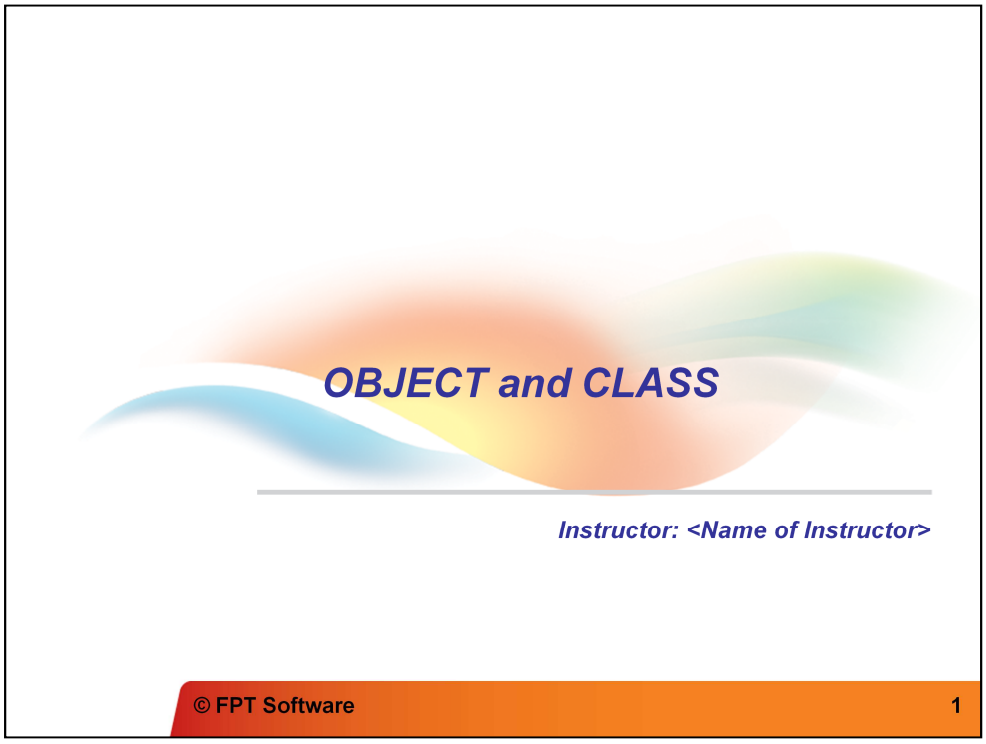


Latest updated by: HanhTT1

Introduction to class

- Whether using C++ built-in data types or redefinitions, so far all of our variables were declared with simple data types, also called primitive types. This made is easy to recognize that, for example, the number of fingers of a hand could be represented with a simple natural number, or that a simple word made of a group of alphabetical characters could be used to name a hand.

Introduction to class (2)

- If some cases, you may want a group of values to be considered as one entity to represent an object. For example, you may want to use a single data type to represent a hand. C++ allows you to use one or more data types, group them as one, to create a new data type as you see fit. This technique of grouping different values is referred to as a class.
- C++ offers three techniques of defining a new data type: a structure, a class, and a union. These are also referred to as composite data types

Declare a class

```
class CRectangle {
    private:
        int x, y;
    protected:
        int k;
    public:
        int num;
        void set_values (int,int);
        int area ();
};
int main ()  {
    CRectangle rect;
    return 0;
}
```

Access to a member of a class

```
class CRectangle {
    private:
        int x, y;
    protected:
        int k;
    public:
        int num;
        void set_values (int,int);
        int area ();
};

int main ()  {
    CRectangle rect;
    rect.num;                       //OK
    int i = rect.area(); //OK
    rect.x;             //Error
    rect.k;             //Error

    return 0;
}
```

Static member variables

```
class MyClass {
public:
	MyClass(); // Constructor
	void PrintCount();  // Output current value of count
	static int count; // static member to store
};
MyClass::MyClass() {
	this->count++; // Increment the static count
}
void MyClass::PrintCount() {
	cout << "There are currently " << this->count << " instance(s) of MyClass.\n";
}
int MyClass::count = 10;
int main() {
	MyClass* x = new MyClass;
	x->PrintCount();
	MyClass* y = new MyClass;
	x->PrintCount();

	getch();
}
```

Result ????

Constant Object

- For a constant object, can not change values of any member in the object

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class MyClass {
public:
	MyClass(); // Constructor
public:
	int count;
};
MyClass::MyClass() {
	count = 10;
}
int main() {
	const MyClass *x = new MyClass();
	x->count = 20;		//Error, can not change value

	getch();
}
```

Methods of a class

```
#include <iostream>
using namespace std;
class Rectangle {
    public:
        double Length;
        double Height;
    public:
        double Perimeter();
        double Area();
};
int main( int argc, char * argv[]) {
    Rectangle rect = { 48.35, 36.94 };
    cout << "Rectangle Characteristics";
    cout << "\nLength: " << rect.Length;
    cout << "\nHeight: " << rect.Height;
    return 0;
}
```

Method implementation

- When a method is a class' member, it has access to the member variables of the same class. This means that you don't need to pass the variables as arguments: you can just use any of them as if it were supplied.

```
class Rectangle {
public:
        double Length;
        double Height;
public:
        double Perimeter() {
                double p;
                p = (Length + Height) * 2;
                return p;
        }
        double Area();
};
int main( int argc, char * argv[] )  {
        Rectangle rect = { 48.35, 36.94 };
        cout << "\nPerimeter: " << rect.Perimeter();
        return 0;
}
```

Static methods

```
class Car {
public:
    string Make;
    string Model;
    int Year;
    unsigned NumberOfDoors;
    long Mileage;
public:
    static int CarsCount;
    static string ApplicationTitle;
    static void Show()  {
            Car vehicle;
            vehicle.Year = 2004;
            cout << "\nYear: " << vehicle.Year;
    }
};
int main( int argc, char * argv[] )  {
    cout << "\n - Vehicle Characteristics -";
    Car::Show();
    cout << "\nCount: " << Car::CarsCount << endl;
    return 0;
}
```

Constant methods

```
class CRectangle {
    private:
        int x;
        void test();
    protected:
        int num1, num2;
    public:
        int num;
        void set_values (int,int);
        int area () const;
};

int CRectangle::area() const {
    x = x + 1;          //Error,  do not modify member's variables
    return x;
}
```

Class header file

- Class header file: The building block or foundation of an class is made of the class' creation, listing all of its members. The header file has an extension of .h

```
class Box {
public:
    double Length;
    double Width;
    double Height;
private:
    const double FaceArea() const;
    const double TopArea() const;
    const double RightArea() const;
public:
    void Show();
};
```

Class source code

- The file used to implement the class is called a source file. Source file has an extension with .cpp, this file is usually created in the same project, it is specified with an include line that encloses the file name with double-quotes.

  Example: #include "Book.h"

```
#include <iostream>
#include "box.h"
using namespace std;
const double Box::FaceArea() const  {
    return Length * Height;
}
const double Box::TopArea() const  {
    return Length * Width;
}
```

Constructor

- A constructor is a special method that is created when the object is created or defined. This particular method holds the same name as that of the object and it initializes the instance of the object whenever that object is created. The constructor also usually holds the initializations of the different declared member variables of its object. Unlike some of the other methods, the constructor does not return a value, not even void.

```
class MyClass {
public:
        MyClass(int, double); // Constructor
private:
        int count;
        double sal;
};
MyClass::MyClass(int m, double n) {
        count = m;
        sal = n;
}
int main()  {
        MyClass *x = new MyClass(10, 3000);
        getch();
}
```

Constructor Overloading

- Like an ordinary method, a construction can be overloaded. This means that you can have different constructors following the rules of overloading a function. Since we saw that a constructor can be used to initialize the member variables of its object, you can use multiple constructors to apply different initializations.

```
class MyClass {
public:
        MyClass();
        MyClass(int, double);
private:
        int count;
        double sal;
};
MyClass::MyClass(int m, double n)          {
        count = m;
        sal = n;
}
MyClass::MyClass()              {
}
int main()  {
        MyClass *x = new MyClass();
        MyClass y(10, 300);
        getch();
}
```

Destructor

- As opposed to a constructor, a destructor is called when a program has finished using an instance of an object. A destructor does the cleaning behind the scenes. Like the default constructor, the compiler always create a default destructor if you don't create one. Like the default constructor, a destructor also has the same name as its object. This time, the name of the destructor starts with a tilde.

```
class MyClass {
public:
        MyClass();
        ~MyClass();
private:
        int count;
        double sal;
};
MyClass::MyClass()              {
}
MyClass::~MyClass() {
}
int main()  {
        MyClass *x = new MyClass();
        getch();
}
```

Using Destructor

```
class MyClass {
public:
    MyClass();
    ~MyClass();
private:
    int num;
    int *p;
};
MyClass::MyClass() {
    num = 100;
    p = &num;
}
MyClass::~MyClass() {
    delete p;
}
int main()  {
    MyClass *x = new MyClass();
    getch();
}
```

Class and Function

- One of the most regular operations you will perform in your program consists of mixing objects and functions. Fortunately, C++ allows you to pass an object to a function or to return an object from a function. An object you use in a program can come from any source: an object built-in the operating system (part of the Win32 library), an object shipped with Borland C++ Builder, an object that is part of the C++ language, or one that you create.

```
class MyClass {
public:
    MyClass();
};
MyClass::MyClass() {
}
class MySchool {
public:
    MySchool( MyClass p);
private:
    MyClass pSchool;
};
MySchool::MySchool( MyClass p) {
    pSchool = p;
}
int main() {
    MyClass *pClass = new MyClass();
    MySchool *pSchool = new MySchool(*pClass);
    getch();
}
```

Returning object

- As a data type, you can use an object as returning value of a function. The issue this time is a little different. When returning an object, you should be familiar with the construction of the object, especially its constructor. Since you will mostly return an object as complete as possible, the returned variable is a constructor. Therefore, when building the object inside of the function, make sure that the object can be used as a variable.

```
class MyClass {
};
class MySchool  {
public:
        MySchool( MyClass p);
        MyClass Display();
private:
        MyClass pSchool;
};
MySchool::MySchool( MyClass p)          {
        pSchool = p;
}
MyClass MySchool::Display() {
        return pSchool;
}
```

Passing an Object by reference

- Since an object is a data type, it enjoys the features of the other, regular, variables. For example, instead of returning an object from a function, when this is not possible because of some circumstances, you can pass an object by reference. Like another variable, when an object is passed by reference, any alteration made on the object will be kept when the function exists

```
void ObtainDimensions(TCone& Tone) {
        double r, h;
        cout << "Specify the dimensions of the tent\n";
        cout << "Radius: ";
        cin >> r;
        cout << "Radius: ";
        cin >> h;
        Tone.setRadius(r);
        Tone.setHeight(h);
}
```

Operator Overloading

- Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables.
- When overloaded, the operators are implemented as functions using the operator keyword.

Assignment Operator Overloading

- The assignment operator works by giving the value of one variable to another variable of the same type or closely similar.
- The assignment operator is defined using the operator keyword followed by the = symbol and the traditional parentheses of every function. As a function, the operator must be declared in the body of the class, along with the other method members. In order for this function to be accessible outside of the class, it should be listed in the public section. Like any member method, the function should have a return value.

Assignment Operator Overloading (2)

- Example:

```
TRectangle& operator=(const TRectangle& Rect);
TRectangle& TRectangle::operator=(const TRectangle& Rect) {
        Length = Rect.Length;
        Height = Rect.Height;
        return *this;
}
```

Assignment Operator Overloading - Exp

```
class MyClass {
public:
    MyClass(int, int);
    MyClass& operator=(const MyClass& old);
    void Display();
private:
    int top, left;
};

MyClass::MyClass(int x, int y) {
    top = x;
    left = y;
}

MyClass& MyClass::operator =(const MyClass &old) {
    top = old.top;
    left = old.left;
    return *this;
}
```

Overloading the Arithmetic Operators

- In C++, the addition operation is applied with the + symbol. Depending on the circumstance, it can be considered a unary or a binary operator. It is used as a unary operator when applied to one variable.
- The addition binary operator is used on two variables to add their values. Suppose you have two variables A and B. To add their values, you would type A + B.

- Unary Operator
- Binary Operator

Unary Operator

```
class MyClass {
    friend MyClass operator+(const MyClass& x, const MyClass& y);
public:
    MyClass(int, int);
    MyClass operator+(const int num);
    void Display();
private:
    int top;
    int     left;
};
MyClass::MyClass(int x, int y) {
    top = x;
    left = y;
}
MyClass MyClass::operator +(const int num) {
    top = top + num;
    left = left + num;
    return *this;
}
```

Unary Operator - Cont

```
void MyClass::Display() {
	cout << "New left = " << left;
	cout << "New top = " << top;
}

int main() {
	int n = 1000;
	MyClass pOld(10, 100);
	pOld = pOld + n;
	pOld.Display();

	getch();
}
```

Binary Operator

```
class MyClass {
      friend MyClass operator+(const MyClass& x, const MyClass& y);
public:
      MyClass(int, int);
      MyClass& operator=(const MyClass& old);
      void Display();
private:
      int top;
      int       left;
};
MyClass::MyClass(int x, int y) {
      top = x;
      left = y;
}
MyClass& MyClass::operator =(const MyClass &old) {
      top = old.top;
      left = old.left;
      return *this;
}
```

Binary Operator - Cont

```
MyClass operator +(const MyClass &curr, const MyClass &old) {
    MyClass p(1,1);
    p.top = curr.top + old.top;
    p.left = curr.left + old.left;
    return p;
}
void MyClass::Display() {
    cout << "New left = " << left;
    cout << "New top = " << top;
}
int main() {
    MyClass pOld(10, 100);
    MyClass pNew(1, 1);
    pNew = pOld;
    pNew = pNew + pOld;
    pNew.Display();

    getch();
}
```

Class and Exception handling

- Exceptions are an integral and unavoidable part of the operating system and programming. One way you can handle them is to create classes whose behaviors are prepared to deal with abnormal behavior. There are two main ways you can involve classes with exception handling routines: classes that are involved in exceptions of their own operations and classes that are specially written to handle exceptions for other classes.
- The simplest way to take care of exceptions in classes is to use any normal class and handle its exceptions. If concerned with exceptions, the minimum thing you can do in your program is to make it "aware' of eventual exceptions. This can be taken care of by including transactions or other valuable processing in a try block, followed by a three-dot catch as in catch(...).

Class and Exception handling - Exp

```
class Tcalculator {
    public:
        double Operand1;
        double Operand2;
        char Operator;
};
int main(int argc, char* argv[]) {
    TCalculator Calc;
    double Result;
    // Request two numbers from the user
    cout << "This program allows you to perform an operation on two numbers\n";
    cout << "To proceed, enter two numbers\n";
    try {
        cout << "First Number: ";
        cin >> Calc.Operand1;
        cout << "Operator: ";
        cin >> Calc.Operator;
        cout << "Second Number: ";
        cin >> Calc.Operand2;
        // Make sure the user typed a valid operator
```

Class and exception handling - Exp

```
        if (Calc.Operator != '+' && Calc.Operator != '-' && Calc.Operator != '*' && Calc.Operator != '/')
                throw Calc.Operator;
        if (Calc.Operator == '/')
          if(Calc.Operand2 == 0)
                throw 0;
        switch (Calc.Operator) {
          case '+':
                Result = Calc.Operand1 + Calc.Operand2;
                break;
          case '-':
                Result = Calc.Operand1 - Calc.Operand2;
                break;
        }
    // Display the result of the operation
        cout << "\n" << Calc.Operand1 << " " << Calc.Operator << " " << Calc.Operand2 << " = " << Result;
    }
    catch(const char n) {
        cout << "\nOperation Error: " << n << " is not a valid operator";
    }
    catch(const int p) {
        cout << "\nBad Operation: Division by " << p << " not allowed";
    }
  return 0;
}
```

Friend relation

- Levels limit access right to members of a class:
  - ✓ public
  - ✓ protected
  - ✓ private
- To access protected and private members in a class, normally use methods to get its values.
- When two classes are declared friend, all members in a class may be acessed by members of left class.
- Friend relation can be declared between classes or functions.

Friend function

- No function outside a class can be access to private or protected members of that class except friend function

```
class CRectangle {
  private:
        int width, height;
  public:
        void set_values (int, int);
        int area ()
          {return (width * height);}
        friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
        width = a;
        height = b;
}
CRectangle duplicate (CRectangle rectparam) {
        CRectangle rectres;
        rectres.width = rectparam.width*2;
        rectres.height = rectparam.height*2;
        return (rectres);
}
```

```
int main () {
        CRectangle rect, rectb;
        rect.set_values (2,3);
        rectb = duplicate (rect);
        cout << rectb.area();
        return 0;
}
```

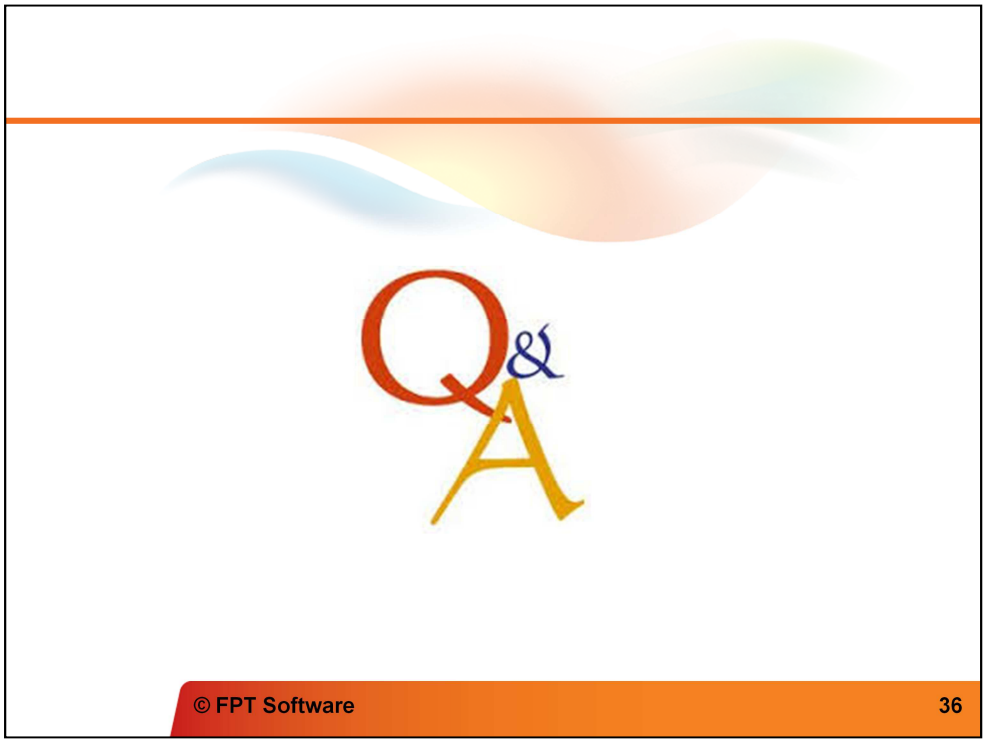
Friend class

- Whenever 2 classes declared friend, members of this class can be accessed to members of left class.

```
class CSquare;
class CRectangle {
  private:
            int width, height;
  public:
            int area ()
                {return (width * height);}
  void convert (CSquare a);
};

class CSquare {
  private:
            int side;
  public:
            void set_side (int a)
                {side=a;}
  friend class CRectangle;    //CRectange is CSquare's friend
};
void CRectangle::convert (CSquare a)
{
            width = a.side;
            height = a.side;
}
```

```
int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```


Q&A