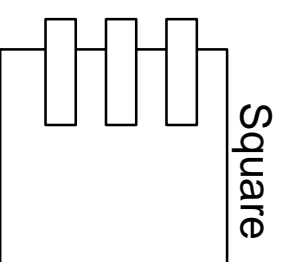
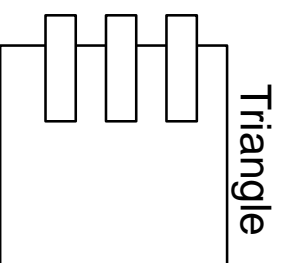
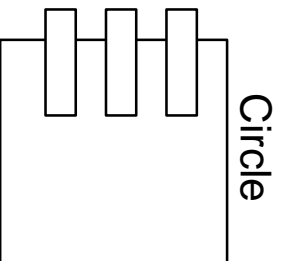


The C++ Programming Language

- Inheritance.
- Virtual functions.



Requirements of treating a collection of shapes together.

```
f() {  
    an array of circles;  
    an array of triangles;  
    an array of squares;  
    for(i = 0; i < MAX; i++) { /* circles */ }  
    for(i = 0; i < MAX; i++) { /* triangles */ }  
    // ...  
}
```

Support Data Abstraction by using C++:

```
class point { /*...*/ };
class colour { /*...*/ };
enum kind {circle, triangle, square};

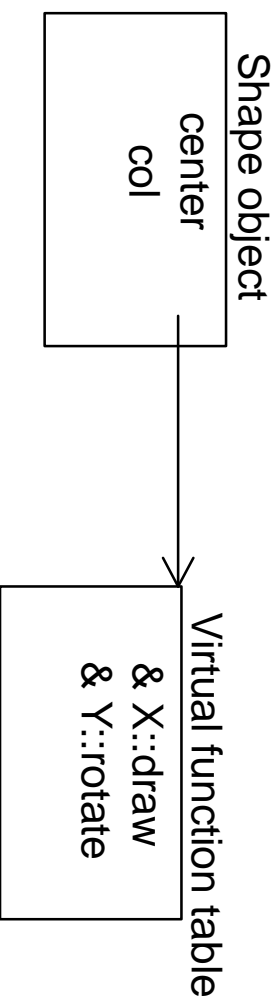
class shape {
    point center;
    colour col;
    kind k;
    // representation of shape
public:
    point where(){return center;}
    void draw();
    void move(point to){center = to; draw ();}
    void rotate(int);
    // ...
};
```

```
void
shape::draw() {
    switch(k) {
        case circle:
            // draw a circle
            break;
        case triangle:
            // draw a triangle
            break;
        case square:
            // draw a square
            break;
    }
}
```

Object-Oriented Programming: Inheritance and Virtual

Function:

```
class shape {
    point center;
    colour col;
public:
    point where(){return center;}
    void move(point to){center = to; draw ();}
    virtual void draw();
    virtual void rotate(int);
};
class circle : public shape {
    int radius;
public:
    void draw();
    void rotate(int);
};
```



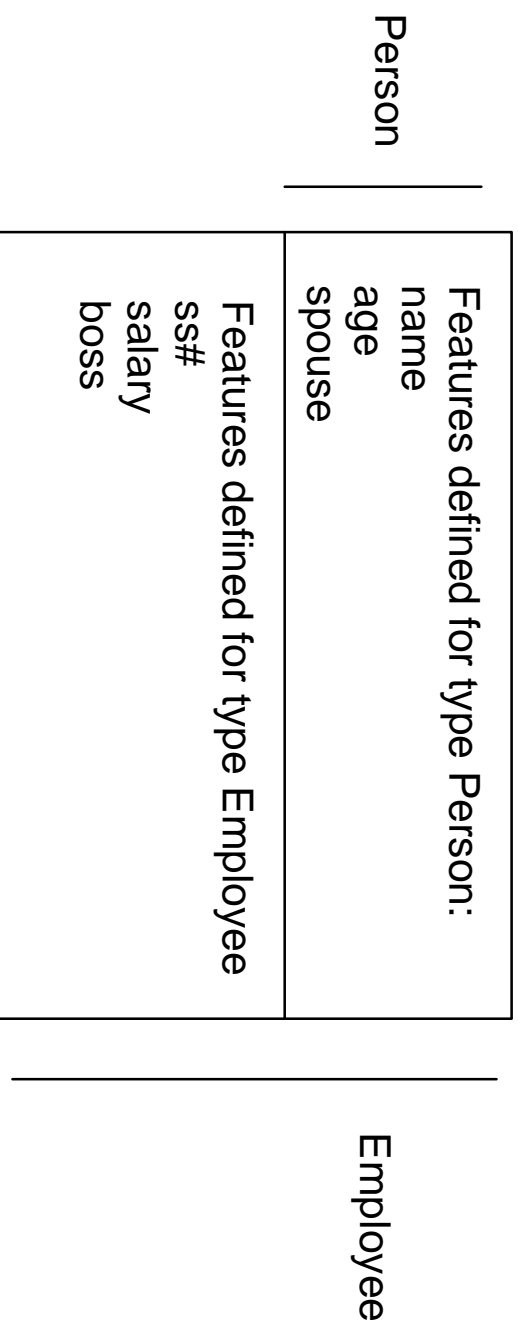
For a circle object, both X and Y are circle!

```
void rotate_all(shape v[], int size, int angle)
// rotate all members of an array "v" of size
// "size", "angle" degrees.
{
    int i = 0;
    while (i < size){
        v[i].rotate(angle);
        i = i + 1;
    }
}
```

The General Idea of Inheritance

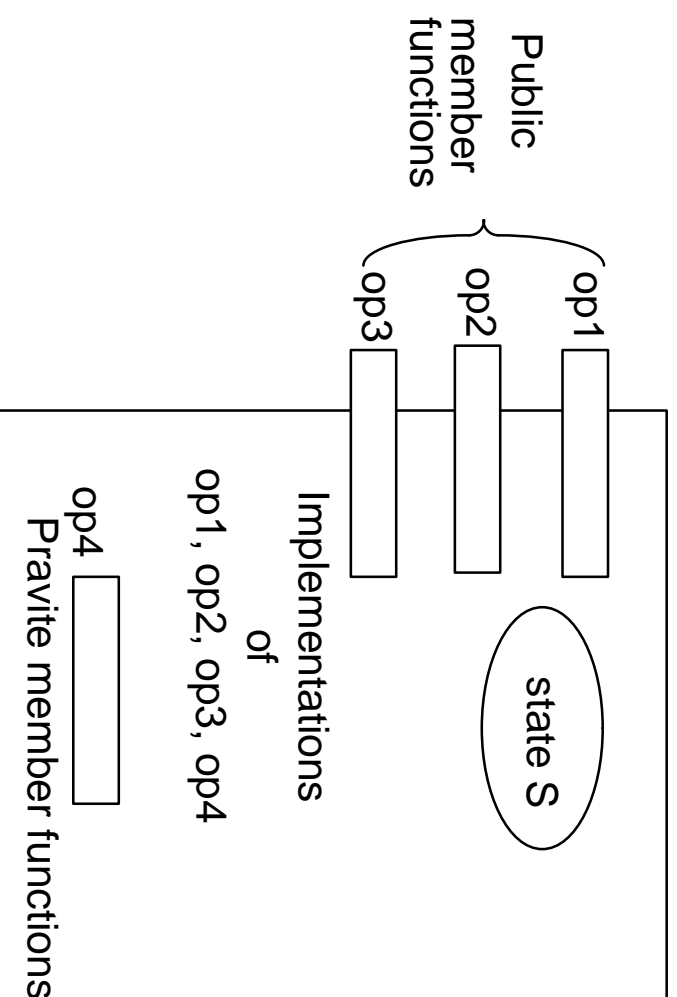
- Inheritance is the implementation support of so-called is-a relationships between object types.
- First, is-a relationship implies an explicit subset relationship between the extension of the supertype and the extension of the subtype.
- Second, Inheritance is an aspect of the is-a relationship. An instance of a subtype inherits all features of its supertype and its ancestors. Subtype — somewhat contradictory to the name, is an augmentation of the supertype.
- Third, Substitutability of subtype instances for supertype instances: All structural representation and operations.

Schematical Illustration of Inheritance



- Deal an Employee instance as a Person or an Employee instance.
- It would violate the principle of substitutability if a subtype had excluded some operation from its public clause that included in the public clause of one of its supertypes.

Using C++ Terminology



Public/Private Base Classes:

- A public base class defines an is—a relationship
- A private base class reflects a form of inheritance not based on subtype relationships. The entire public interface of the base class becomes private in the derived class.

```
class Student : public Person { /* ... */ };  
  
class Stack : private Array { /* ... */ };
```

Access Control

```
class A {  
    private:  
        int i;  
        void f();  
    protected:  
        void g();  
    public:  
        void h();  
};
```

- Private : can be used only by member functions and friends of the class in which it is declared.
- Protected : can be used only by member functions and friends of the class in which it is declared, and by member functions and friends of classes derived from this class.
- Public : can be used by any function.

Private Members

```
class X {  
    private:  
        void f(int);  
        int a;  
};  
  
void X::f(int i){  
    a += i;  
}  
  
void g(X& x){  
    int i = x::a;           // error  
    x.f(2);                // error  
    x.a++;                 // error  
}
```

Protected Members

```
class X {  
    // private by default  
    int priv;  
    protected:  
        int prot;  
    public:  
        int publ;  
        void m();  
};  
void X::m()  
{  
    priv = 1;           // ok  
    prot = 2;         // ok  
    publ = 3;         // ok;  
}
```

```
class Y : public X {
    void mderived();
};

Y::mderived() {
    priv = 1;           // error
    prot = 2;          // ok
    publ = 3;          // ok
}

void f(Y* p) {
    p->priv = 1;       // error
    p->prot = 2;       // error
    p->publ = 3;       // ok
};
```

“public base class”:

public members of the base class → public

protected members of the base class → protected

Initialisation and Clean up

```
class Person {  
    char* name;  
public:  
    Person();  
    ~Person();  
};
```

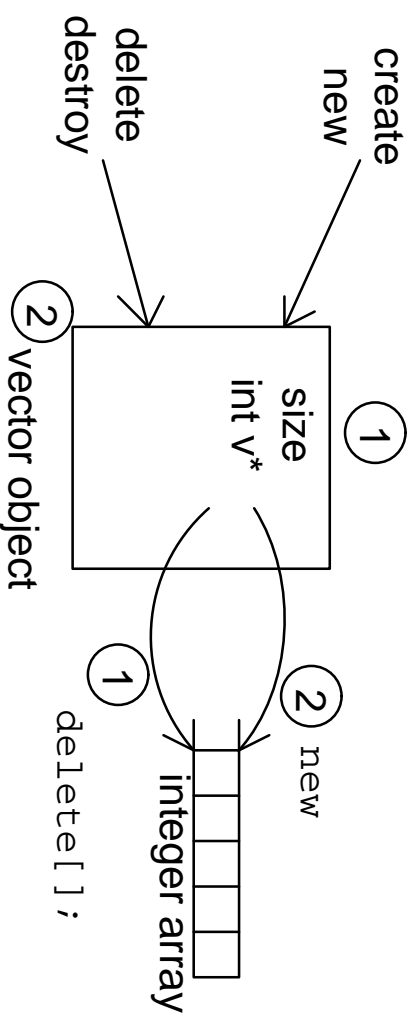
Constructor :

- a distinguished function to do the allocation and initialisation.
- A constructor is defined by having the same name as its class.

Destructor :

- to clean up objects after their last use.
- A destructor is identified by having the same name as its class prefixed by ~

new vs delete & constructor vs destructor

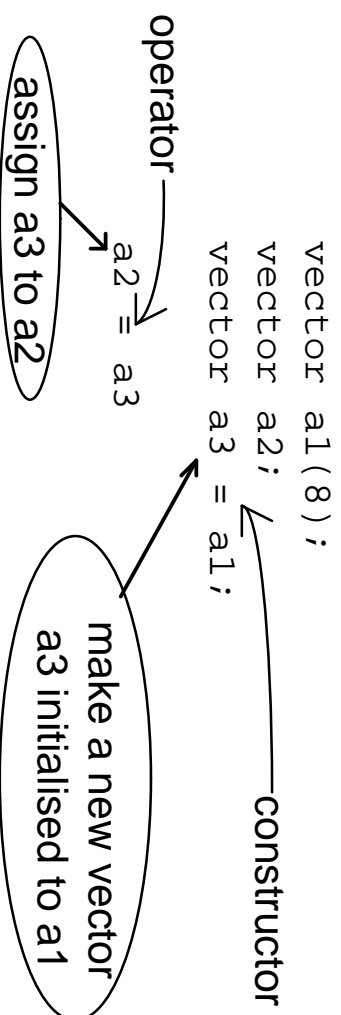


References

- A reference is an alternative name for an object.
- X& means reference to X.
- A reference must be initialized.
- A reference cannot be changed to another object after initialisation.

```
int i = 1;
int& r = i; // r and i refer to the same int i
int x = r; // x = 1
int ii = 0;
int& rr = ii;
rr++; // ii is incremented to 1
int v[2];
int& g(int i){return v[i];}
g(3) = 7;
```

Assignment and initialisation



In C++, a constructor of the form `T (const T&) is a copy constructor.`

```
class vector {  
    // ...  
public:  
    void operator = (const vector&);  
    vector (const vector&);  
};
```

More About initialisation: Default Values

It is often nice to provide several ways of initialising an object.

```
class date {
    int month, day, year;
public:
    date(int d, int m, int y);
    date(int d, int m); // today's year
    date(int d); // today's month and year
    date() // default date: today
    date(const char*);
};

date today(4);
date july4("july 4, 1994");
date now();
```

More About initialisation (continue)

```
class date {
    public:
        date(int d = 0, int m = 0, int y = 0);
};

date::date(int d, int m, int y){
    day   = d ? d : today.day;
    month = m ? m : today.month;
    year  = y ? y : today.year;
    //...
}
```

Self Reference

```
class dlink {
    dlink* prev;
    dlink* suc;
public:
    void append(dlink *);
};

void
dlink::append(dlink* p){
    p->suc = suc;           // p->suc = this->suc
    p->pre = this;
    suc->pre = p;          // this->suc->pre-> = p
    suc = p;              // this->suc = p
}

dlink* list_head;
list_head->append(a);
list_head->append(b);
```

The Scope Resolution Operator

```
int x;                // global

void f(){
    int x;
    x = 1;
}

    int x;
    x = 2;
}
    x = 3;
}
int *p = &x;
```

The scope extends from the point of declaration to the end of the block in which its declaration occurs.

Scope (continued)

```
int x;
void f2 () {
    int x = 1;
    ::x = 2;
}

class X {
    int m;
public:
    int readm () { return m; }
    void setm (int m) { X::m = m; }
};
```

Scope (continued)

```
class my_file {
    //...
    public:
        int open(const char*, const char*);
};

int
myfile::open(const char* name,
             const char* spec)
{
    // use the UNIX(2) open()
    if (::open(name, flag) { /* ... */ }
    // ...
}
```

Friend Function

It gives a non-member function access to the hidden members of the class. Its use is a method of escaping the strict strong-typing and data-hiding restrictions

```
vector  
multiply(const matrix& m, const vector& v){  
    vector r;  
    for(int i = 0; i < 3; i++){  
        // r[i] = m[i] * v  
        r.elem(i) = 0;  
        for(int j = 0; j < 3; j++){  
            r.elem(i) += m.elem(i, j) * v.elem(j);  
        }  
    }  
    return r;  
}
```

```
class vector {
    float v[4];
    friend vector
        multiply(const matrix&, const vector&);
};
class matrix {
    vector v[4];
    friend vector
        multiply(const matrix&, const vector&);
};
vector multiply(const matrix& m, const vector& v){
    vector r;
    for(int i = 0; i < 3; i++){ // r[i] = m[i] * v
        r.v[i] = 0;
        for(int j = 0; j < 3; j++)
            r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}
```

Inline Function

```
class A {  
    int size;  
    public:  
    // ...  
    int get_size() { return size; }  
};  
  
inline int  
A::get_size() {  
    return size;  
}  
  
int  
A::get_size() {  
    return size;  
}
```