C++ CLASS CONSTRUCTOR AND DESTRUCTOR

http://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

The Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;
   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization Lists to Initialize Fields:

In case of parameterized constructor, you can use following syntax to initialize the fields:

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

Above syntax is equal to the following syntax:

```
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows:

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

The Class Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde   and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

8.5 — Constructors

BY ALEX, ON SEPTEMBER 5TH, 2007

Constructors

A **constructor** is a special kind of class member function that is executed when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

Unlike normal functions, constructors have specific rules for how they must be named:
1) Constructors should always have the same name as the class (with the same capitalization)
2) Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

Here is an example of a class that has a default constructor:

```cpp
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    Fraction() // default constructor
    {
         m_nNumerator = 0;
         m_nDenominator = 1;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

This class was designed to hold a fractional value as an integer numerator and denominator. We have defined a default constructor named Fraction (the same as the class). When we create an instance of the Fraction class, this default constructor will be called immediately after memory is allocated, and our object will be initialized. For example, the following snippet:

```cpp
Fraction cDefault; // calls Fraction() constructor
std::cout << cDefault.GetNumerator() << "/" << cDefault.GetDenominator() << std::endl;
```

produces the output:

```
0/1
```

Note that our numerator and denominator were initialized with the values we set in our default constructor! This is such a useful feature that almost every class includes a default constructor. Without a default constructor, the numerator and denominator would have garbage values until we explicitly assigned them reasonable values.

Constructors with parameters

While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values. Fortunately, constructors can also be declared with parameters. Here is an example of a constructor that takes two integer parameters that are used to initialize the numerator and denominator:

```cpp
#include <cassert>
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    Fraction() // default constructor
    {
         m_nNumerator = 0;
         m_nDenominator = 1;
    }

    // Constructor with parameters
    Fraction(int nNumerator, int nDenominator=1)
    {
        assert(nDenominator != 0);
        m_nNumerator = nNumerator;
        m_nDenominator = nDenominator;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

Note that we now have two constructors: a default constructor that will be called in the default case, and a second constructor that takes two parameters. These two constructors can coexist peacefully in the same class due to function overloading. In fact, you can define as many constructors as you want, so long as each has a unique signature (number and type of parameters).

So how do we use this constructor with parameters? It's simple:

```cpp
Fraction cFiveThirds(5, 3); // calls Fraction(int, int) constructor
```

This particular fraction will be initialized to the fraction 5/3!

Note that we have made use of a default value for the second parameter of the constructor with parameters, so the following is also legal:

```cpp
Fraction Six(6); // calls Fraction(int, int) constructor
```

In this case, our default constructor is actually somewhat redundant. We could simplify this class as follows:

```cpp
#include <cassert>
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    // Default constructor
    Fraction(int nNumerator=0, int nDenominator=1)
    {
        assert(nDenominator != 0);
        m_nNumerator = nNumerator;
        m_nDenominator = nDenominator;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

This constructor has been defined in a way that allows it to serve as both a default and a non-default constructor!

```cpp
Fraction cDefault; // will call Fraction(0, 1)
Fraction cSix(6); // will call Fraction(6, 1)
Fraction cFiveThirds(5,3); // will call Fraction(5,3)
```

Classes without default constructors

What happens if we do not declare a default constructor and then instantiate our class? The answer is that C++ will allocate space for our class instance, but will not initialize the members of the class (similar to what happens when you declare an int, double, or other basic data type). For example:

```cpp
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;
};

int main()
{
    Date cDate;
    // cDate's member variables now contain garbage
    // Who knows what date we'll get?

    return 0;
}
```

In the above example, because we declared a Date object, but there is no default constructor, m_nMonth, m_nDay, and m_nYear were never initialized. Consequently, they will hold garbage values. Generally speaking, this is why providing a default constructor is almost always a good idea:

```cpp
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

public:
    Date(int nMonth=1, int nDay=1, int nYear=1970)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
};

int main()
{
    Date cDate; // cDate is initialized to Jan 1st, 1970 instead of garbage

    Date cToday(9, 5, 2007); // cToday is initialized to Sept 5th, 2007

    return 0;
}
```