***Object Oriented Programming***

->call by value:

#include <stdio.h>

void swap(int a, int b)

{

    int t = a;

    a = b;

    b = t;

}

int main()

{

    int a, b;

    printf("\n Enter a and b:");

    scanf("%d%d", &a, &b);

    swap(a, b); // photocopies are passes

    printf("a=%d b=%d", a, b);

}

->call by address:

#include <stdio.h>

void swap(int \*x, int \*y)

{

    int t = \*x;

    \*x = \*y;

    \*y = t;

}

int main()

{

    int a, b;

    printf("\n Enter a and b:");

    scanf("%d%d", &a, &b);

    swap(&a, &b); // addresses are passed

    printf("a=%d b=%d", a, b);

}

->call by references: it is available in c++, not in c language

#include<iostream>

using namespace std;

void fun(int &a, int &b)

{

    int temp = a;

    a = b;

    b = temp;

}

int main()

{

    int a = 56;

    int b = 23;

    fun(a,b);

    cout<<"a = "<<a<<" b = "<<b<<endl;

    return 0;

}

->Referenced Variable:

#include <iostream>

using namespace std;

int main()

{

    int a = 10; // a is a normal integer variable

    int &b = a; // b is a reference variable

    a++;

    cout << " a=" << a << "b=" << b << endl; // a=11, b=11

    b++;

    cout << " a=" << a << "b=" << b << endl; // a=12, b=12

}

->Inline functions:

Using the keyword inline, we are placing a request to the compiler to execute the function as inline. In case of inline function, the program control is not halted and diverted to the body of the UDF. Instead, the function code is copied at the place of calling point.

So space and time requirement are minimized. However, the compiler may ignore our request and simply execute the function as a normal UDF.

Scenarios when a function is not executed as inline: [compiler ignores our request]

1) If the function is recursive.

2) If the function contains goto statement.

3) If the function contains loops [for, while, do-while

#include <iostream>

using namespace std;

inline void print(int a, char c = '+')

{

    for (int i = 1; i <= a; i++)

        cout << c << " ";

    cout << endl;

}

int main()

{

    print(10);

    print(5, '\*');

    print('+');

}

->Encapsulation: Wrapping up of data and methods in a single unit.

#include <iostream>

#include <string.h>

using namespace std;

class Student

{

private:    //by default it is private

    int roll;

    float cgpa;

    char name[100];

public:

    void getData()

    {

        cout << "\n Enter roll: ";

        cin >> roll;

        cout << "\n Enter cgpa: ";

        cin >> cgpa;

        cout << "\n Enter name: ";

        cin >> name;

    }

    void putData()

    {

        cout << "\n Roll=" << roll << " cgpa=" << cgpa << " name=" << name;

    }

};

int main()

{

    Student s[3]; // array of objects creation

    for (int i = 0; i < 3; i++)

        s[i].getData();

    for (int i = 0; i < 3; i++)

        s[i].putData();

}

->array of objects

#include <iostream>

#include <string.h>

using namespace std;

class Student

{

    int roll;

    float cgpa;

    char name[100];

public:

    void getData()

    {

        cout << "\n Enter roll: ";

        cin >> roll;

        cout << "\n Enter cgpa: ";

        cin >> cgpa;

        cout << "\n Enter name: ";

        cin >> name;

    }

    void putData()

    {

        cout << "\n Roll=" << roll << " cgpa=" << cgpa << " name=" << name;

    }

};

int main()

{

    Student \*s[3]; // array of object pointer creation

    Student ob[3]; // array of object

    for (int i = 0; i < 3; i++)

        s[i] = &ob[i];

    for (int i = 0; i < 3; i++)

        s[i]->getData();

    for (int i = 0; i < 3; i++)

        s[i]->putData();

}

-> Scope resolution Operator (::)

To access the global version of the variable, the :: operator is used.

#include <iostream>

using namespace std;

int a = 1;

int main()

{

    int a = 2;

    {

        int a = 3;

        cout << "a=" << a << endl;     // 3

        cout << "::a=" << ::a << endl; // 1

        {

            int a = 4;

            cout << "a=" << a << endl;     // 4

            cout << "::a=" << ::a << endl; // 1

        }

    }

    cout << "a=" << a << endl; // 2

    cout << "::a=" << ::a << endl; // 1

}

->to define the member functions out of the class :: operator is used

#include <iostream>

#include <string.h>

using namespace std;

class Student

{

    int roll;

    float cgpa;

    char name[100];

public:

    void getData();

    void putData();

    void fun(Student &); // pass by reference, modifications will be reflected in main()

};

void Student::getData()

{

    cout << "\n Enter roll: ";

    cin >> roll;

    cout << "\n Enter cgpa: ";

    cin >> cgpa;

    cout << "\n Enter name: ";

    cin >> name;

}

void Student::putData()

{

    cout << "\n Roll=" << roll << " cgpa=" << cgpa << " name=" << name;

}

void Student::fun(Student &sob)

{

    cout << "\n Roll=" << sob.roll << " cgpa=" << sob.cgpa << " name=" << sob.name;

    sob.roll = 10;

}

int main()

{

    Student ob; // array of object

    ob.getData();

    ob.fun(ob);

    ob.putData(); // roll will be displayed as 10, bcz fun() was called through reference

}

->static variables:

#include <iostream>

using namespace std;

class Sample

{

    int i;

    static int s;

public:

    void get()

    {

        i = 10;

        s++;

    }

    void print() // A non-static member method can access the static and non-static member variables

    {

        cout << "i=" << i << endl;

        cout << "s=" << s << endl;

    }

};

int Sample::s = 5;  //have to initialize the static variable in this way

int main()

{

    Sample sob1, sob2, sob3;

    sob1.get();

    sob1.print(); // 6

    sob2.print(); // 6

    sob3.print(); // 6

    sob2.get();

    sob1.print(); // 7

    sob2.print(); // 7

    sob3.print(); // 7

    sob3.get();

    sob1.print(); // 8

    sob2.print(); // 8

    sob3.print(); // 8

}

->Static functions:

#include <iostream>

using namespace std;

class Sample

{

    int i;

    static int s;

public:

    void get()

    {

        i = 10;

        s++;

    }

    void print()    //a non-static function can access the both variables

    {

        cout << "i=" << i << endl;

        cout << "s=" << s << endl;

    }

    static void display()

    {

        // cout<<"i="<<i<<endl;  //A non-static variable cannot be accessed from a static method

        cout << "s=" << s << endl;

    }

};

int Sample::s = 5;

int main()

{

    Sample sob1, sob2;

    sob1.get();

    sob1.display();

    sob2.display();

    Sample::display();  //static function can be invoked using the Class name followed by :: operator

}

this operator:

1) It is used to refer the current object.

2) It cannot be used in static method.

3) A static method can be called using this-> within a non-static member method. But the reverse is not possible.

#include <iostream>

using namespace std;

class Point

{

    int x, y;

public:

    void getData(int x, int y)

    {

        this->x = x;

        (\*this).y = y;  //this->y = y;

        this->fun(); // correct

    }

    void putData()

    {

        cout << "x=" << this->x;

        cout << "y=" << (\*this).y;

    }

    static void fun()

    {

        cout << " \n Hi ";

        // this->putData();  //incorrect

    }

};

int main()

{

    Point ob;

    ob.getData(1, 2);

    ob.putData();

    Point::fun();

    ob.fun();

}

Function overloading:

[from higher to lower datatype conversion- not done automatically]

[we cannot overload two functions differing in return type only, the number and type of the parameters must be different.]

[it is compile time polymorphism. also known as static binding.]

#include <iostream>

using namespace std;

class Sample

{

    int a;

    float b;

public:

    void sum(int x, int y)

    {

        a = x;

        b = y;

        cout << "version 1: sum=" << a + b << endl;

    }

    void sum(int x, float y)

    {

        a = x;

        b = y;

        cout << "version 2: sum=" << a + b << endl;

    }

    void sum(double x, int y)

    {

        a = y;

        b = x;

        cout << "version 3: sum=" << a + b << endl;

    }

    void sum(double x, double y)

    {

        a = x;

        b = y;

        cout << "version 4: sum=" << a + b << endl;

    }

};

int main()

{

    Sample ob;

    ob.sum(1, 2);

    ob.sum(1, 2.5F);

    ob.sum(1.5F, 3);

    ob.sum(1.5F, 2.5F);

}

->Functions with default values:

#include <iostream>

using namespace std;

class Sample{

    int a;

    float b;

public:

    void sum(int x = 2, float y = 1.0)

    {

        a = x;

        b = y;

        cout << "version 1: sum=" << a + b << endl;

    }

};

int main()

{

    Sample ob;

    ob.sum();

    ob.sum(10);

    ob.sum(10, 20);

    ob.sum(1, 2.5F);

    ob.sum(1.5F, 3);

    ob.sum(1.5F, 2.5F);

}

->Constructor Overloading:

#include<iostream>

using namespace std;

class Demo{

    int a;

    public:

    Demo()

    {

        a = 20;

        cout<<"default constructor a = "<<a<<endl;

    }

    Demo(int a = 50)    //work as default as well as a param constructor

    {

        this->a = a;

        cout<<"param constructor a = "<<a<<endl;

    }

};

int main()

{

    //Demo ob;    //error

    Demo ob2(10);

    Demo ob3(30);

}

Destructor: [it is invoked to de-allocate the memory that was allocated by the constructor.]

#include <iostream>

using namespace std;

class Sample

{

    int a, b;

public:

    Sample() // def-constructor

    {

        cout << "def\_constructor called \n";

        a = 1;

        b = 2;

    }

    Sample(int x) // a one-param constructor

    {

        cout << "one\_param\_constructor called \n";

        a = x;

        b = 2 \* x;

    }

    Sample(int x, int y)

    {

        cout << "two\_param\_constructor called \n";

        a = x;

        b = y;

    }

    void display()

    {

        cout << "a=" << a << "b=" << b << endl;

    }

    ~Sample() // takes no param.. so overloading not possible

    {

        cout << "destructor called: "<<endl;

    }

};

int main()

{

    Sample s1(10);

    s1.display();

    Sample s2;

    s2.display();

    Sample s3(10,15);

    s3.display();

}

Output:

one\_param\_constructor called

a=10b=20

def\_constructor called

a=1b=2

two\_param\_constructor called

a=10b=15

destructor called:

destructor called:

destructor called:

Demo:

#include <iostream>

using namespace std;

class Sample

{

    int a;

    static int i;

public:

    Sample() // def-constructor

    {

        cout << "def\_constructor called \n";

        a = 1;

        i++;

    }

    void display()

    {

        cout << "a=" << a << "i=" << i << endl;

    }

    ~Sample() // takes no param.. so overloading not possible

    {

        cout << "\n destructor called: ";

    }

};

int Sample::i = 1; // initialize the static member variable, [if other than 0]

int main()

{

    Sample s[10];

    s[1].display();

    s[2].display();

    Sample s3;

    s3.display();

}

/\*output:

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

def\_constructor called

a=1i=11

a=1i=11

def\_constructor called

a=1i=12

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:

 destructor called:\*/

->Return the reference:

#include <iostream>

using namespace std;

int &fun(int &x, int &y) // called by value, but returning reference

{

    if (x < y)

        return x;

    else

        return y;

}

int main()

{

    int a = 10, b = 5;

    fun(a, b) = -1;

    cout << "a=" << a << endl; // a = 10

    cout << "b=" << b << endl; // b = -1

    // in this case we first pass a = 10 and b = 5 to the function fun(), then fun()returns the address of y, and y = b;so b will be updated to - 1;

->Constant member functions and variables:

#include <iostream>

using namespace std;

class Sample

{

    mutable int a;

    const int b, c;

public:

    Sample(int a = 0) : b(2), c(1) // a const data member is initialized in this way only

    {

        this->a = a;

        // c=a;               // a const data member cannot be initialized in this way

    }

    void get()

    {

        a = 1;

        //  c++;              // a const data member once initialized, cannot be changed

    }

    void set() const // read-only function

    {

        a++;    //can write because it is made mutable

        // b++;

        // c++;

        cout << "a=" << a << "b=" << b << "c=" << c << endl;

    }

};

int main()

{

    Sample s1(10);

    s1.set();

    s1.get();

    s1.set();

}

->function call chaining:

#include <iostream>

using namespace std;

class Sample

{

    int x, y;

public:

    Sample(int a, int b)

    {

        x = a;

        y = b;

    }

    Sample &set\_x(int a1)

    {

        x = a1;

        return \*this;

    }

    Sample &set\_y(int b1)

    {

        y = b1;

        return \*this;

    }

    void display()

    {

        cout << "x=" << x << "y=" << y << endl;

    }

};

int main()

{

    Sample ob(5, 15);

    ob.display();

    ob.set\_x(20).set\_y(30).display(); // function call chaining

}

#include <iostream>

using namespace std;

class Sample

{

    int x, y;

public:

    Sample(int a, int b)

    {

        x = a;

        y = b;

    }

    Sample \*set\_x(int a1)

    {

        x = a1;

        return this;

    }

    Sample \*set\_y(int b1)

    {

        y = b1;

        return this;

    }

    void display()

    {

        cout << "x=" << x << "y=" << y << endl;

    }

};

int main()

{

    Sample ob(5, 15);

    ob.display();

    Sample \*ptr = &ob;

    ptr->set\_x(20)->set\_y(30)->display(); // function call chaining

}

->Copy constructor:

#include <iostream>

using namespace std;

class Sample

{

    int x, y;

public:

    Sample(int a = 0, int b = 0)

    {

        x = a;

        y = b;

}

    Sample(const Sample &s) // copy constructor must have to take reference

    {

        cout << "copy constructor called \n";

        x = s.x;

        y = s.y;

    }

    void display()

    {

        cout << "x=" << x << "y=" << y << endl;

    }

};

int main()

{

    Sample s1(10, 20);

    Sample s2(s1); // copy const

    Sample s3(s2); // copy const

    s3.display();

    Sample s4 = s1; // copy const

    s4.display();

    Sample s5;

    s5 = s4; // member by member copied , copy constructor not invoked

    s5.display();

}

->friend functions:

It is not a member function, it acts as a bridge between two classes.

#include <iostream>

using namespace std;

class B;    //forward declaration

class A

{

    int a;

public:

    A(int a = 10)

    {

        this->a = a;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

    friend void swap(A&, B&);

};

class B

{

    int b;

public:

    B(int b = 5)

    {

        this->b = b;

    }

    void display()

    {

        cout << "b=" << b << endl;

    }

    friend void swap(A&, B&); // if we call by value, then photocopies of the objects will be created,

    //modifications will not be reflected in main() function

};

void swap(A &aa, B &bb)

{

    int t;

    t = aa.a;

    aa.a = bb.b;

    bb.b = t;

}

int main()

{

    A obA(100);

    B obB(200);

    swap(obA, obB);

    obA.display();

    obB.display();

}

->friend class:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 10)

    {

        this->a = a;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

    friend class B; // B is declared to be a friend of class A

};

class B

{

    int b;

public:

    B(int b = 5)

    {

        this->b = b;

    }

    void set(A ob)

    {

        b = ob.a;

    }

    void print(A &ob) // pass by reference takes less space than pass by value

    {

        cout << "a=" << ob.a;

    }

    void display()

    {

        cout << "b=" << b << endl;

    }

};

int main()

{

    A obA(50);

    B obB(60);

    obB.set(obA);

    obB.display();

    obB.print(obA);

}

->Complex addition:

#include <iostream>

using namespace std;

class Complex

{

    float real, img;

public:

    Complex(float r = 0, float i = 0)

    {

        real = r;

        img = i;

    }

    Complex add(Complex&);

    friend Complex mult(Complex&, Complex&);

    void display()

    {

        cout << real << "+i" << img << endl;

    }

};

Complex Complex::add(Complex &c)

{

    Complex t;

    t.real = real + c.real;

    t.img = img + c.img;

    return t;

}

Complex mult(Complex &c1, Complex &c2)

{

    Complex t;

    t.real = c1.real \* c2.real - c1.img \* c2.img;

    t.img = c1.real \* c2.img + c1.img \* c2.real;

    return t;

}

int main()

{

    Complex c1(2, 3);

    Complex c2(4, 2);

    Complex c3;

    c3 = c1.add(c2);

    c3.display();

    c3 = mult(c1, c2);

    c3.display();

}

\*Both way friend class declaration is not worked:

#include <iostream>

using namespace std;

class B;

class A

{

    int a;

public:

    A(int a = 10)

    {

        this->a = a;

    }

    void print(B ob)    //not possible, though we make the A class frined to class B

    {

        cout<<"B ob b = "<<ob.b<<endl;

    }

    friend class B; // B is declared to be a friend of class A

};

class B

{

    int b;

public:

    B(int b = 5)

    {

        this->b = b;

    }

    void print(A &ob) // pass by reference takes less space than pass by value

    {

        cout << "A ob a = " << ob.a;

    }

    friend class A;

};

int main()

{

    A obA(50);

    B obB(60);

    obB.print(obA);

    obA.print(obB);

}