Class: Object-Oriented Programming

- **Polymorphism**
  - Templates

```
 g++ demoTemplate.cpp -o _a; ./_a
A set of integers: 42 92 79 96 66 4 85 93
After InsertionSort: 4 42 66 79 85 92 93 96
A set of strings: machine intelligence system automation
After InsertionSort: automation intelligence machine system
```

```
#include "compare.h" // in which <iostream> is already included
using namespace std;
template <class Elem> inline void swap(Elem A[], int i, int j){
// Reflection: why the inline mechanism is adopted?
        Elem temp = A[i]; A[i] = A[j]; A[j] = temp;}
template <class Elem, class Comp>
void InsertionSort(Elem A[], int n){
        for (int i=1; i<n; i++)
                for(int j=i; (j>0) && (Comp::lt(A[j], A[j-1])); j--)
                        swap(A, j, j-1);}
template <class Elem> void printA(Elem A[], int n){
        for(int i=0; i<n; i++)  cout << A[i] << " "; cout << "\n";}
int main(){
        int n_int=8,n_char=4; int a_int[] = {42, 92, 79, 96, 66, 4, 85, 93};
        // A double-quotes pair marked string terminates with '\0' by default
        const char *a_char[] = {"machine", "intelligence", "system", "automation"};
        cout << "A set of integers: "; printA(a_int, n_int);
        InsertionSort<int,IntCompare>(a_int, n_int);
        cout << "After InsertionSort: "; printA(a_int, n_int);

        cout << "A set of strings: "; printA(a_char, n_char);
        InsertionSort<const char*,CharsCompare>(a_char, n_char);
        cout << "After InsertionSort: "; printA(a_char, n_char);
        return 0;
}
```

```
#include <iostream>
using namespace std;

// Integer comparison class
class IntCompare{
public: static bool lt(int x, int y){ return x < y; }};
// Chars comparison class
class CharsCompare{
public:
        // Each chain of chars terminates with '\0'
        static bool lt(const char* x, const char* y){
                while (*x!='\0' && *y!='\0' && *x==*y){x++;y++;}
                if (*x < *y) return true;
                return false;
        }
};
"compare.h" [dos] 55L, 1481C
```

Class: Object-Oriented Programming

- **Inheritance**
  – Derived classes (members reusage)

```
#include <iostream>
#include <cmath>
using namespace std;

class V2{ // two-dimensional vector space
protected:float x,y;
//private:float x,y;
public: V2(){} V2(const V2 &c){x=c.x;y=c.y;}
        V2(float xi, float yi){x=xi;y=yi;} // constructor overloading
        float E(int i) const{return 1==i?x:y;} // get element {1,2}
        // function 'const' indicates it does not modify the object
        // for which it is called (e.g. involved with const arguments)
        void S(){cout<<'['<<x<<','<<y<<']';}
        friend V2 operator +(V2,V2);friend V2 operator -(V2,V2);
        friend V2 operator *(float,V2);friend V2 operator *(V2,float);
};
V2 operator+(V2 a, V2 b){return V2(a.x+b.x,a.y+b.y);}
V2 operator-(V2 a, V2 b){return V2(a.x-b.x,a.y-b.y);}
V2 operator*(float a, V2 c){return V2(a*c.x,a*c.y);}
V2 operator*(V2 c, float a){return V2(a*c.x,a*c.y);}
```

```
class V2EN: public V2{ // 2D vector with Euclidean norm
public: V2EN(){} V2EN(float xi, float yi):V2(xi,yi){}
        V2EN(const V2 &c): V2(c){} // constructor overloading
        // which (different-type 'copy construction') allows both
        // 1) {V2EN}={V2} (different-type 'copy assignment')
        // 2) V2EN {FUNC}{... return {V2};}
        //V2EN& operator=(const V2& c){x=c.E(1);y=c.E(2); return *this;}
        // which (different-type 'copy assignment') allows only
        // 1) {V2EN}={V2} but not 2) V2EN {FUNC}{... return {V2};}
        double ENorm() // 'protected' x,y in V2 for derived classes
        {return sqrt(x*x+y*y);} // if private x,y in V2, what happens?
        //double ENorm() // 'private/protected' for general public
        //{return sqrt(pow(this->E(1),2)+pow(this->E(2),2));}
};
// explicit V2 conversion without taking advantage of slicing
V2EN operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2EN operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2EN operator*(float a, V2EN c){return a*(V2)c;}
V2EN operator*(V2EN c, float a){return (V2)c*a;}
/* if 'V2EN& operator=' is used instead of 'V2EN(const V2&c):V2(c)'
V2 operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2 operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2 operator*(float a, V2EN c){return a*(V2)c;}
V2 operator*(V2EN c, float a){return (V2)c*a;}*/
```

Class: Object-Oriented Programming

- **Inheritance**
  - Derived classes (members reusage)

```
 g++ demoInheritance.cpp -o _a; ./_a
a=[4,3];b=[3,4];an=[4,3];bn=[3,4]
a+b=[7,7]
2*a=[8,6]
an-bn=[1,-1]
bn*2=[6,8]
|an|=5;|cn|=10;
[4,3]-[3,4]=[1,-1]
[4,3]*-2=[-8,-6]
[4,3]+[3,4]=[7,7]
-2*[3,4]=[-6,-8]
```

```
int main(){
        V2 a(4,3), b(3,4), c; V2EN an(4,3), bn(3,4), cn;
        cout<<"a=";a.S();cout<<";b=";b.S();cout<<';';
        cout<<"an=";an.S();cout<<";bn=";bn.S();cout<<endl;
        c=a+b;cout<<"a+b=";c.S();cout<<endl;
        c=2*a;cout<<"2*a=";c.S();cout<<endl;
        cn=an-bn;cout<<"an-bn=";cn.S();cout<<endl;
        cn=bn*2;cout<<"bn*2=";cn.S();cout<<endl;
        cout<<"|an|="<<an.ENorm()<<';'<<"|cn|="<<cn.ENorm()<<";\n";

        c=a-b;a.S();cout<<'-';b.S();cout<<'=';c.S();cout<<endl;
        c=a*-2;a.S();cout<<'*'<<"-2=";c.S();cout<<endl;
        cn=an+bn;an.S();cout<<'+';bn.S();cout<<'=';cn.S();cout<<endl;
        cn=-2*bn;cout<<"-2*";bn.S();cout<<'=';cn.S();cout<<endl;
        return 0;
}
```

```
class V2EN: public V2{ // 2D vector with Euclidean norm
public: V2EN(){} V2EN(float xi, float yi):V2(xi,yi){}
        V2EN(const V2 &c): V2(c){} // constructor overloading
        // which (different-type 'copy construction') allows both
        // 1) {V2EN}={V2} (different-type 'copy assignment')
        // 2) V2EN {FUNC}{... return {V2};}
        //V2EN& operator=(const V2& c){x=c.E(1);y=c.E(2); return *this;}
        // which (different-type 'copy assignment') allows only
        // 1) {V2EN}={V2} but not 2) V2EN {FUNC}{... return {V2};}
        double ENorm() // 'protected' x,y in V2 for derived classes
        {return sqrt(x*x+y*y);} // if private x,y in V2, what happens?
        //double ENorm() // 'private/protected' for general public
        //{return sqrt(pow(this->E(1),2)+pow(this->E(2),2));}
};
// explicit V2 conversion without taking advantage of slicing
V2EN operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2EN operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2EN operator*(float a, V2EN c){return a*(V2)c;}
V2EN operator*(V2EN c, float a){return (V2)c*a;}
/* if 'V2EN& operator=' is used instead of 'V2EN(const V2&c):V2(c)'
V2 operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2 operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2 operator*(float a, V2EN c){return a*(V2)c;}
V2 operator*(V2EN c, float a){return (V2)c*a;}*/
```

Class: Object-Oriented Programming

- **Inheritance**
  - Derived classes (members reusage)

```
class V2EN: public V2{ // 2D vector with Euclidean norm
public: V2EN(){} V2EN(float xi, float yi):V2(xi,yi){}
        V2EN(const V2 &c): V2(c){} // constructor overloading
        // which (different-type 'copy construction') allows both
        // 1) {V2EN}={V2} (different-type 'copy assignment')
        // 2) V2EN {FUNC}{... return {V2};}
        //V2EN& operator=(const V2& c){x=c.E(1);y=c.E(2); return *this;}
        // which (different-type 'copy assignment') allows only
        // 1) {V2EN}={V2} but not 2) V2EN {FUNC}{... return {V2};}
        double ENorm() // 'protected' x,y in V2 for derived classes
        {return sqrt(x*x+y*y);} // if private x,y in V2, what happens?
        //double ENorm() // 'private/protected' for general public
        //{return sqrt(pow(this->E(1),2)+pow(this->E(2),2));}
};
// explicit V2 conversion without taking advantage of slicing
V2EN operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2EN operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2EN operator*(float a, V2EN c){return a*(V2)c;}
V2EN operator*(V2EN c, float a){return (V2)c*a;}
/* if 'V2EN& operator=' is used instead of 'V2EN(const V2&c):V2(c)'
V2 operator+(V2EN a, V2EN b){return (V2)a+(V2)b;}
V2 operator-(V2EN a, V2EN b){return (V2)a-(V2)b;}
V2 operator*(float a, V2EN c){return a*(V2)c;}
V2 operator*(V2EN c, float a){return (V2)c*a;}*/
```

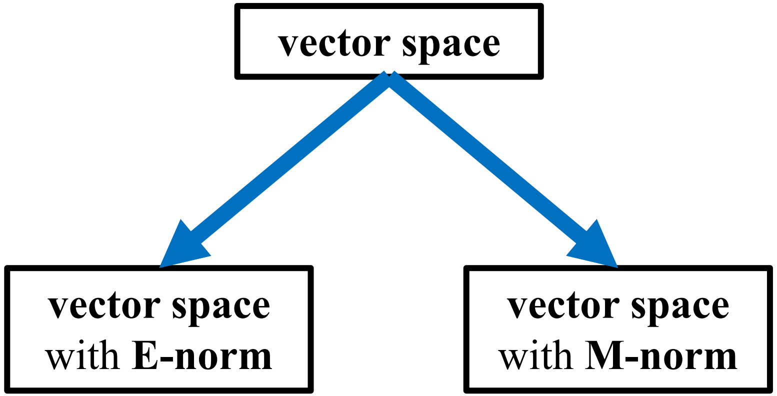
Ce n'est pas la peine!

slicing

```
class V2EN: public V2{ // 2D vector with Euclidean norm
public: V2EN(){} V2EN(float xi, float yi):V2(xi,yi){}
        V2EN(const V2 &c): V2(c){} // constructor overloading
        // which (different-type 'copy construction') allows both
        // 1) {V2EN}={V2} (different-type 'copy assignment')
        // 2) V2EN {FUNC}{... return {V2};}
        //V2EN& operator=(const V2& c){x=c.E(1);y=c.E(2); return *this;}
        // which (different-type 'copy assignment') allows only
        // 1) {V2EN}={V2} but not 2) V2EN {FUNC}{... return {V2};}
        double ENorm() // 'protected' x,y in V2 for derived classes
        {return sqrt(x*x+y*y);} // if private x,y in V2, what happens?
        //double ENorm() // 'private/protected' for general public
        //{return sqrt(pow(this->E(1),2)+pow(this->E(2),2));}
};
/*********************************************************
// Slicing: partial copy when the derived converted to the base
// When '{V2EN}{+/...}{V2EN}' is used, 'V2 operator{+/...}(V2,V2)' is
// applied. The two operands {V2EN} are first sliced to {V2}, so
// '{V2EN}{+/...}{V2EN}' does 'V2{V2EN}{+/...}V2{V2EN}' and returns {V2}
*********************************************************/
```

Class: Object-Oriented Programming

- **Inheritance**
  - Derived classes (members reusage)



1) Repetitive works avoidance
2) Concepts organization

```
class V2EN: public V2{ // 2D vector with Euclidean norm
public: V2EN(){} V2EN(float xi, float yi):V2(xi,yi){}
        V2EN(const V2 &c): V2(c){} // constructor overloading
        // which (different-type 'copy construction') allows both
        // 1) {V2EN}={V2} (different-type 'copy assignment')
        // 2) V2EN {FUNC}{... return {V2};}
        //V2EN& operator=(const V2& c){x=c.E(1);y=c.E(2); return *this;}
        // which (different-type 'copy assignment') allows only
        // 1) {V2EN}={V2} but not 2) V2EN {FUNC}{... return {V2};}
        double ENorm() // 'protected' x,y in V2 for derived classes
        {return sqrt(x*x+y*y);} // if private x,y in V2, what happens?
        //double ENorm() // 'private/protected' for general public
        //{return sqrt(pow(this->E(1),2)+pow(this->E(2),2));}
};
/*********************************************************
// Slicing: partial copy when the derived converted to the base
// When '{V2EN}{+/...}{V2EN}' is used, 'V2 operator{+/...}(V2,V2)' is
// applied. The two operands {V2EN} are first sliced to {V2}, so
// '{V2EN}{+/...}{V2EN}' does 'V2{V2EN}{+/...}V2{V2EN}' and returns {V2}
*********************************************************/
/*********************************************************
// Define a derived class V2MN in similar way
class V2MN: public V2{ // 2D vector with Mahalanobis norm
        ... ...};
*********************************************************/
```

Class: Object-Oriented Programming

- **Inheritance**
  - Derived classes (members reusage)

private	*class*'s members & friends can use
protected	*class*'s members & friends and *derived classes*'s members & friends can use
public	general public can use

base class's members	**private**	**protected**	**public**
private inheritance	private	private	private
protected inheritance	private	protected	protected
public inheritance	private	protected	public

types of inherited members in derived classes

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions

virtual {TYPE} {FUNCTION}({ARGUMENTS});

virtual means "*may* be redefined later in a class derived from this one"

virtual {TYPE} {FUNCTION}({ARGUMENTS}) **=0**;

=0 says the **virtual** function is **pure virtual**, which means "not defined in this one, but *must* be defined later in any class derived from this one"

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces

```
#include <iostream>
using namespace std;
class Shape{public:virtual float A()=0;virtual float C()=0;virtual void S()=0;};
class Rect: public Shape{private:float x, y, w, h;
public: Rect(float xi, float yi, float wi, float hi)
        {x=xi;y=yi;w=wi;h=hi;}
        float A(){return w*h;} float C(){return 2*(w+h);}
        void S(){printf("Rect{x:%f,y:%f,w:%f,h:%f}\n",x,y,w,h);}
};
class Circ: public Shape{private:float x, y, r;
public: Circ(float xi, float yi, float ri){x=xi;y=yi;r=ri;}
        float A(){return 3.14*r*r;} float C(){return 6.28*r;}
        void S(){printf("Circle{x:%f,y:%f,r:%f}\n",x,y,r);}
};
int main(int argn, char* argc[]){
        // cannot instantiate a pure virtual type, but a reference of it
        // serves somewhat as the void* pointer in C programming language
        Shape* a_shape;
        if (1==argn || argc[1][0]=='r'){
                Rect a_rect(4,3,6,5); a_shape = &a_rect;
        }else{  Circ a_circ(7,8,10); a_shape = &a_circ; }
        a_shape->S();cout << "Shape area: "<<a_shape->A()<<"\n";
        cout << "Shape circumference: "<<a_shape->C()<<"\n";
        return 0;
}
"demoVirtual_pure.cpp" 25L, 1017C                                   2,20
```

virtual {TYPE} {FUNC}({ARG}) =0;

pure virtual

```
 g++ demoVirtual_pure.cpp -o _a; ./_a r; ./_a c
Rect{x:4.000000,y:3.000000,w:6.000000,h:5.000000}
Shape area: 30
Shape circumference: 22
Circle{x:7.000000,y:8.000000,r:10.000000}
Shape area: 314
Shape circumference: 62.8
```

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces

virtual {TYPE} {FUNC}({ARG}) ;

(general i.e. non-pure) virtual

```
#include <iostream>
using namespace std;
class Shape{public: virtual float A(){return 0;};
        virtual float C(){return 0;};virtual void S(){};};
class Rect: public Shape{private:float x, y, w, h;
public: Rect(float xi, float yi, float wi, float hi)
        {x=xi;y=yi;w=wi;h=hi;}
        float A(){return w*h;} float C(){return 2*(w+h);}
        void S(){printf("Rect{x:%f,y:%f,w:%f,h:%f}\n",x,y,w,h);}
};
class Circ: public Shape{private:float x, y, r;
public: Circ(float xi, float yi, float ri){x=xi;y=yi;r=ri;}
        float A(){return 3.14*r*r;} float C(){return 6.28*r;}
        void S(){printf("Circle{x:%f,y:%f,r:%f}\n",x,y,r);}
};
int main(int argn, char* argc[]){
        Shape* a_shape;
        if (1==argn || argc[1][0]=='r'){
                Rect a_rect(4,3,6,5); a_shape = &a_rect;
        }else{  Circ a_circ(7,8,10); a_shape = &a_circ; }
        a_shape->S();cout << "Shape area: "<<a_shape->A()<<"\n";
        cout << "Shape circumference: "<<a_shape->C()<<"\n";
        return 0;
}
```

```
 g++ demoVirtual_def.cpp -o _a; ./_a r; ./_a c
Rect{x:4.000000,y:3.000000,w:6.000000,h:5.000000}
Shape area: 30
Shape circumference: 22
Circle{x:7.000000,y:8.000000,r:10.000000}
Shape area: 314
Shape circumference: 62.8
```

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces

virtual {TYPE} {FUNC}({ARG}) ;

(general i.e. non-pure) virtual

why pure virtual ?

```
 g++ demoVirtual_ndef.cpp -o _a; ./_a r; ./_a c
Rect{x:4.000000,y:3.000000,w:6.000000,h:5.000000}
Shape area: 30
Shape circumference: 22
Circle{x:7.000000,y:8.000000,r:10.000000}
Shape area: 0
Shape circumference: 0
```

```
#include <iostream>
using namespace std;
class Shape{public: virtual float A(){return 0;};
        virtual float C(){return 0;};virtual void S(){};};
class Rect: public Shape{private:float x, y, w, h;
public: Rect(float xi, float yi, float wi, float hi)
        {x=xi;y=yi;w=wi;h=hi;}
        float A(){return w*h;} float C(){return 2*(w+h);}
        void S(){printf("Rect{x:%f,y:%f,w:%f,h:%f}\n",x,y,w,h);}
};
class Circ: public Shape{private:float x, y, r;
public: Circ(float xi, float yi, float ri){x=xi;y=yi;r=ri;}
        // if definition of virtual functions is forgotten
        //float A(){return 3.14*r*r;} float C(){return 6.28*r;}
        void S(){printf("Circle{x:%f,y:%f,r:%f}\n",x,y,r);}
};
int main(int argn, char* argc[]){
        Shape* a_shape;
        if (1==argn || argc[1][0]=='r'){
                Rect a_rect(4,3,6,5); a_shape = &a_rect;
        }else{  Circ a_circ(7,8,10); a_shape = &a_circ; }
        a_shape->S();cout << "Shape area: "<<a_shape->A()<<"\n";
        cout << "Shape circumference: "<<a_shape->C()<<"\n";
        return 0;
}
```

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces

{TYPE} {FUNC}({ARG}) ;

non-virtual

why virtual ?

```
g++ demoVirtual_non.cpp -o _a; ./_a r; ./_a c
Rect{x:4.000000,y:3.000000,w:6.000000,h:5.000000}
Shape area: 0
Shape circumference: 0
Circle{x:7.000000,y:8.000000,r:10.000000}
Shape area: 0
Shape circumference: 0
```

```
#include <iostream>
using namespace std;
class Shape{public: float A(){return 0;}; float C(){return 0;};
        virtual void S(){};};
class Rect: public Shape{private:float x, y, w, h;
public: Rect(float xi, float yi, float wi, float hi)
        {x=xi;y=yi;w=wi;h=hi;}
        float A(){return w*h;} float C(){return 2*(w+h);}
        void S(){printf("Rect{x:%f,y:%f,w:%f,h:%f}\n",x,y,w,h);}
};
class Circ: public Shape{private:float x, y, r;
public: Circ(float xi, float yi, float ri){x=xi;y=yi;r=ri;}
        float A(){return 3.14*r*r;} float C(){return 6.28*r;}
        void S(){printf("Circle{x:%f,y:%f,r:%f}\n",x,y,r);}
};
int main(int argn, char* argc[]){
        Shape* a_shape;
        if (1==argn || argc[1][0]=='r'){
                Rect a_rect(4,3,6,5); a_shape = &a_rect;
        }else{  Circ a_circ(7,8,10); a_shape = &a_circ; }
        a_shape->S();cout << "Shape area: "<<a_shape->A()<<"\n";
        cout << "Shape circumference: "<<a_shape->C()<<"\n";
        return 0;
}
```

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces
- Non-virtual
  - If *F()* of *base class* is not virtual, *base pointer->F()* in *main* always executes *F()* of *base class*, be *F()* redefined or not in *derived class*.
- Virtual (general i.e. non-pure)
  - If *F()* of *base class* is virtual but without *=0*, *base pointer->F()* in *main* executes *F()* of *derived class* if *F()* is refined in *derived class*; otherwise, executes *F()* of *base class*.
- Pure virtual
  - If *A()* of *base class* is pure virtual i.e. with *=0*, *F()* must be redefined in *derived class*, so *base pointer->A()* in *main* definitely executes *F()* of *derived class*.

Class: Object-Oriented Programming

- **Inheritance+Polymorphism**
  - Virtual functions
  - Public interfaces

non-virtual

⇕

virtual

⇕

pure virtual

virtual {TYPE} {FUNCTION}({ARGUMENTS});

virtual means "*may* be redefined later in a class derived from this one"

virtual {TYPE} {FUNCTION}({ARGUMENTS}) **=0**;

=0 says the **virtual** function is **pure virtual**, which means "not defined in this one, but *must* be defined later in any class derived from this one"

THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY