private overload

some operator can't be overload.

the implicit one (on the left) can't automatically change the type.

global overload

1. expilicit first argument
2. type conversion can be performed on both arguments
3. need to be declared **friend**

the return values is const so it can be regarded as the left arguments.

reference

- must be initialized when defined (can't be NULL)
- 引用一经定义便无法修改（一种绑定binding）
- 引用的变化也会导致被引对象的变化

```
int x = 10;
int & y = x;
y = 100;
cout << y << " " << x;
//ans: 100 100
```

- 引用的目标必须有非临时分配的空间（一般不为表达式）

```
void func(int & a);
int i = 1;
func (i++);//error!
```

- **不存在** *引用* 的引用， *数组* 的引用， *指针* 的引用

```
int & * a;//false! it's a pointer to reference.
int * & a;//okay. it's a reference of a pointer.
```

reference in function (overload)

- 基类的引用类型可以用派生类当作参数
- 使用引用的原因
  - 修改调用函数中的数据对象
  - 传递引用而非对象加快效
- 函数返回值也可以为引用
- 引用作为函数返回值或参数，应适当选择对不希望更改的量加const

```
struct Person{
    char *name;
    Person(const char *s){
        name=new char[strlen(s)+1];
        strcpy(name,s);
    }
    Person(const Person & a){
        name = new char[strlen(a.name)];
        strcpy(name, a.name);
    }
    ~Person(){

    }
};

//不需要拷贝构造函数的一种方式
Person & bar(const char *s)
{
    cout<<"in bar()"<<endl;
    Person a(s);
    Person & x = a;
    Person * y = &a;
    return x;
    //return a;
}

int main()
{
    char a[5]="sdfa";
    Person p1(a);
    Person p2(p1);
    cout<<(void *)p1.name<<endl;
    cout<<(void *)p2.name<<endl; //会发现输出的地址是一样的，说明指针在默认情况下也进行了
copy,若要避免则应该自己编写拷贝构造函数
    Person p3 = bar(a);
    cout << (void *)p3.name << endl;

    return 0;
}
```

Static

- only initialize once when encounter its def. but live for whole program
- static member is shared by all the objects of the class
    - ::
    - .
- variables or functions are all inhibited to use out of the .cpp. but .h is ok.

```
#include<cstring>
#include<iostream>
#include "test.h"
using namespace std;

int main()
{
    cout << aaa << " " << bbb;

    return 0;
}//static int aaa=5 in test.h, static int bbb = 8 in test2.cpp;
//aaa is ok while bbb is can't find.
```

Global Object

- construct before main, in appearance order
- destruct when main end or called exit
- global vars   can be shared with *extern*.
- static global vars
- static local vars
- stack
  - local vars
- heap
  - dynamic allocate vars

new & delete

- 对一个指针重复new会导致内存泄漏
- 同一个字段不能delete多次
- 可以delete一个空指针
- delete和new配套使用
- 默认类型：delete **[]** x 的[]加不加无所谓
- 自定义类：delete调用析构函数。此时对单个变量，加[]会无限析构；对变量数组，不加[]会只执行一次然后报错。

```
int main(){

    char std[10]="sdfew";
    char * a = new char[5];
    char * b = a;
    delete [] a;
    cout << *b << endl;//error! b and a point to same memory which is delete
    a = std;
    cout << a;//ok
    //delete删除的是指向的空间而非变量本身。
    //变量和指向的地址都存在，但地址没有空间。指向某段地址，但被删掉空间的指针不是空指针
    return 0;
}
```

Reference

- must be initialized when defined (can't be NULL)
- 引用一经定义便无法修改（一种绑定binding)
- 引用的变化也会导致被引对象的变化

```
int x = 10;
int & y = x;
y = 100;
cout << y << " " << x;
//ans: 100 100
```

- 引用的目标必须有非临时分配的空间（一般不为表达式）

```
void func(int & a);
int i = 1;
func (i++);//error!
```

- **不存在** *引用* 的引用， *数组* 的引用， *指针* 的引用

```
int & * a;//false! it's a pointer to reference.
int * & a;//okay. it's a reference of a pointer.
```

Const

OOP Characteristic

- Everything is an object.
- A program is a bunch of objects telling each other what to do by sending messages.
- Each object has its own memory made up of other objects.
- Every object has a type.
- All objects of a particular type can receive the same messages.

Ctor & Dtor

- the default ctor
    - system will automatically create a d_ctor if there's no any ctor.
    - **don't** use `X a()` when there's ctor with arguments but no d_ctor. use `X a` without bracket.(see the code under ctor with arguments)

```
class X{
    int i;
public:
    X();
}
void f(){    X a; X b(); /*both call the public X() ctor.*/}
```

- the ctor with arguments
    - 构造函数的参数名不能与成员变量名相同。
    - **initializer list**： write the initialize before the bracket (the constructor body).
        - 初始化顺序与初始化列表的顺序无关，而只与声明顺序有关，销毁顺序则相反。
        - 在内存分配时直接初始化（写在构造函数内，即带参数的一般构造函数做法是**先用默认构造声明**，再在函数体内部赋值，开销增加）。
            - 这也是为什么default ctor中说必须要有一个默认构造函数，因为这个时候已经有了含参构造函数，系统不会auto default ctor。但默认含参构造函数如果不用列表初始化则又要先调用要给default ctor。
        - **可以**初始化*const*，引用等写在构造函数内部不能赋值初始化的量。

```
...
    X(int i){}
...
void f(){    X a(12);    }//what's more, because X(int i)exist, there won't automatically create a default X(). need to add one by oneself.
```

```
class MyClass {
private:
    int a;
    double b;
public:
    MyClass(int x, double y) : b(y), a(b + x) { cout<< a << " " << b <<endl;}
    // Even though 'b' is before 'a' in the list, 'a' is initialized first!
};
void foo(){    MyClass(3, 4) } //output: 3 4.(a is a random number)
```

- **不能用goto，switch 跳过构造函数的初始化。**
    - 原因：程序在进入一个大括号（新的作用域）时就对内部所有变量申请了空间。但对象的构造函数并不会被立刻执行，而是要到程序六到达该对象定义位置时才初始化。故而若在初始化前就跳出，则留下一段未初始化的空间，可能导致错误。
- default ctor
    - called automatically when objects goes out of scope
    - the closing brace of the scope

```cpp
class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  cout << "inside Tree constructor" << endl;
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}

int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
}
/*output:
before opening brace
inside Tree constructor
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace */
```

- 类内成员变量本质是field字段。作用范围和生命周期伴随其依附的整个对象。所以往往长于局限在调用函数段内的形参和生命代码段内的局部变量。

function overload

- 遵循最佳匹配原则
- 可以给默认值，顺序严格从左到右赋值，不能出现先赋值再不赋值。

Const Object

- const对象： 前加const即可。**初始化后** *不能* 改变的对象。即初始化中的赋值是可以的。
- const成员变量： 不可改变的成员变量。故而初始化赋值也不行，只能使用初始化列表初始化。
- const成员函数： 不改变成员变量（对象状态）的函数。形式：void X() **const** { };
- 限制：
  - const对象只能调用const成员函数
  - 非const对象可以调用const和非const成员函数

inline function

- 纯粹的编译器实现代码复制粘贴，类似空间换时间
- 定义要写在.h文件里

passing parameters

- pass in object: if store it inside function
- pass in reference: if want to do something
- pass in const pointer: if only get value
- pass out object: if create it in the function
- pass out reference or the pointer of the passed in only
- never new something and return the pointer.

class embedded

- all embedded object must be initialized
- 嵌套类的列表初始化：

```
class A {
public:
    int x;    std::string y;
    A(int num, std::string str) : x(num), y(str) {}
};

class B {
public:
    double z;    A w;
    B(double num1, int num2, std::string str) : z(num1), w(num2, str){}
};

void foo() {    B b(1.2, 3, "example");    }
// 这里我们在初始化B类实例的时候，直接传入初始化A类需要的参数
```

- 嵌套类的private/public
  - 嵌套类在类的private：任何成员变量/函数均不可被外部函数调用

    - 嵌套类在类的public： public可被外部函数调用，private不可
    - 不管嵌套类在何处，要调用嵌套类的private，都必须要调用嵌套类，用嵌套类去调用其private。即，仅当嵌套类被放在public时在外部函数中才能通调用该嵌套类来调用其private。

inheritance

- public、private、protected：
    - public: to all
    - private: to self and friend
    - protected: to self and friend and derived
- 不被继承：ctor，dtor，赋值运算符，private
- 继承类可以`:public` `: private` `: protected` 三种方式继承，分别为公共继承、私有继承，受保护继承。
    - 这三种继承类都可以访问基类的public和protected，都不能访问private
    - `: private` 私有继承，将基类的所有量(pub/pri/pro)都变成private。故而私有继承类的子类不能访问基类的任何量(pub/pri/pro)
    - `: protected` 受保护继承，将基类的pub/pro都变成protected。故而私有继承类的子类仍能访问基类的pub/pro，但其他代码不能从B的方式访问A的pub。
    - `: public` 公共继承，基类的pub和pro权限不变。故而继承类的子类仍能访问基类的public和protected，其他代码能从B的方式访问A的pub。

```
class A {
public:    int pub = 1;
protected:    int pro = 2;
};
class B : protected A {
public:    int pubB = 5; int access(){return pub+pro;}
};
class C : public B {
public:       int accessC(){return pub+pro;}
}
void foo() {
    A a;    B b;    C c;
    std::cout << a.pub << " ";
    // std::cout << a.pro << "\n";
    // std::cout << b.pub << "\n";
    //wrong! B中的A:pub已经是prot了。故而无法通过B访问pub
    std::cout << b.access() << " " << c.accessC() << " " << c.pubB;
    //但B仍能访问A的pub和prot，C仍能访问A的pub和prot。但布恩那个通过c.pub调用A的pub，因为对B和C而言A.pub是prot。但可以用过c.pubB调用B中新的public量。        output: 1 3 3 5
    return 0;
}
```

- 继承类的构造：
    - 先构造基类，再构造子类。其余顺序同声明顺序
    - 如果没有给基类参数，则基类用默认构造。

    - dtor顺序与ctor相反

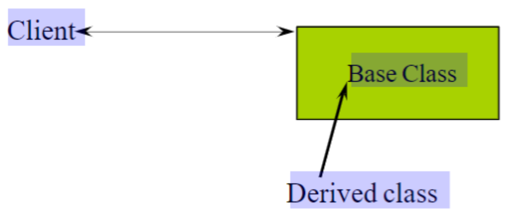
```
class A {
public:
    A() { i = 10; cout << "A()" << i<< endl; }
    ~A() { cout << "~A()" <<endl;}
    int i;
};
class B : public A {
public:
    B() { i = 20; cout << "B()" << i << endl; }
    ~B() {cout << "~B()" <<endl;}
    int i;
};
void foo() {
    cout << sizeof(A) << " " << sizeof(B) << endl;
    B b;
    int* p = (int*) &b;
    cout << p[0] << " "<< p[1] <<endl;
}
//output:
4 8     A()10 B()20    10 20    ~B() ~A()
```

- name hiding
    - 派生类和基类若有同名函数，则无论参数是否相同，基类都会被派生类掩盖。此时若想调用基类中同名不同参的函数，会导致编译器报错。
    - 用using 的方式来显示被掩盖的函数。

```
class Base {
public:
    void foo(int) {std::cout << "Base::foo(int)" << std::endl;}
};
class Derived : public Base {
public:
    using Base::foo;  // 让 `Base::foo(int)` 在 `Derived` 中可见
    void foo() {std::cout << "Derived::foo()" << std::endl;}
};
void foo() {
    Derived d;    d.foo(42);    d.foo();
    //前者调用base， 后者调用derived
    //但若去掉using Base::foo，则前者会编译报错。被彻底掩盖。
}
```

When is protected not protected?

- When your derived classes are ill-behaved!
- Protected is public to all derived classes
- For this reason
  - make member *functions* protected
  - keep member *variables* private



Friend

Up casting

- it's safe to up casting from a derived class to base class.

```
class Base { /* … */ };
class Derived : public Base { /* … */ };
Derived d;
Base* pb = &d;  // upcasting
```

Copy Constructor

- when apply a parameter transfer in function, the parameter is copied -- **create a new object from an existing one**

```
void func(Type p){...}
Type a;
func(a);    //a is copied to p above.
```

- 当没有拷贝构造函数时，系统会自动生成。
  - 对于数字、对象、数组等非动态分配内存的数据，深拷贝
  - 对于动态内存分配的指针，浅拷贝
    - cstring::strlen -- return the length
    - cstring::strcpy -- copy until /0 (include)
  - 嵌套类——成员逐一初始化。对每个对象递归的调用其拷贝构造
- 拷贝构造的调用时刻
  - 按值调用

```
void func(Person x);
Person p1();
func(p1);//call copy ctor to copy p1 to x in func.
```

  - 初始化

    ```
    Person p1("Fred");
    Person p2(p1);
    Person p3 = p2;
    //all are initialize instead of assignment. using copy ctor.
    ```

  - 函数返回

```
Person func(){
    Person p1("Leo"); return p1;
}
Person p2 = func();
```

- 拷贝构造只能进行一次，赋值可以多次。拷贝构造需要delete。
- 大多数情况，可以直接用系统默认。对于动态指针，需要自定义。当不需要拷贝构造，写一个空的拷贝构造以避免生成默认导致编译错误。

STL

1. Containers

vector, deque, list, forward_list, array, string

- when using iterator of list, **don't use** `p < s.end()` as vector given that the elements may not be stored in order. **use** `p != s.end()`

2. Algorithms

3. Iterators

- declaring: Container::iterator varname;
- Front: varname = obj.begin();
- end: varname = obj.end();
- can in/decrement or assignment

Overload OP

- restrictions
  - can only overload existing operators
  - overload preserve the number of oprands
  - overload preserve the precedence of operation
  - `.` `.*` `::` `?:` `sizeof` `()_cast` can't be overload
- overload op is a member of class function
  - the first argument can be implicit
  - 调用重载的对象(可隐式)不能进行类型转换

```
Integer x(1), y(2), z;
z = x + y;
z = x + 5;    //ok, automatically convert the second
z = 3 + x;    //wrong. the first is not allowed to convert
```

- overload op is a global function
  - the first member should be explicit
  - need to be arguments' class's friend if using pri/prot instead of pub

```
z=x+y;// operator+(x, y)
z=x+3;// operator+(x, Integer(3))
z=3+y;// operator+(Integer(3), y)
z=3+7;// Integer(10)
```

- argument passing:
  - for member function, use const don't change the class
  - for global function, use reference if left hand change
- return value:
  - return the const type for operators like +.
  - return bool or int for operators like =.
  - **return reference for []** , which could only be member function
- as for i++, ++i, i--, --i
  - ++i -- `::operator++()`
  - i++ -- `::operator++(int)`
- add const at the end of op overload of relation, given it don't change the value of the arguments
- as for assignment =
  - could only be member function.
  - **return reference !**

  ```
  A = B = C; // executed as A = (B = C)
  ```

  - **check not equal to itself !** (if use dynamically allocated)

  ```
  MyClass& operator=(const MyClass& other) {
      if (this != &other) { // Protect against self-assignment
          // Allocate new memory and copy the elements
          int* newData = new int[other.size];
          std::copy(other.data, other.data + other.size, newData);
          delete[] data; // Free the old memory
          data = newData; // Assign the new memory
          size = other.size;
      }
      return *this;
  }
  ```

- automatically create if not explicitly provided.
    - *memberwise assignment* . **默认赋值重载= 和默认拷贝构造类似，对于嵌套类等，会递归地对每一个成员默认生成赋值重载=来单个赋值**

```
class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl; return *this;
  }
};
class Truck { Cargo b; };
void foo() { Truck a, b; a = b; }
// Prints: "inside Cargo::operator=()"
```

- functor () ?

Type Conversion

- conversion operator
- compiler perform implicit convert by
    - single argument constructor

    ```
    class One {public:    One() {}    };
    class Two {public:    Two(const One&) {}    };
    void f(Two) {}
    int main() {    One one;    f(one);    }
    //Wants a Two, has a One, implicitly use two's construction
    ```

    - implicit type conversion operator
- explicit: inhibit the implicit use of the construction.

    ```
    class One {public:    One() {}    };
    class Two {public:    explicit Two(const One&) {}    };
    void f(Two) {}
    int main() {
      One one;
    //f(one); // No auto conversion allowed
      f(Two(one)); // OK -- user performs conversion
    }
    ```

- **conversion operator:** general form-- `X::operator T()``X`
    - 没有显式参数，隐式地“接受”类的当前对象作为参数;没有返回类型，返回类型是正在定义的T类型。
    - 当编译器发现要从`X`到`T`的转换，会自动地使用这个函数
    - 如果不希望发生隐式的类型转换，可以函数前加`explicit`

```
class Orange; // Class declaration
class Apple {
public:
  operator Orange() const; // Convert Apple to Orange
};
class Orange {
public:
  Orange(Apple); // Convert Apple to Orange
};
void f(Orange) {}
int main() {    Apple a;    /* f(a);*/    }
// Error: ambiguous conversion. from apple to orange and to apple and to
orange and loop. if insist to do like this, add explicit before constrution
and use explicitly in f
```

Casting

- `static_cast`:
  - 把一种类型静态地（在编译时）转换为另一种类型
  - 可以非const转const，不能const转非const
  - 不能downcast转换（不安全）

```
int e = 10;
const int f = static_cast<const int>(e);//correct
const int g = 20;
int *h = static_cast<int*>(&g);
//error: static_cast can not remove the const property
Class A {public: virtual test() {...}}
Class B: public A {public: virtual test() {...}}
A *pA1 = new B();
B *pB = static_cast<B*>(pA1);//downcast not safe
```

- `dynamic_cast`：用在含有虚函数的类的对象指针之间的转换，依赖于类型信息，在运行时进行检查。如果对象类型与目标类型不符合，返回`null`。
  - 此处**downcast** 联系 继承中的 **upcast** 。downcast不安全因为派生类有基类没有的内容。而upcast安全因为基类的内容派生类都有。

```
class A {public: virtual int test() {...}}
class B: public A {public: virtual int test() {...}}
class C {public: virtual int test() {...}}
A *pA1 = new B();
B *pB= dynamic_cast<B*>(pA1);  //safe downcast
C *pC= dynamic_cast<C*>(pA1);  //not safe, will return a NULL pointer
//B 是 A 的派生类， C 与 A，B 无关
```

- `const_cast`：用于修改类型的`const`或`volatile`属性。例如，使用`const_cast`去掉`const`属性，从而改变一个`const`对象的值。
- `reinterpret_cast`：基本不考虑类型，几乎可以转换任何类型到任何其它类型，包括指针到整型、整型到指针、引用到指针等等。在进行此类转换时需要格外小心，因为不安全的使用可能会导致严重错误。

    - 不能同类转同类

Operator reload

返回class 而非void： 连续赋值

返回引用而非值：

- 避免临时变量的拷贝构造
- 左值逻辑正确

inheritance overload

- 基类派生类可以有同名变量。派生覆盖基类。
- 若基类有对同名变量操作的函数，当调用该基类函数时，操作的是基类的对象。
- 多态中函数的初始化参量静态绑定。即使多态派生类，初始化变量仍是父类。
- 相同函数名不同参列表，子类函数也会掩盖父类。

explicit 拒绝隐式调用

常出现于初始化与赋值阶段用常数做=初始化