

OOP

Lecture 16

7/12/2021

Default Parametric Functions :

The default value of an argument is provided in a function declaration that is automatically assigned to the parameters that are written in the function prototype by the compiler. The function with default parameters can be called with or without parameters. If the caller of the function doesn't provide a value for the argument then the default value will be assigned to the parameter. In case any value is passed the default value is overridden.

Default values can be assigned by simply initializing the variables written inside the parentheses of the function at the time of declaration.

Example : **void function(int a = 10, int b = 20);**

- This Function can be called with 0, 1, or 2 parameters. If the value of the parameter is passed then it will be overwritten, otherwise default values will be used inside the function.
- There may be a case when we have few arguments default and few arguments compulsory, like,

void function(int a, int b = 20, int c = 30);

In such a case, The function can be called with 1, 2, or 3 arguments.

- There's an important point to note that all the default arguments should be written at the right side, and the arguments without default values should be on the left. Like,

void function(int a = 10, int b, int c = 30);

void function(int a, int b = 20, int c = 30, int d);

These 2 statements will give an error as the non-default argument is written after the default argument.

Sample Programs :

```
#include <iostream>

using namespace std;

int func(int x = 10, int y = 20, int z = 30)
{
    cout << "X: " << x << "\nY: " << y << "\nZ: " << z;
    cout << "\n-----\n";
}

int main()
{
    func();
    func(2);
    func(2, 4);
    func(2, 4, 8);

    return 0;
}
```

Select E:\program.exe

X: 10
Y: 20
Z: 30

X: 2
Y: 20
Z: 30

X: 2
Y: 4
Z: 30

X: 2
Y: 4
Z: 8

sources Compile Log Debug Find Results X

- Output Filename: E:\program.exe

```

#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
    return (x + y + z + w);
}

int main()
{
    // Statement 1
    cout << sum(10, 15) << endl;

    // Statement 2
    cout << sum(10, 15, 25) << endl;

    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}

```

```

E:\program.exe
25
50
80
-----
Process exited after
Press any key to co

```

The Default Parameters technique is different from **Function Overloading**. Function overloading is basically applying different logic for different numbers of parameters, but the default values are used when we have to apply the same logic on any number of parameters.

Moving Through the Data:

We have already covered the basics of what is moving through the data in the files. The `seekg()` and `seekp()` functions are used to move the reading and writing pointers in the file to read or write at a specific portion in the file. In Simple Words, They are used to control the positions of where we want to read the data from a file and where to write the data into a file.

seekg() and seekp() :

The seekg() and seekp() functions move the pointers across the data in a file. These functions take 2 arguments and the second argument has a default value in its definition. It means that these functions can be called with 1 or 2 arguments.

Syntax :

- seekg(no. of bytes)
- seekg(no. of bytes, reference point)

The Syntax will remain the same for the seekp() function as well.

Explanation :

- The first argument is “no. of bytes”, which should be an integer value. It represents the number of bytes which we want to move in the file. It can move both forward and backward, which we will see further in the lecture. Suppose, we passed the value 10 here, then it will move the pointer 10 bytes forward and start reading or writing (depending upon seekg or seekp) from that location.
- The second argument is the “reference point”. The reference point is the point which tells the compiler to move the pointer from that point. It means that if we are telling the compiler to move the pointer then we have to tell the point or position as well from where we want to move the specified number of bytes forward or backward. There are three reference points which are used to move the pointer. These are:
 1. **ios :: beg** : It is used to move the pointer from the start. It means that if we write the function like this,

file.seekg(10, ios :: beg);

It will take the reading pointer at the 11th byte of the file, leaving 10 bytes behind from the start of the file and will start reading from this location.

2. **ios :: end** : It is used to move the pointer to the end. It means that if we write the function like this,

file.seekp(0, ios :: end);

It will take the writing pointer at the end of the file, moving 0 bytes from the end of the file and will start writing at this location.

3. **ios :: curr** : It is used to move the pointer from the current position of the pointer. It means that if we write the function like this,

file.seekg(10, ios :: curr);

It will take the reading pointer at the 11th byte from the current position, leaving 10 bytes behind and will start reading from this location.

Moving Backwards in the file :

We can pass the negative values in the first argument of seekg() and seekp() functions to move backwards in the file. Like, we write the statement like this,

file.seekp(-30, ios :: end);

It will take the writing pointer at 31st byte counting backwards from the end of the file, i.e moving 30 bytes in backwards direction from the end of the file and it will start writing at this location. The writing still would be normal; it means that it will still write in the forward direction.

file.seekg(-10, ios :: curr);

It will take the reading pointer at the 11th byte counting backwards from the current position of the pointer, i.e moving 10 bytes in backwards direction from the current position and it will start reading from this location. The reading still would be normal; it means that it will still read in the forward direction.

So, what happens when we pass only 1 parameter in the seek functions?

The answer would be simple, when we pass only 1 parameter i.e the number of bytes to move, the reference point will be default. In all cases the default reference point of the pointer is ios :: beg, which means that when we pass only the number of bytes it will move the pointers according to the start of the file.

tellg() and tellp() functions :

tellg() and tellp() functions are used to check the current positions of the reading and writing pointers. These functions take no arguments and return the current position of the pointer i.e the total number of bytes the pointers have passed from the start of the file.

Mostly, these functions work best in combination. Like, we can save the positions of the pointers in temporary variables and move the pointers to do some specific changes and return to its original position.

Example :

Consider this code,

```
file.seekg( 0, ios :: end );  
int size = file.tellg();  
file.seekg( 0 );
```

It is a very typical code to find the size of the file in bytes. The first statement will move the read pointer to the end and the second statement will tell the position of the pointer in bytes and will store it in the size variable. The third statement will again move the pointer at the start.

Formatted Data :

Formatted output means that **fixed sized record** (rows) in the file. When we write numbers in its original form it will take a variable number of bytes in the file like if we write 10 it will take 2 bytes and when we write 100 it will take 3 bytes. Such data is not formatted and it will take a variable number of space and bytes in the file. Like,

```
10 2000 5 54574 115 111111
```

It is an example of non-formatted data. Here, we can see that the first number takes 2 bytes next is taking 4 bytes and third is taking only 1 byte. So, if we want to access 5th number directly, there is no hard and fast rule with which we can jump directly (as we can only jump by giving a fixed

number of bytes), So, this non-formatted is not helpful for us. To jump or move to a specific number we have to use the formatted data, which means that there should be a fixed number of spaces between each number. We can assign a fixed width to every number in which it can fit, so that we can access that number with a defined number of bytes. Here, is an example of formatted data :

10	2000	5	54574	115	111111
----	------	---	-------	-----	--------

Here, we can see that we have assigned a fixed width to every number of 8 spaces. It means that every number will take at least and at most 8 bytes. Now, if we want to access 5th number, we just need to jump (8x4) 32 bytes and from 33rd byte our 5th number will start.

Sample Program :

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

using namespace std;

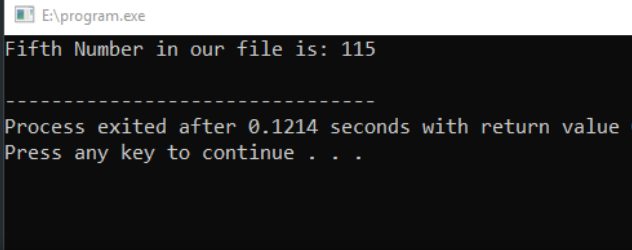
int main()
{
    int arr[] = {10, 2000, 5, 54574, 115, 111111}, n;
    fstream out("file.txt", ios :: out | ios :: in );

    for ( int i = 0; i < 6; i++ )
        out << setw(8) << arr[i];

    out << '\n';
    out.clear();

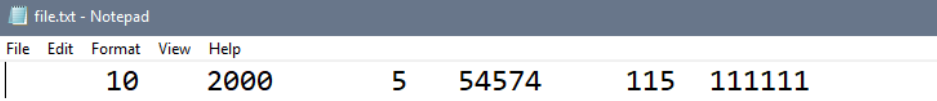
    out.seekg(8 * 4);
    out >> n;
    cout << "Fifth Number in our file is: ";
    cout << n << endl;

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



Output of the program:

```
Fifth Number in our file is: 115
-----
Process exited after 0.1214 seconds with return value
Press any key to continue . . .
```



Fixed Size Record :

Fixed sized records take more space to maintain size but give efficiency while reading or manipulating the data in the file.

Self Assessment :

Consider a file containing 20 numbers with a fixed width of 8 spaces for each number. One of the numbers is mistakenly written 431, Write a code to replace the number 431 by 413. The number can occur anywhere in the file.

Solution :

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

using namespace std;

int main()
{
    int arr[] = {10, 2000, 5, 54574, 115, 111111, 123, 431, 87, 78, 34, 324, 23, 453, 23, 45, 543, 65, 54,
    int position;

    fstream out("file.txt", ios :: out | ios :: in );

    for ( int i = 0; i < 20; i++ )
        out << setw(8) << arr[i];

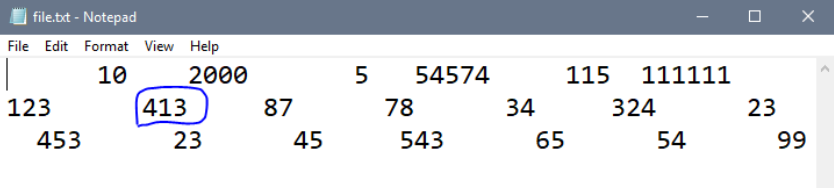
    out << '\n';
    out.clear();

    out.seekg(0, ios :: beg);

    for ( int i = 0; i < 20; i++ )
    {
        out >> setw(8) >> n;

        if ( n == 431 )
        {
            position = out.tellp();
            out.seekp(position - 8, ios :: beg);
            out << setw(8) << 413;
            break;
        }
    }

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



Here, tellp() tells the position of the reading pointer which will be right after 431 as it is already read. So we moved the writing pointer 8 positions (bytes) backward to the reading pointer and over-written the number by 413.

Text Files :

Till now, we were working with only text files. Text files are human readable files and these are portable. It means that we can open text files in almost all of the platforms alike. These files are with extensions .txt .c .cpp etc. Now, we are going to work with binary files. Binary files contain data in binary format, i.e in the form of 0s and 1s. So, these files are not human readable. Binary files may or may not be compatible with other platforms. It means that a binary file created with C++ file may or may not work with other languages. On the other hand, binary files are much faster than the text files, because in case of text files, all the data is already in the binary form, so no transformation of data is required when reading and writing from the memory. So, it directly reads the data from the specified address of the memory. We just need to give the address and the size of the data which is to be fetched from the memory.

Format in Binary Files :

Data is already formatted in binary format, we don't need to do anything. We know that, at binary level we deal with the bytes. So, in C++ the size of the integer is 4 bytes. It will take 4 bytes in the binary file as well no matter if we are saving 5 or 500, it will take fixed size.

Writing Class object to the file :

We can write any object to the file as well, by just saving its data members in the file. Here, we are going to write an object of a class in both text and binary files.

```

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

class Sample
{
    int x, y;
    float f;
    double d;
public:
    Sample()
    {
        x = 10; y = 20; f = 12.213; d = 421.123123;
    }

    void writeInFile( ostream& file )
    {
        file << setw(8) << x;
        file << setw(8) << y;
        file << setw(8) << setprecision(2) << fixed << f;
        file << setw(8) << setprecision(2) << fixed << d;
    }
};

int main()
{
    fstream txtFile, binFile;
    txtFile.open("file.txt", ios :: out);
    binFile.open("file.bin", ios :: out);

    Sample s;

    binFile.write( (char*) &s, sizeof(s) );
    s.writeInFile(txtFile);

    txtFile.close();
    binFile.close();
    return 0;
}

```

file.txt - Notepad

File	Edit	Format	View	Help
10	20	12.21	421.12	

If we compare the size of text and binary files we can find out something interesting.

The image displays two Windows file property windows and a Notepad window. The 'file.bin Properties' window shows a file of type 'BIN File (.bin)' with a size of 25 bytes. The 'file.txt Properties' window shows a 'Text Document (.txt)' of 32 bytes. A Notepad window titled 'file.txt - Notepad' contains the text '1.123123;'. Handwritten in blue ink over the Notepad window is a calculation: '10' over '8', '20' over '8', '12.21' over '8', and '421.12' over '8', followed by '= 32B'. This illustrates that the text file's size is determined by the number of characters (including spaces) it contains, which in this case is 32 characters, each taking 1 byte.

file.bin Properties

General Security Details Previous Versions

file.bin

Type of file: BIN File (.bin)

Opens with: Pick an app Change...

Location: E:\

Size: 25 bytes (25 bytes)

Size on disk: 0 bytes

Created: Today, December 8, 2021, 4 minutes ago

Modified: Today, December 8, 2021, 1 minute ago

Accessed: Today, December 8, 2021, 1 minute ago

Attributes: ☐ Read-only ☐ Hidden Advanced...

OK Cancel Apply

file.txt Properties

General Security Details Previous Versions

file.txt

Type of file: Text Document (.txt)

Opens with: Notepad Change...

Location: E:\

Size: 32 bytes (32 bytes)

Size on disk: 0 bytes

Created: Today, December 8, 2021, 1 hour ago

Modified: Today, December 8, 2021, 1 minute ago

Accessed: Today, December 8, 2021, 1 minute ago

Attributes: ☐ Read-only ☐ Hidden Advanced...

OK Cancel Apply

file.txt - Notepad

File Edit Format View Help

1.123123;

10 20 12.21 421.12

8 8 8 8 = 32B

Here, the size of the text file is fixed with the number of spaces it will take, in this case it is taking 32 spaces and the size of the file is 32 bytes. But, how the size of a binary file is calculated, we will discover this myth in the next lecture. Till then Stay tuned. Stay HAPPY and SAFE.

-THE END-