before dereferencing it. Pointers are often set to 0 to signal that they are not currently valid.

Pointers

- ***The value of the pointer corresponds to the address of the pointed value***



```
#include <iostream>
using namespace std;
int main()
{

int ageUser(16);

int *ptr(0);

ptr = &ageUser;

cout << "The address of 'ageUser' is : " << &ageUser << endl;

cout << "The value of the pointer is : " << ptr << endl;

cout << "The content of the pointer is : " << *ptr << endl;

return 0;
}
```

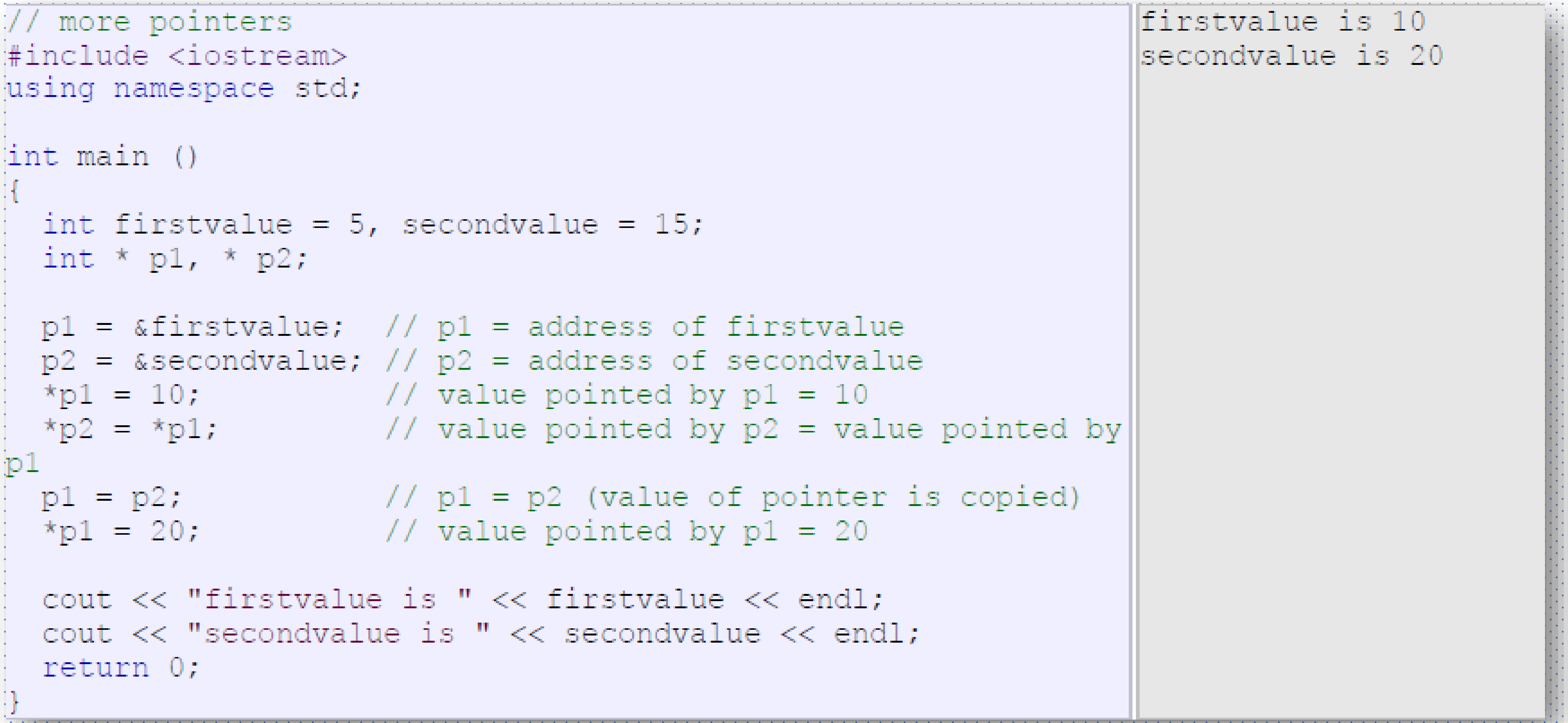


```
The address of 'ageUser' is : 0x6afef8
The value of the pointer is : 0x6afef8
The content of the pointer is : 16
```

Pointers

- The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways:
  - ➢ When declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer – say, a pointer to an `int` or char, not an `int` or `char` value.
  - ➢ When using a pointer that has been set to point to some value, * is placed before the pointer name to dereference it – to access or set the value it points to.
- A similar distinction exists for &, which can be used either to indicate a reference data type (as in `int &x`;), or to take the address of a variable (as in `int *ptr = &x`;).

Pointers

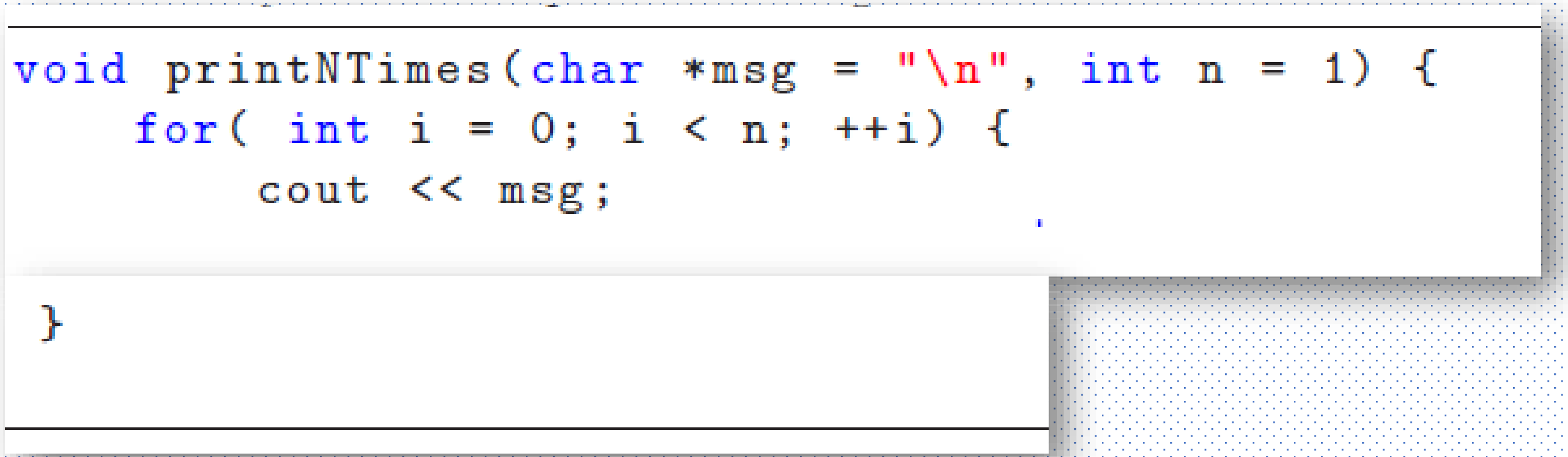


```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by
p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Pointers

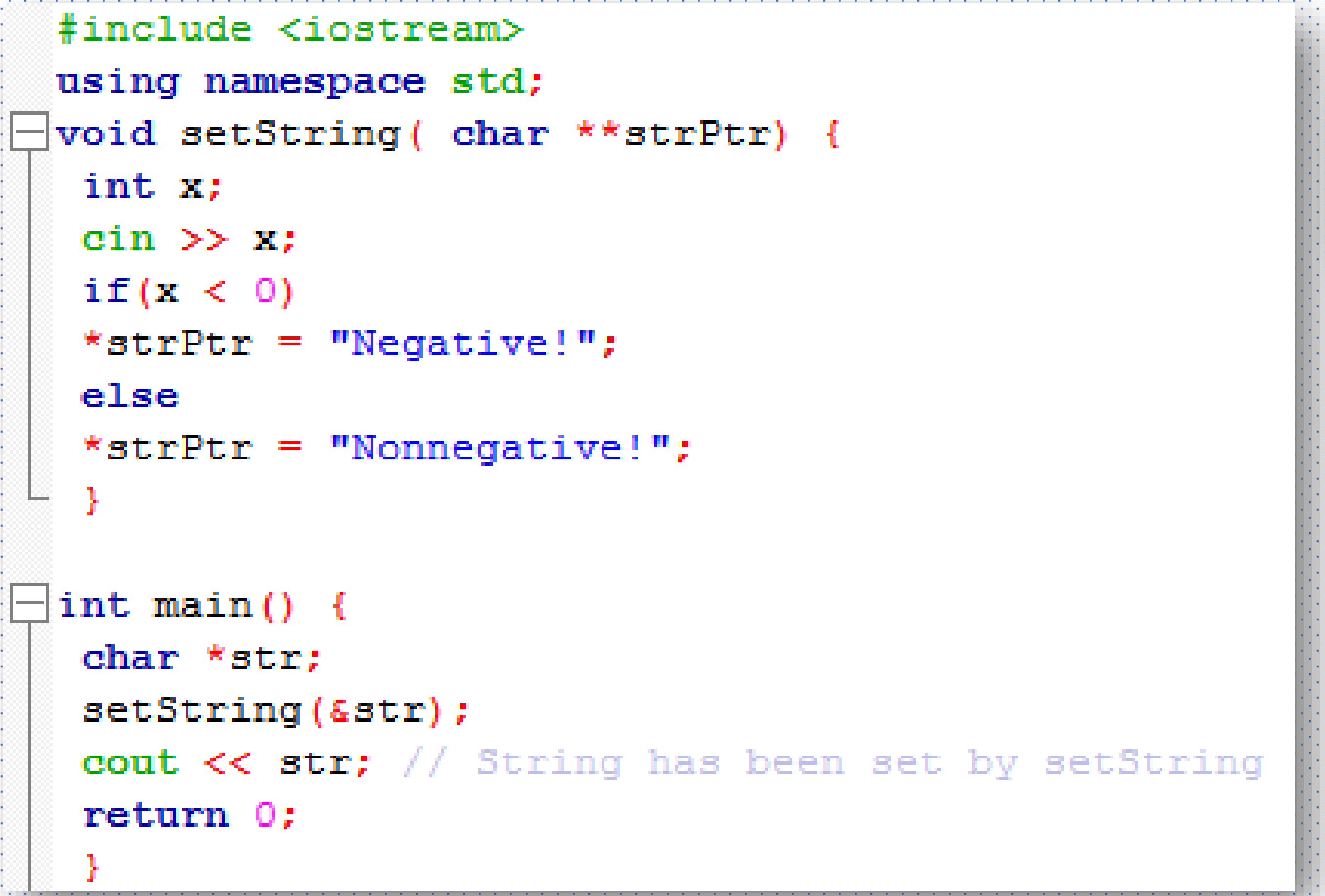
Say you have a function with 1 argument, but that argument is usually the same. For instance, say we want a function that prints a message n times, but most of the time it will only need to print it once:



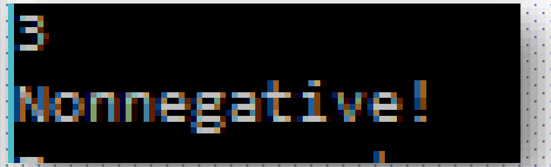
```
void printNTimes(char *msg = "\n", int n = 1) {
    for( int i = 0; i < n; ++i) {
        cout << msg;

}
```

Pointers

We can have pointers to any type, including **pointers to pointers.** This is commonly used in C(and less commonly in C++) to allow functions to set the values of pointers in their calling functions. For example:



```
#include <iostream>
using namespace std;
void setString( char **strPtr) {
 int x;
 cin >> x;
 if(x < 0)
 *strPtr = "Negative!";
 else
 *strPtr = "Nonnegative!";
 }

int main() {
 char *str;
 setString(&str);
 cout << str; // String has been set by setString
 return 0;
 }
```

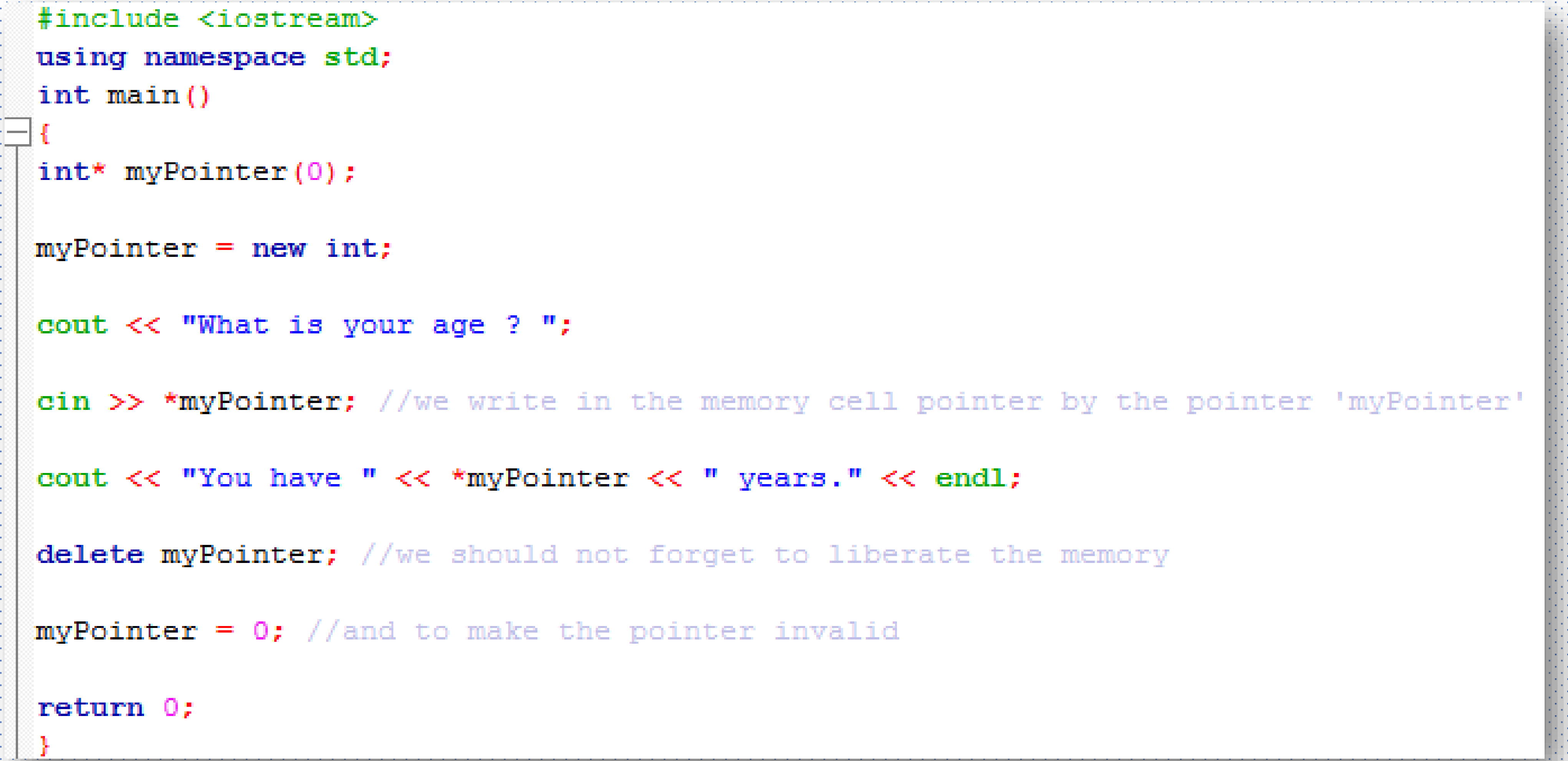


```
3
Nonnegative!
```

Pointers

- If we wish to manually ask a memory cell to the computer (allocation), we need to use the operator **`new`**. This operator returns a **pointer** pointing towards that memory cell you asked for. ***Be very careful when using this approach because if you change the value of the pointer, you loss the only way of accessing that memory cell. That memory cell will be lost though it will continue to occupy space.***
- Once we do not need the cell again, we liberate the cell (deallocate) by using the operator `delete`.
- ***Do not forget to liberate the memory, or else your program will occupy more and more space till there will be no space left, and then it will bug.***
- ***After viewing the example in the next slide, one may feel that this approach makes things more difficult but we shall understand its necessity when dealing with the creation of windows***

Pointers



```
#include <iostream>
using namespace std;
int main()
{
int* myPointer(0);

myPointer = new int;

cout << "What is your age ? ";

cin >> *myPointer; //we write in the memory cell pointer by the pointer 'myPointer'

cout << "You have " << *myPointer << " years." << endl;

delete myPointer; //we should not forget to liberate the memory

myPointer = 0; //and to make the pointer invalid

return 0;
}
```

Pointers

```
pointer = new type
pointer = new type [number_of_elements]
```

```
int * bobby;
bobby = new int [5];
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution. This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
bobby = new (nothrow) int [5];
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

```
delete pointer;
delete [] pointer;
```

Pointers

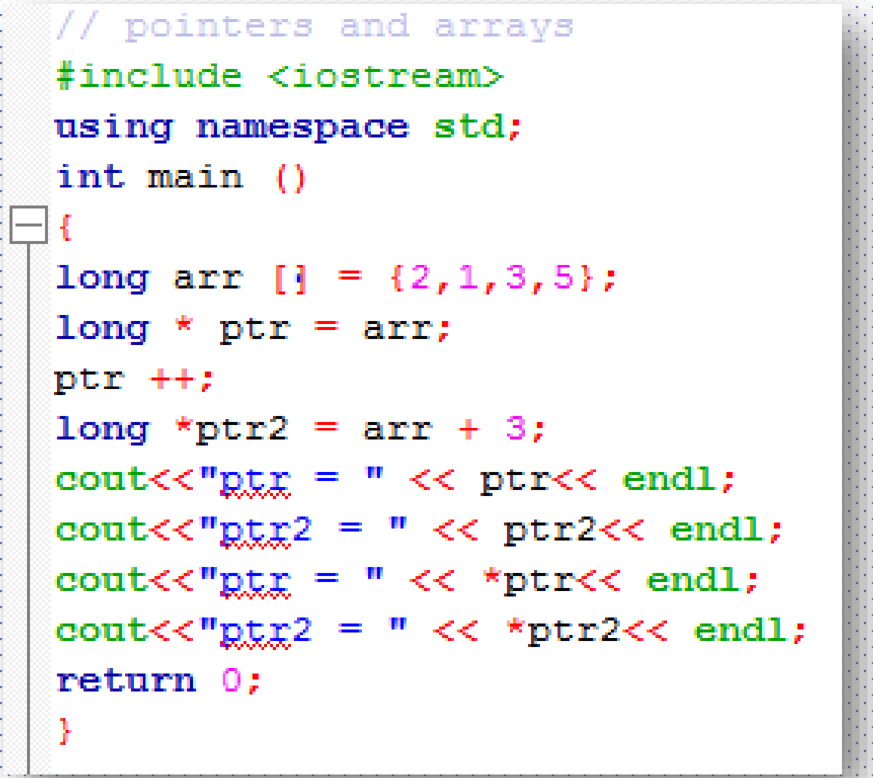
```cpp
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

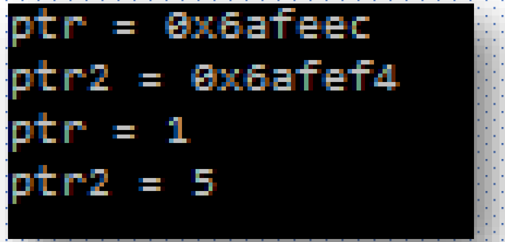
Notice how the value within brackets in the `new` statement is a variable value entered by the user (`i`), not a constant value:

```cpp
p= new (nothrow) int[i];
```

Pointers



```
// pointers and arrays
#include <iostream>
using namespace std;
int main ()
{
long arr [] = {2,1,3,5};
long * ptr = arr;
ptr ++;
long *ptr2 = arr + 3;
cout<<"ptr = " << ptr<< endl;
cout<<"ptr2 = " << ptr2<< endl;
cout<<"ptr = " << *ptr<< endl;
cout<<"ptr2 = " << *ptr2<< endl;
return 0;
}
```

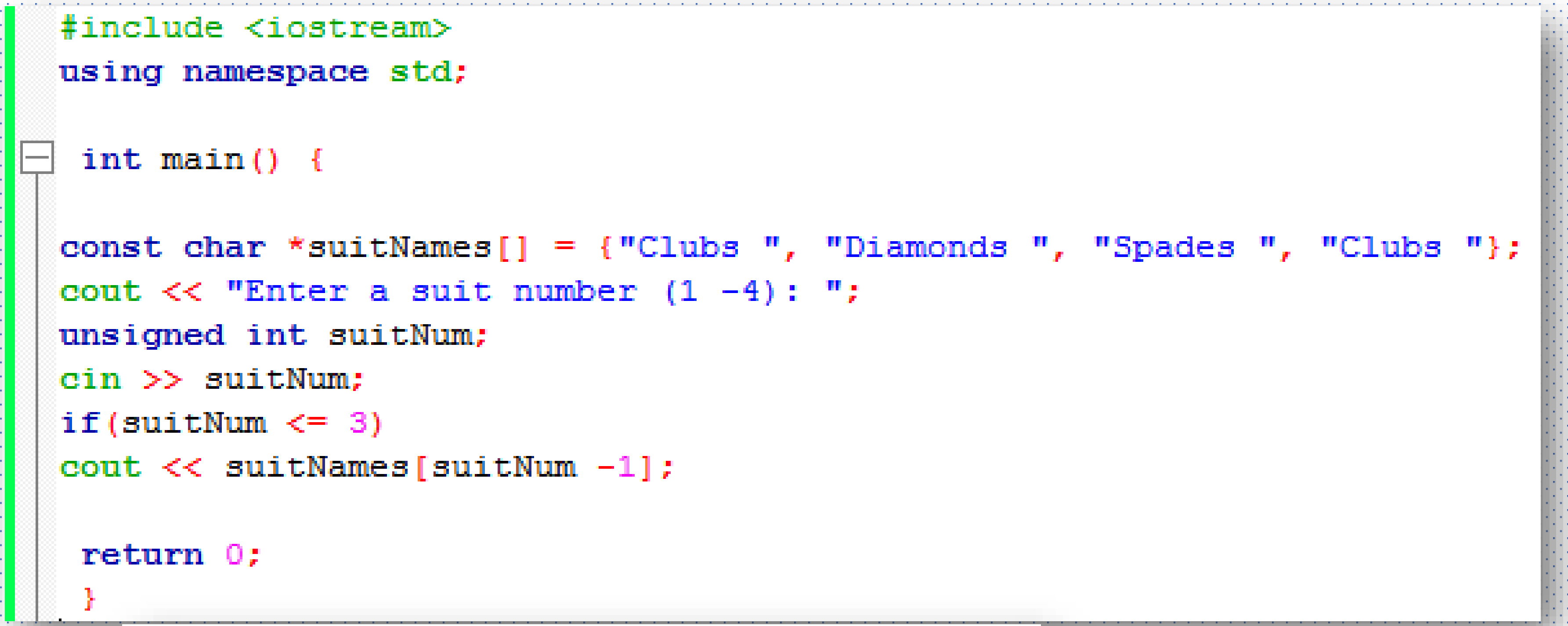


```
ptr = 0x6afeec
ptr2 = 0x6afef4
ptr = 1
ptr2 = 5
```

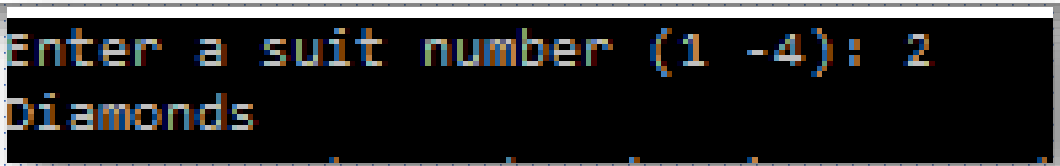
Because of the interchangeability of pointers and array names, array-subscript notation (the form `myArray[3]`)can be used with pointers as well as arrays. When used with pointers, it is referred to as *pointer-subscript notation*.

Analternative is *pointer-offset notation*, in which you explicitly add your off set to the pointer and dereference the resulting address. For instance, an alternate and functionally identical way to express **`myArray[3]` is `*(myArray + 3)`.**

Pointers



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

 int main() {

const char *suitNames[] = {"Clubs ", "Diamonds ", "Spades ", "Clubs "};
cout << "Enter a suit number (1 -4): ";
unsigned int suitNum;
cin >> suitNum;
if(suitNum <= 3)
cout << suitNames[suitNum -1];

 return 0;
 }
```



```
Enter a suit number (1 -4): 2
Diamonds
```

Pointers

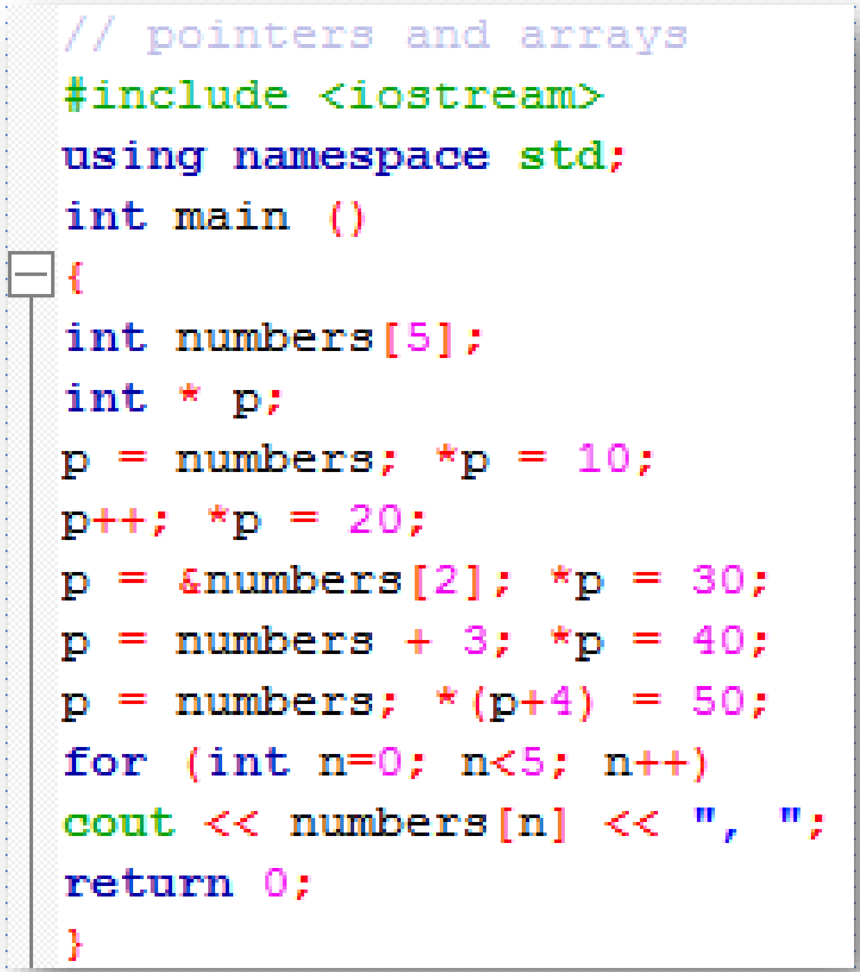
It is important to note that arrays in C++ are pointers to continuous regions in memory. Therefore the following code is valid:

```
const int ARRAY_LEN = 100;
int arr[ARRAY_LEN];
int *p = arr;
int *q = &arr[0];
```

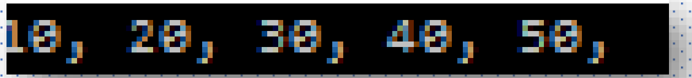
Now p and q point to exactly the same location as `arr` (ie. `arr[0]`), and `p`, `q` and `arr` can be used interchangeably. You can also make a pointer to some element in the middle of an array(similarly to q):

```
int *z = &arr[10];
```

Pointers



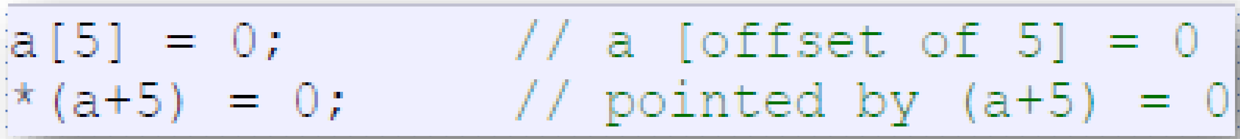
```
// pointers and arrays
#include <iostream>
using namespace std;
int main ()
{
int numbers[5];
int * p;
p = numbers; *p = 10;
p++; *p = 20;
p = &numbers[2]; *p = 30;
p = numbers + 3; *p = 40;
p = numbers; *(p+4) = 50;
for (int n=0; n<5; n++)
cout << numbers[n] << ", ";
return 0;
}
```



```
10, 20, 30, 40, 50,
```

In the chapter about arrays we used brackets ([]) several times in order to specify the index of an element of the array to which we wanted to refer. Well, these bracket sign operators [] are also a dereference operator known as *offset operator*. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

These two expressions are equivalent and valid both if a is a pointer or if a is an array.



```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

Pointers

You should now be able to see why the type of a string value is `char *`: a string is actually an array of characters. When you set a `char *` to a string, you are really setting a pointer to point to the first character in the array that holds the string.

You cannot modify string literals; to do so is either a syntax error or a runtime error, depending on how you try to do it. (String literals are loaded into read-only program memory at program startup.) You can, however, modify the contents of an array of characters. Consider the following example:

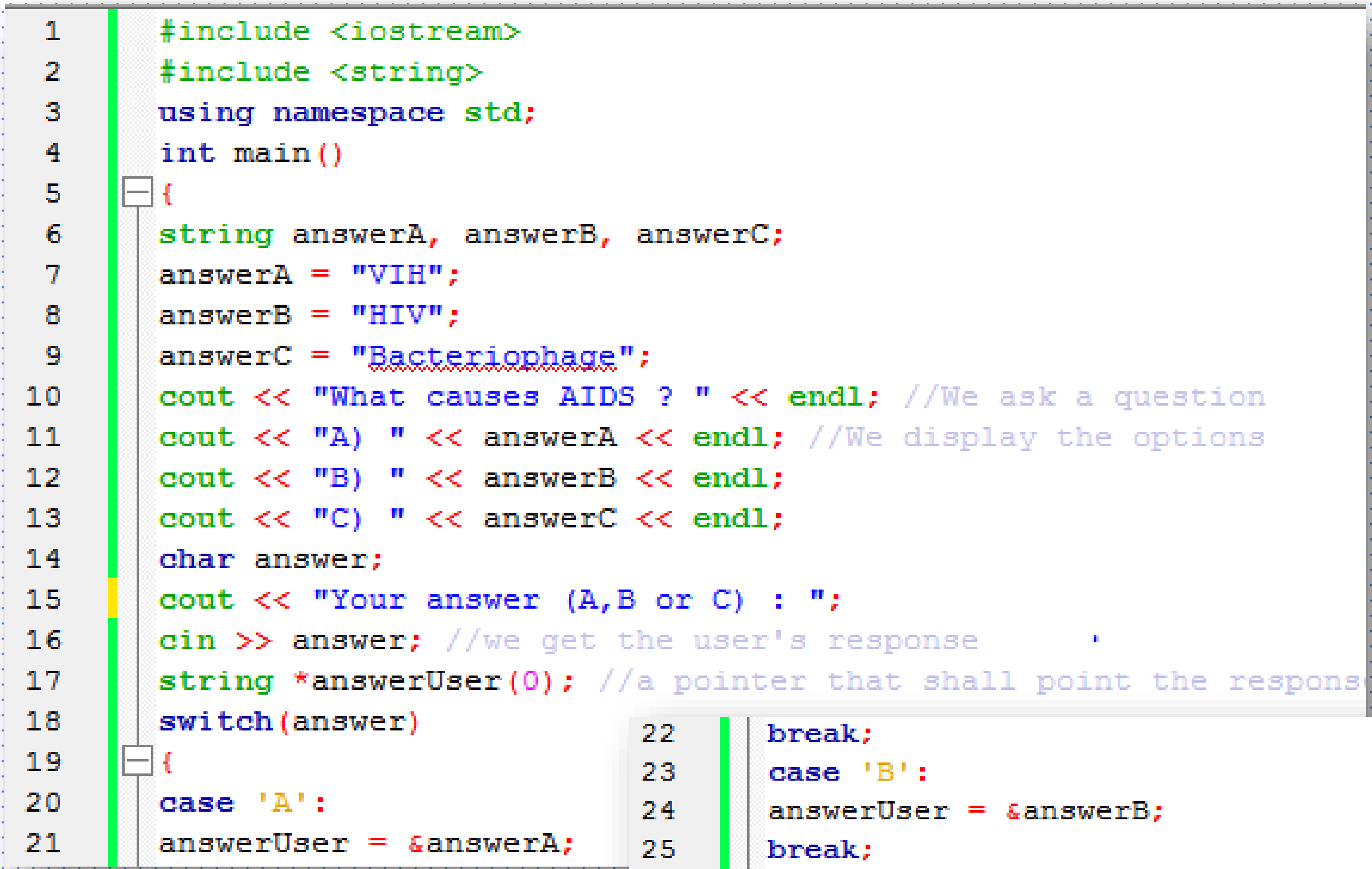
```
char courseName1[] = {'6', '.', '0', '9', '6', '\0 ' };
char *courseName2 = "6.096 ";
```

Attempting to modify one of the elements courseName1 is permitted, but attempting to modify one of the characters in courseName2 will generate a runtime error, causing the program to crash.

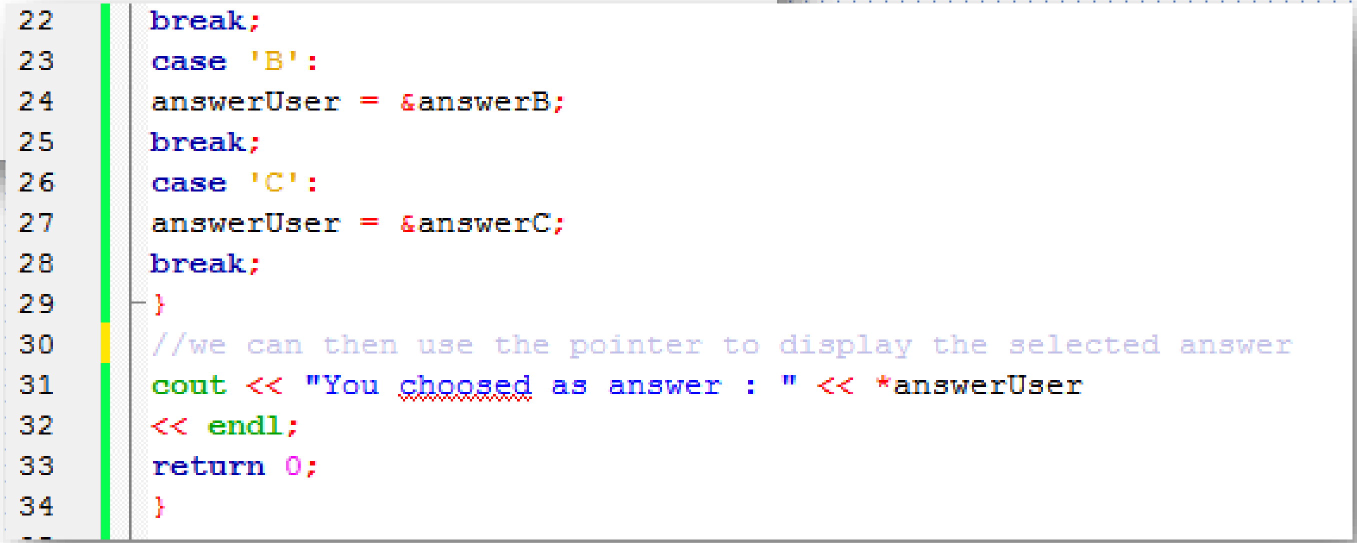
Pointers

We discussed in lecture how variables can be declared at global scope or file scope –if a variable is declared outside of any function, it can be used any where in the file. For any thing besides global constants such as error codes or fixed array sizes, this is usually a bad idea; if you need to access the same variable from multiple functions, most often you should simply pass the variable around as an argument between the functions. **Avoid global variables when you can**.

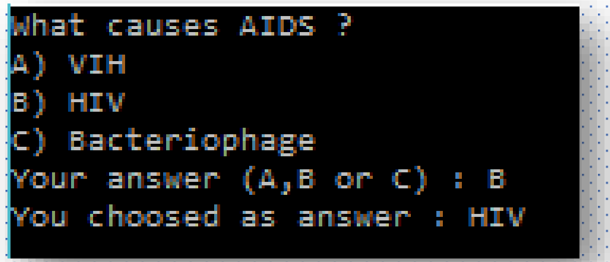
Pointers



```cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
string answerA, answerB, answerC;
answerA = "VIH";
answerB = "HIV";
answerC = "Bacteriophage";
cout << "What causes AIDS ? " << endl; //We ask a question
cout << "A) " << answerA << endl; //We display the options
cout << "B) " << answerB << endl;
cout << "C) " << answerC << endl;
char answer;
cout << "Your answer (A,B or C) : ";
cin >> answer; //we get the user's response
string *answerUser(0); //a pointer that shall point the respons
switch(answer)
{
case 'A':
answerUser = &answerA;
```



```cpp
break;
case 'B':
answerUser = &answerB;
break;
case 'C':
answerUser = &answerC;
break;
}
//we can then use the pointer to display the selected answer
cout << "You choosed as answer : " << *answerUser
<< endl;
return 0;
}
```



```
What causes AIDS ?
A) VIH
B) HIV
C) Bacteriophage
Your answer (A,B or C) : B
You choosed as answer : HIV
```

Pointers

IMPORTANCE OF POINTERS

Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:

- More flexible pass-by-reference
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
- Use polymorphism – calling functions on data without knowing exactly what kind of data it is.

WHEN TO USE POINTERS

There are 3 main applications:

- Managing the moment of creation and of destruction
- Sharing a data with many pieces of your code (*e.g. in games*)
- Selecting a value among many others (*view previous slide*)

Pointers: Summary

- *Each variable is stored in memory in a different address*
- *We can obtain the address of a variable by using* `&variable`*.*
- *A pointer is a variable that points on the address of another variable.*
- *By default, a pointer displays the address it contains. But* `*pointer` *displays the value found in the address indicated by the pointer.*
- *We can manually reserve a memory cell using* `new`*. In this case, we should free the cell when we do not need it again using* `delete`*.*

THIS TOPIC (POINTERS) IS NATURALLY COMPLEX TO UNDERSTAND ALL AT ONCE, SO PLEASE READ IT SEVERAL TIMES.

Exercises:

Exercise 1:

1. Write a function `printArray` to print the contents of an integer array with the string "`,` " between elements (but not after the last element). Your function should return nothing.

2. Write a reverse function that takes an integer array and its length as arguments. Your function should reverse the contents of the array, leaving the reversed values in the original array, and return nothing.

3. Assume the existence of two constants WIDTH and LENGTH. Write a function with the following signature: `void transpose( const int input[][LENGTH], int output[][WIDTH]);` Your function should transpose the WIDTH × LENGTH matrix in input, placing the LENGTH × WIDTH transposed matrix into output.

4. What would happen if, instead of having output be an "out argument," we simply declared a new array within transpose and returned that array?

5. Rewrite your function from part 2 to use pointer-offset notation instead of array-subscript notation.

Exercises:

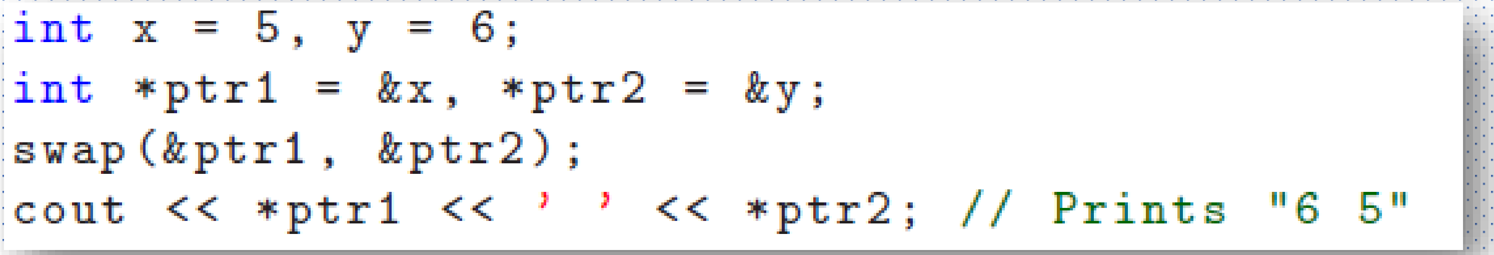
Exercise 2:

1. Write a function that returns the length of a string (**`char *`**), excluding the final NULL character. It should not use any standard-library functions. You may use arithmetic and dereference operators, but not the indexing operator(**`[]`**).

2. Write a function that swaps two integer values using call-by-reference.

3. Rewrite your function from part 2 to use pointers instead of references.

4. Write a function similar to the one in part 3, but instead of swapping two values, it swaps two pointers to point to each other’s values. Your function should work correctly for the following example invocation:

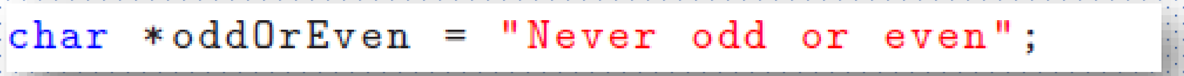


```
int x = 5, y = 6;
int *ptr1 = &x, *ptr2 = &y;
swap(&ptr1, &ptr2);
cout << *ptr1 << ’ ’ << *ptr2; // Prints "6 5"
```

Exercises:

Exercise 2:

5. Assume that the following variable declaration has already been made:



```
char *oddOrEven = "Never odd or even";
```

Write a single statement to accomplish each of the following tasks (assuming for each one that the previous ones have already been run). Make sure you understand what happens in each of them.

a) Create a pointer to a char value named **`nthCharPtr`** pointing to the 6th character of **`oddOrEven`** (remember that the first item has index 0). Use the indexing operator.

b) Using pointer arithmetic, update **`nthCharPtr`** to point to the 4th character in **`oddOrEven`**.

c) Print the value currently pointed to by **`nthCharPtr`**.

d) Create a new pointer to a pointer (a **`char **`**) named **`pointerPtr`** that points to **`nthCharPtr`**.

e) Print the value stored in **`pointerPtr`**.

f) Using **`pointerPtr`**, print the value pointed to by **`nthCharPtr`**.

g) Update **`nthCharPtr`** to point to the next character in **`oddOrEven`** (i.e. one character past the location it currently points to).

h) Using pointer arithmetic, print out how far away from the character currently pointed to by **`nthCharPtr`** is from the start of the string.

DATA STRUCTURES

Structures

READ ON THE FOLLOWING

- ***Structures,***
- ***Linked list,***
- ***Queues,***
- ***Stacks,***
- ***Trees,***
- ***Binary trees,***
- ***Binary search,***
- ***Sorting algorithms.***

PART 2
OBJECT ORIENTED PROGRAMMING

1. The Context of Software Development
2. Writing a C++ Program
3. Values and Variables
4. Expressions and Arithmetic
5. Conditional Execution
6. Iteration
7. Other Conditional and Iterative Statements
8. Using Functions
9. Writing Functions
10. Managing Functions and Data
11. Sequences
12. Sorting and Searching
13. Standard C++ Classes
14. Custom Objects
15. Fine Tuning Objects
16. Building some Useful Classes
17. Inheritance and Polymorphism
18. Memory Management
19. Generic Programming
20. The Standard Template Library
21. Associative Containers
22. Handling Exceptions