8.5a — Constructor member initializer lists

BY ALEX ON NOVEMBER 13TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 12TH, 2016

In the previous lesson, for simplicity, we initialized our class member data in the constructor using the assignment operator. For example:

```
class Something
{
private:
    int m_value1;
    double m_value2;
    int *m_value3;

public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 0;
        m_value2 = 0.0;
        m_value3 = 0;
    }
};
```

When the class's constructor is executed, m_value1, m_value2, and m_value3 are created. Then the body of the constructor is run, where the member data variables are assigned values. This is similar to the flow of the following code in non-object-oriented C++:

```
int m_value1;
double m_value2;
int *m_value3;

m_value1 = 0;
m_value2 = 0.0;
m_value3 = 0;
```

While this is valid within the syntax of the C++ language, it does not exhibit good style (and may be less efficient than initialization).

However, as you have learned in previous lessons, some types of data (e.g. const and reference variables) must be initialized on the line they are declared. Consider the following example:

```
class Something
{
private:
    const int m_value;

public:
    Something()
    {
        m_value = 5; // error: const vars can not be assigned to
    }
};
```

This produces code similar to the following:

```
const int m_value; // error: const vars must be initialized with a value
m_value = 5; //  error: const vars can not be assigned to
```

Assigning values to const or reference member variables in the body of the constructor is clearly not sufficient in some cases.

Member initializer lists

To solve this problem, C++ provides a method for initializing class member variables (rather than assigning values to them after they are created) via a **member initializer list** (often called a "member initialization list"). Do not confuse these with the similarly named initializer list that we can use to assign values to arrays.

In lesson **2.1 -- Fundamental variable definition, initialization, and assignment**, you learned that you could initialize variables in three ways: explicitly, implicitly, and via uniform initialization (C++11 only).

```
int value1 = 5; // explicit initialization
double value2(4.7); // implicit initialization
char value3 {'a'} // uniform initialization
```

Using an initialization list is almost identical to doing implicit initialization (or uniform initialization in C++11).

This is something that is best learned by example. Revisiting our code that does assignments in the constructor body:

```
class Something
{
private:
    int m_value1;
    double m_value2;
    int *m_value3;

public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 0;
        m_value2 = 0.0;
        m_value3 = 0;
    }
};
```

Now let's write the same code using an initialization list using implicit initialization:

```
class Something
{
private:
    int m_value1;
    double m_value2;
    int *m_value3;

public:
    Something() : m_value1(0), m_value2(0.0), m_value3(0) // Implicitly initialize our member
variables
    {
    // No need for assignment here
    }
};
```

The member initializer list is inserted after the constructor parameters. It begins with a colon (:), and then lists each variable to initialize along with the value for that variable separated by a comma. Note that we no longer need to do the assignments in the constructor body, since the initializer list replaces that functionality. Also note that the initializer list does not end in a semicolon.

Here's an example of a class that has a const member variable:

```
class Something
{
private:
    const int m_value;

public:
    Something(): m_value(5) // Implicitly initialize our member variables
    {
```

```
    }
};
```

Rule: Use member initializer lists to initialize your class member variables instead of assignment.

Uniform initialization in C++11

In C++11, instead of implicit initialization, uniform initialization can be used:

```
class Something
{
private:
    const int m_value;

public:
    Something(): m_value { 5 } // Uniformly initialize our member variables
    {
    }
};
```

We strongly encourage you to begin using this new syntax (even if you aren't using const or reference member variables) as initialization lists are required when doing composition and inheritance (subjects we will be covering shortly).

Rule: Favor uniform initialization over implicit initialization if you compiler is C++11 compatible

Initializing array members with member initializer lists

Consider a class with an array member:

```
class Something
{
private:
    const int m_array[5];

};
```

Prior to C++11, you can only zero an array member via a member initialization list:

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array {} // zero the member array
    {
        // If we want the array to have values, we'll have to use assignment here
    }

};
```

However, in C++11, you can fully initialize a member array using uniform initialization:

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array { 1, 2, 3, 4, 5 } // use uniform initialization to initialize our mem
ber array
    {
    }

};
```

Summary

Member initializer lists allow us to initialize our members rather than assign values to them. This is the only way to initialize members that require values upon initialization, such as const or reference members, and it can be more performant than assigning values in the body of the constructor.

Quiz Time

1) Write a class named RGBA that contains 4 member variables of type uint8_t named m_red, m_green, m_blue, and m_alpha (#include cstdint to access type uint8_t). Assign default values of 0 to m_red, m_green, and m_blue, and 255 to m_alpha. Create a constructor that uses a member initializer list that allows the user to initialize values for m_red, m_blue, and m_green (all three must be provided), and optionally m_alpha. Include a print() function that outputs the value of the member variables.

If you need a reminder about how to use the fixed width integers, please review lesson **2.4a -- Fixed-width integers and the unsigned controversy**.

Hint: If your print() function isn't working correctly, make sure you're casting uint8_t to an int.

The following code should run:

```
int main()
{
    RGBA teal(0, 127, 127);
    teal.print();

    return 0;
}
```

and produce the result:

```
r=0 g=127 b=127 a=255
```

Show Solution


8.5b -- Non-static member initialization


Index


8.5 -- Constructors

Share this:

C++ TUTORIAL | PRINT THIS POST

35 comments to 8.5a — Constructor member initializer lists

srividya
October 16, 2009 at 12:37 am · Reply

Its a just an awesome example

vvidhu1988
April 24, 2011 at 2:26 am · Reply

hi i have a question. Let say there is class A and Class B

```
Class A {
 int value,
 B bMember,

 Public:
  A() {
   value = 0;
  }
};
```

here when i create an object to Class A say

```
A aMember;
```

Is the constructor of class A is called and initializes ‘value’ to 0 or before calling Class A’s constructor, whether Calss B’s constructor will be called? since we also have object(bmember) of class type B.

Please explain when class B’s constructor will be called.

Whether in the above code i need to initialize ‘bMember’ of class type B in the Class A’s constructor?? since bMember is also a member in class A. I don’t think it is a proper implementation. Can anyone explain?

Nikhil Singhal
June 29, 2011 at 4:23 am · Reply

Hi,

The Ans is Very Simple……

B constructor first called, because first declaration part is done then A's Constructor is called, so here B's object is created first.
I think you got the answer
if have any confusion run the following code in debug mode:

```
class B {
    int BValue;
public:
    B()
    {
        BValue = 0;
        printf("n Class B Constructor is Called");
    }
};
class A {
    int value;
    B bMember;

public:
    A()
    {
    value = 0;
    printf("n Class A Constructor is Called n ");
    }
};


int main(int argc, char* argv[])
{
    A a1;
    return 0;
}
```

MrPlow442
June 28, 2012 at 4:35 pm · Reply

One question.

Do i have to use body brackets {}?
Can't i just write?
Something() : m_nValue(0), m_dValue(0.0), m_pnValue(0) ; ?

Sam Ebenezer
August 15, 2014 at 6:29 pm · Reply

Yes, the starting and ending curly braces are MANDATORY!!

Alex
December 21, 2015 at 11:53 am · Reply

Yep, the curly braces define the body of the constructor. If you don't provide them, the compiler will get confused.

petewguy1234
August 14, 2015 at 4:10 am · Reply

Good explanation! Thank you!

Abhijeet
October 6, 2015 at 5:44 am · Reply

Hi Alex,

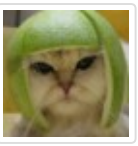
You've used

```
variable(value)
```

to initialize the variables. There is also the option of using

```
variable{value}
```

Is there any inherent benefit to using one over the other?



Alex
October 6, 2015 at 4:11 pm · Reply

variable(value) works in all versions of C++, but has the downside of doing implicit conversions. variable { value } is the new uniform initialization pattern established in C++11. If your compiler is C++11 capable, the { } pattern is mostly better because it won't do implicit conversions.

This is covered in lesson **2.1 -- Basic addressing and variable declaration**.

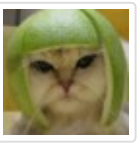
Jason
October 21, 2015 at 5:43 pm · Reply

Hi Alex,

Greatly enjoying your tutorial and very excited to see it completely updated. The pages you have recently worked on are truly epic.

I may have missed it, but why does using an initialization list resolve the issues that require const and reference member fields to be initialized at the time of their declaration?

Best wishes and thank you!



Alex
October 22, 2015 at 1:08 pm · Reply

> why does using an initialization list resolve the issues that require const and reference member fields to be initialized at the time of their declaration?

Because the initialization list provides initialization values for the member variables, whereas assigning values to the members in the body of the constructor is considered an assignment (not an initialization, even though the constructor itself is called to initialize the class object). As you're aware, const variables and references require that they be initialized.

chiva
December 16, 2015 at 7:07 am · Reply

Hello I am a new one here. I am studying in university now.
Now let's me release my questions that I don't understand.
I would like to understand well on How to make Array Initialize using with default constructor.
Please share and explain

I thank you all in advanced

Alex
December 17, 2015 at 1:29 pm · Reply

Prior to C++11, I don't think there's a way to initialize a member array.

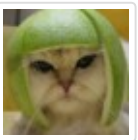
In C++11, you can use uniform initialization:

```
class Foo
{
private:
    int m_x[5]; // member array

public:
    Foo() : m_x { 1, 2, 3 }
    {

    }
};
```

v.kurenkov
December 22, 2015 at 6:51 am · Reply

Good Day!
Your result ain't right (r=0 g=0 b=0 a=255), there should be r=0 g=127 b=127 a=255

Thank you, site's just awesome!



Alex
December 23, 2015 at 9:52 am · Reply

Yup, thanks for pointing that out. Fixed!

Pisces
December 27, 2015 at 4:20 am · Reply

Hello and thank you for these excellent tutorials! This must be the best learning site ever.

Could you elaborate if default values initialize or assign variables? What will happen in this case?:

```
class Something
{
private:
    int m_value1 = 1;
    double m_value2 = 1.0;
    int *m_value3 = 0;

public:
    Something() : m_value1(0), m_value2(0.0), m_value3(0)
    {
    }
};
```

Thanks!

Jeff
December 27, 2015 at 9:20 pm · Reply

I'd like to add to this question. In the quiz solution, if I understand correctly, it seems the user can also safely instantiate "teal" with

```
RGBA teal;
```

since all members have default values, is that correct?

Alex
December 28, 2015 at 4:02 pm · Reply

```
RGBA teal;
```

would call the default constructor, which would assign m_red, m_green, and m_blue the value 0. This would be the color black, not teal.

Alex
December 28, 2015 at 3:35 pm · Reply

Initialize. However, if your constructor has an initializer list that gets executed, those will take precedence.

So, in your example case, assuming your Something() default constructor is called, your Something object will end up with values 0, 0.0, and 0 respectively.

Nicolas
January 11, 2016 at 5:43 pm · Reply

Dear Alex,

First of all, thank you so much for your tutorials, they have helped me a lot! The new updates also help a lot. The quizzes are a great learning tool.

I am using a library called GDAL (I already installed it properly and tested it). Now I am trying to create some classes that use some other classes from GDAL. I first created the LC_Raster and it worked fine. Then I created the LC_Product and I get this error: "error C2512: ‘LC_Raster::LC_Raster’ : no appropriate default constructor available"

I would appreciate a lot if you could please give me some insight about the mistake I am making, since I have spent hours trying to find it without success.

I am using Visual Studio 2010, so I think it does not have C++11 and that is why the initialization of std::vector is weird. In a few days I will have my usual computer back and I would be able to use Visual Studio 2015.

Here is the code:

```

#include “stdafx.h”
#include
#include
#include
#include
#include
#include
#include “gdal_priv.h”
#include “gdal_alg.h”
#include “cpl_conv.h”
#include “cpl_string.h”
#include
#include
```

```
#include

class LC_Raster
{
private:
std::string m_parentProductName;
int m_year;
std::string m_rasterName;
GDALDataset* m_SourceRasterDS;

public:
LC_Raster(std::string parentProductName, int year): m_parentProductName(parentProductName), m_year(year)
{
std::stringstream os;
os <> yearAsString;

m_rasterName = “Recon_” + m_parentProductName + “_” + yearAsString + “.tif”;

m_SourceRasterDS = (GDALDataset*) GDALOpen(m_rasterName.c_str(),GA_ReadOnly);
}

void setProductName (std::string name)
{
m_parentProductName = name;
}

void setRasterYear (int year)
{
m_year = year;
}

std::string getParentProductName () {return m_parentProductName; }
int getRasterYear () {return m_year; }
std::string getFileName () {return m_rasterName; }
GDALDataset* getRasterGDAL () {return m_SourceRasterDS; }
};

class LC_Product
{
private:
std::string m_productName;
std::vector m_productYears;
int m_numberOfRasetrs;
std::vector m_rastersOfProduct;

public:

LC_Product(std::string productName, std::vector productYears): m_productName(productName),
m_productYears(productYears)
{
m_numberOfRasetrs = productYears.size();
m_rastersOfProduct.resize(productYears.size());

for (int iii=0; iii < productYears.size(); iii++)
{
m_rastersOfProduct[iii] = LC_Raster(productName,productYears[iii]);
}

}

std::string getProductName () {return m_productName; }
std::vector getProductYears () {return m_productYears; }
```

```
int getNumberOfRasters () {return m_numberOfRasetrs; }

std::vector calculateNeighboringYears (int yearToPredict, int deltaT)
{
std::vector neighborYears;
int numberOfNeighboringYears = neighborYears.size(); //This is the same as 0

for (int iii=0; iii = (yearToPredict-deltaT)) && (m_productYears[iii] <= (yearToPredict+deltaT)))
{
numberOfNeighboringYears += 1;
neighborYears.resize(numberOfNeighboringYears);
neighborYears[numberOfNeighboringYears-1] = m_productYears[iii];
}
}

return neighborYears;
}
};

int _tmain(int argc, _TCHAR* argv[])
{
GDALAllRegister();

//Array with the name of the products
std::array productNames = {“BC”,”Hist”,”LF”,”MODIS”,”NLCD”,”NLCD92″};

//Vectors with the year of eahc product
std::vector yearsBC;
for (int yearBC = 1938; yearBC <= 2005; yearBC++)
{
yearsBC.push_back(yearBC);
}
std::vector yearsHist (1,1977);
int arrayYearsLF[] = {2001,2008,2010,2012};
std::vector yearsLF (arrayYearsLF, arrayYearsLF + sizeof(arrayYearsLF) / sizeof(int) );
std::vector yearsMODIS;
for (int yearMODIS = 2001; yearMODIS <= 2012; yearMODIS++)
{
yearsMODIS.push_back(yearMODIS);
}
int arrayYearsNLCD[] = {2001,2006,2011};
std::vector yearsNLCD (arrayYearsNLCD, arrayYearsNLCD + sizeof(arrayYearsNLCD) / sizeof(int) );
std::vector yearsNLCD92 (1,1992);

LC_Product NLCD(productNames[4], yearsNLCD);

LC_Raster rasterTrial(“NLCD”,2001);

std::cout << rasterTrial.getFileName() <GetGeoTransform(GeoTransform);

for (int iii=0; iii < 6; iii++)
{
std::cout << GeoTransform[iii] << "n";
}

return 0;
}
```

```
1
```

I apologize, I could not figure how to format the code.

Thank you so much for your attention and time.

Nicolas

January 12, 2016 at 6:33 am · Reply

Here is the code in the appropriate format, I apologize again.

```
#include "stdafx.h"
#include <string>
#include <vector>
#include <array>
#include <iostream>
#include <cassert>
#include <sstream>
#include "gdal_priv.h"
#include "gdal_alg.h"
#include "cpl_conv.h"
#include "cpl_string.h"
#include <ogr_spatialref.h>
#include <cmath>
#include <math.h>


class LC_Raster
{
private:
    std::string m_parentProductName;
    int m_year;
    std::string m_rasterName;
    GDALDataset* m_SourceRasterDS;

public:
    LC_Raster(std::string parentProductName, int year): m_parentProductName(parentPr
oductName), m_year(year)
    {
        std::stringstream os;
        os << m_year;

        std::string yearAsString;
        os >> yearAsString;

        m_rasterName = "Recon_" + m_parentProductName + "_" + yearAsString + ".tif";

        m_SourceRasterDS = (GDALDataset*) GDALOpen(m_rasterName.c_str(),GA_ReadOnl
y);
    }

    void setProductName (std::string name)
    {
        m_parentProductName = name;
    }

    void setRasterYear (int year)
    {
        m_year = year;
    }

    std::string getParentProductName () {return m_parentProductName; }
    int getRasterYear () {return m_year; }
    std::string getFileName () {return m_rasterName; }
    GDALDataset* getRasterGDAL () {return m_SourceRasterDS; }
};
```

```
class LC_Product
{
private:
    std::string m_productName;
    std::vector<int> m_productYears;
    int m_numberOfRasetrs;
    std::vector<LC_Raster> m_rastersOfProduct;

public:

    LC_Product(std::string productName, std::vector<int> productYears): m_productNam
e(productName), m_productYears(productYears)
    {
        m_numberOfRasetrs = productYears.size();
        m_rastersOfProduct.resize(productYears.size());

        for (int iii=0; iii < productYears.size(); iii++)
        {
            m_rastersOfProduct[iii] = LC_Raster(productName,productYears[iii]);
        }

    }

    std::string getProductName () {return m_productName; }
    std::vector<int> getProductYears () {return m_productYears; }
    int getNumberOfRasters () {return m_numberOfRasetrs; }

    std::vector<int> calculateNeighboringYears (int yearToPredict, int deltaT)
    {
        std::vector <int> neighborYears;
        int numberOfNeighboringYears = neighborYears.size(); //This is the same as 0

        for (int iii=0; iii < m_productYears.size(); iii++)
        {
            if ((m_productYears[iii] >= (yearToPredict-deltaT)) && (m_productYears[i
ii] <= (yearToPredict+deltaT)))
            {
                numberOfNeighboringYears += 1;
                neighborYears.resize(numberOfNeighboringYears);
                neighborYears[numberOfNeighboringYears-1] = m_productYears[iii];
            }
        }

        return neighborYears;
    }
};




int _tmain(int argc, _TCHAR* argv[])
{
    GDALAllRegister();

    //Array with the name of the products
    std::array<std::string,6> productNames = {"BC","Hist","LF","MODIS","NLCD","NLCD9
2"};

    //Vectors with the year of eahc product
    std::vector<int> yearsBC;
    for (int yearBC = 1938; yearBC <= 2005; yearBC++)
    {
        yearsBC.push_back(yearBC);
    }
    std::vector<int> yearsHist (1,1977);
```

```
127     int arrayYearsLF[] = {2001,2008,2010,2012};
128     std::vector<int> yearsLF (arrayYearsLF, arrayYearsLF + sizeof(arrayYearsLF) / si
129 zeof(int) );
130     std::vector<int> yearsMODIS;
131     for (int yearMODIS = 2001; yearMODIS <= 2012; yearMODIS++)
132     {
133         yearsMODIS.push_back(yearMODIS);
134     }
135     int arrayYearsNLCD[] = {2001,2006,2011};
136     std::vector<int> yearsNLCD (arrayYearsNLCD, arrayYearsNLCD + sizeof(arrayYearsNL
137 CD) / sizeof(int) );
138     std::vector<int> yearsNLCD92 (1,1992);
139
140
141     LC_Product NLCD(productNames[4], yearsNLCD);
142
143
144
145
146
147     LC_Raster rasterTrial("NLCD",2001);
148
149     std::cout << rasterTrial.getFileName() << "\n";
150
151     GDALDataset* SourceRasterDS = rasterTrial.getRasterGDAL();
152     double GeoTransform[6];                         // Get the geotransform object
153     SourceRasterDS->GetGeoTransform(GeoTransform);
154
    for (int iii=0; iii < 6; iii++)
    {
        std::cout << GeoTransform[iii] << "\n";
    }

    return 0;
}
```

Nicolas
January 12, 2016 at 7:24 am · Reply

I got the code to build and work by creating a default constructor for LC_Raster and a default constructor for LC_Product. However, I don't understand why that is necessary since I never call the default constructors in main.

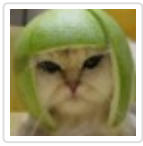
Alex
January 12, 2016 at 2:14 pm · Reply

Taking a quick look at this, I'm not sure why you'd be getting an error about not having an appropriate default constructor. As far as I can see, in both places where you're declaring an object of type LC_Raster, you're doing so with parameters. What line of code is it complaining about?

Nicolas
January 12, 2016 at 4:00 pm · Reply

Dear Alex,

Thank you for your help and time. I found the issue. In the constructor of LC_Product I have a std::vector m_rastersOfProduct containing LC_Raster variables. Te vector is resized, and as soon as it is resized, it tries to call the default constructor of LC_raster, which does not exist. Thus, the solution is to not resize the vector and instead, push_back the elements, or to create a default constructor for LC_raster.



Alex
January 13, 2016 at 12:29 pm · Reply

Ah, makes sense. Glad you were able to figure it out. Thanks for sharing.

nick
January 11, 2016 at 10:13 pm · Reply

[code]
#include <iostream>
#include <cstdint>
using namespace std;
class RGBA {
public:
RGBA (uint8_t red,uint8_t green,uint8_t blue)
:m_red(0),m_green(0),m_blue(0),m_alfa(255){
m_red=red;
m_green=green;
m_blue=blue;
}
void print() {cout<<"R : "<<(int)m_red<<"\nG : "<<(int)m_green<<"\nB : "<<(int)m_blue<<"\nA : "<<(int)m_alfa;
};
private:
uint8_t
m_red,
m_green,
m_blue,
m_alfa;
};
int main(){
RGBA o(0,127,127);
o.print();
return 0;
}
[/cdoe]

cei
February 5, 2016 at 1:48 pm · Reply

Hi. I am a bit confused: is it better to use member initializer lists or directly assigning a default value to member variables via non-static member initialization?
Up until now I have always used the former, but now it seems to me more convenient to use the latter, as it makes a class easier to understand (no need to jump to the member initializer list to see what the initial values of the class' members are).

Alex
February 5, 2016 at 2:00 pm · Reply

Non-static member initialization is preferred, if you have a compiler that supports it, for precisely the reason you've identified.

You can still use member initialization lists for any values the constructor needs to change from the default value.

Lokesh
February 8, 2016 at 1:10 am · Reply

At the beginning of this lesson, the names of variables used in code and those used to explain the code below it are different.
"When the class's constructor is executed, m_nValue, m_dValue, and m_chValue are created." It should be:
"When the class's constructor is executed, m_value1, m_value2, and m_value3 are created."

Alex
February 9, 2016 at 10:27 am · Reply

Fixed, thanks.

Lokesh
February 8, 2016 at 1:39 am · Reply

In section "Initializing array members with member initializer lists", you said that prior to C++11, you could only zero the elements using a blank list for initializing the members of an array. Following is the example code provided:

```
1  class Something
2  {
3  private:
4      const int m_array[5];
5
6  public:
7      Something(): m_array {} // zero the member array
8      {
9          // If we want the array to have values, we'll have to use assignment here
10     }
11
12 };
```

However, since the array elements are of type <const int>, we cannot assign values to them once we have initialized them(in this case with zero for all elements). So, my question is how did we initialize elements of an integer array of type <const int> prior to C++11 when an instance of a class is created ?

Alex
February 9, 2016 at 10:29 am · Reply

Good question. Prior to C++11, it wasn't possible!

You could work around by using std::vector though (since the std::vector constructor could be used to initialize a const std:vector).

Lokesh
February 10, 2016 at 6:03 am · Reply

Thanks.

Lokesh
February 8, 2016 at 3:31 am · Reply

I think you should not use the words "explicit assignment" as there is no such thing as implicit assignment. It is more confusing since we use the terms implicit and explicit when referring to initialization (uniform

initialization is also implicit which happens to work for all data types). You should say just "assignment". I am referring to the sentence:
"Note that we no longer need to do the explicit assignments in the constructor body, …"

Alex
February 9, 2016 at 11:09 am · Reply

Agreed. The article has been updated.