Object – Oriented Programming

Week 3, Spring 2009

Object

Weng Kai

http://fm.zju.edu.cn

Wednesday, 4 Mar., 2009

Topics today

- Constructors and destructor
- new and delete
- string
- access control

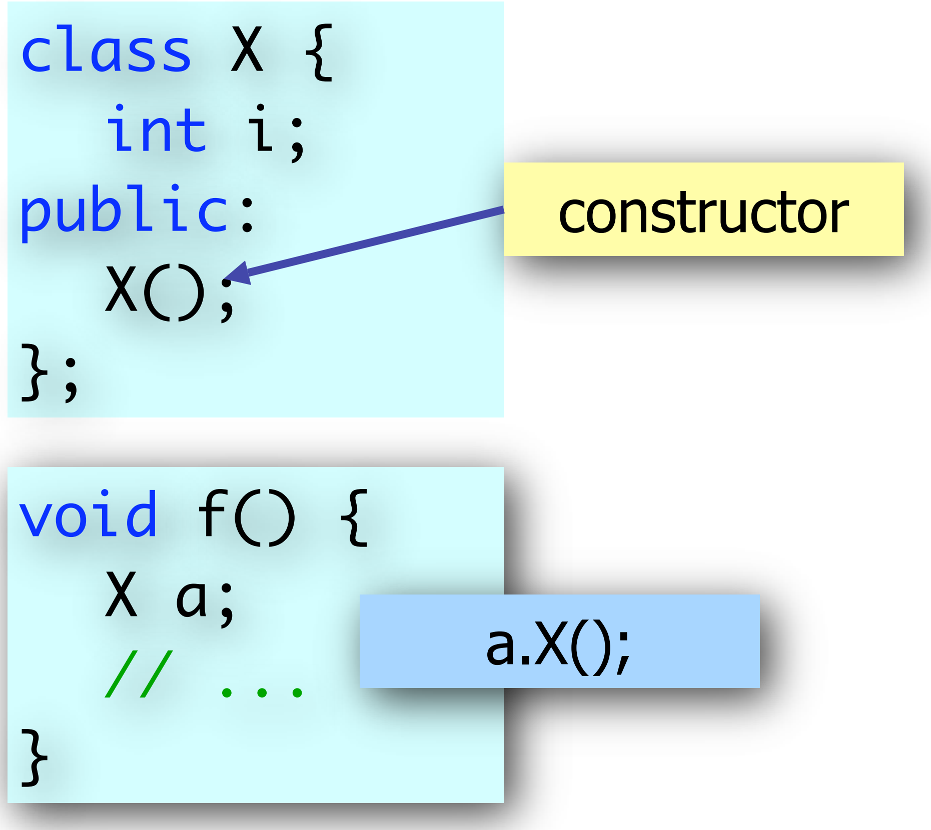
Point::init()

```
class Point {
public:
    void init(int x,int y);
    void print() const;
    void move(int dx,int dy);

private:
    int x;
    int y;
} ;

Point a;
a.init(1,2);
a.move(2,2);
a.print();
```

Guaranteed initialization with the constructor

- If a class has a constructor, the compiler automatically calls that constructor at the point an object is created, before client programmers can get their hands on the object.
- The name of the constructor is the same as the name of the class.

How a constructor does?



```
class X {
   int i;
public:
   X();
};
```

constructor

```
void f() {
   X a;
   // ...
}
```

a.X();

Constructors with arguments

- The constructor can have arguments to allow you to specify how an object is created, give it initialization values, and so on.

```
Tree(int i) {…}

Tree t(12);
```

- Constructor1.cpp

The destructor

- In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor.
- The destructor is named after the name of the class with a leading tilde (~). The destructor never has any arguments.

```
class Y {
public:
  ~Y();
};
```

When is a destructor called?

- The destructor is called automatically by the compiler when the object goes out of scope.
- The only evidence for a destructor call is the closing brace of the scope that surrounds the object.

Storage allocation

- The compiler allocates all the storage for a scope at the opening brace of that scope.
- The constructor call doesn't happen until the sequence point where the object is defined.
  - Examlpe: Nojump.cpp

Aggregate initialization

- `int a[5] = { 1, 2, 3, 4, 5 };`
- `int b[6] = {0};`
- `int c[] = { 1, 2, 3, 4 };`
  - `sizeof c / sizeof *c`
- `struct X { int i; float f; char c; };`
  - `X x1 = { 1, 2.2, 'c' };`
- `X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };`
- `struct Y { float f; int i; Y(int a); };`
- `Y y1[] = { Y(1), Y(2), Y(3) };`

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

```
Y y2[2] = { Y(1) };
Y y3[7];
Y y4;
```

“auto” default constructor

- If you have a constructor, the compiler ensures that construction *always* happens.
- *If* (and only if) there are no constructors for a class (**struct** or **class**), the compiler will automatically create one for you.
  - Example:AutoDefaultConstructor.cpp

Dynamic memory allocation

- **new**
  - `new int;`
  - `new Stash;`
  - `new int[10]`
- **delete**
  - `delete p;`
  - `delete[] p;`

new and delete

- new is the way to allocate memory as a program runs. Pointers become the only access to that memory
- delete enables you to return memory to the memory pool when you are finished with it.
- Example: use_new.cpp

Dynamic Arrays

```
int * psome = new int [10];
```

- The new operator returns the address of the first element of the block.

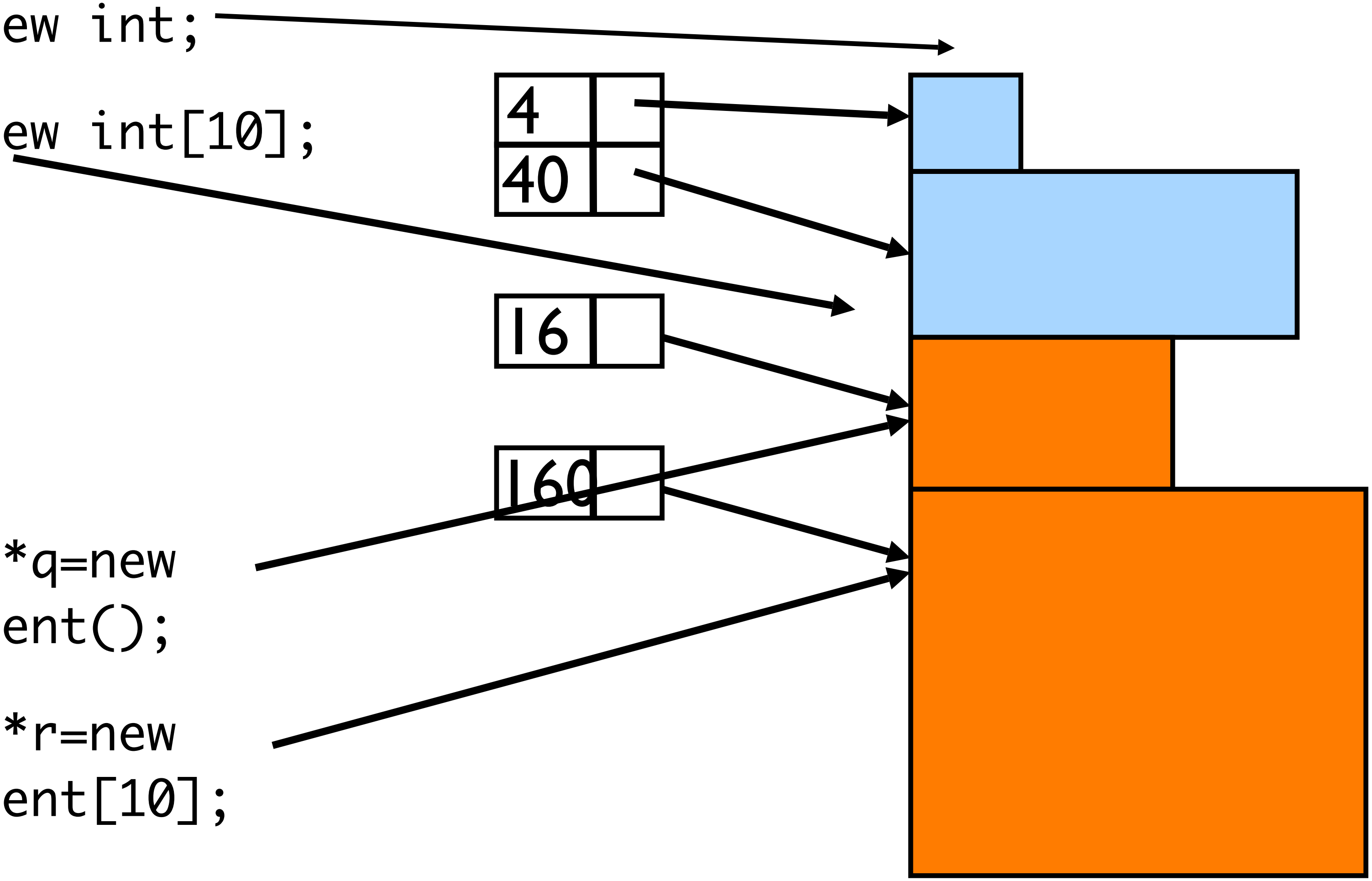
```
delete [] psome;
```

- The presence of the brackets tells the program that it should free the whole array, not just the element

- Example: arraynew.cpp

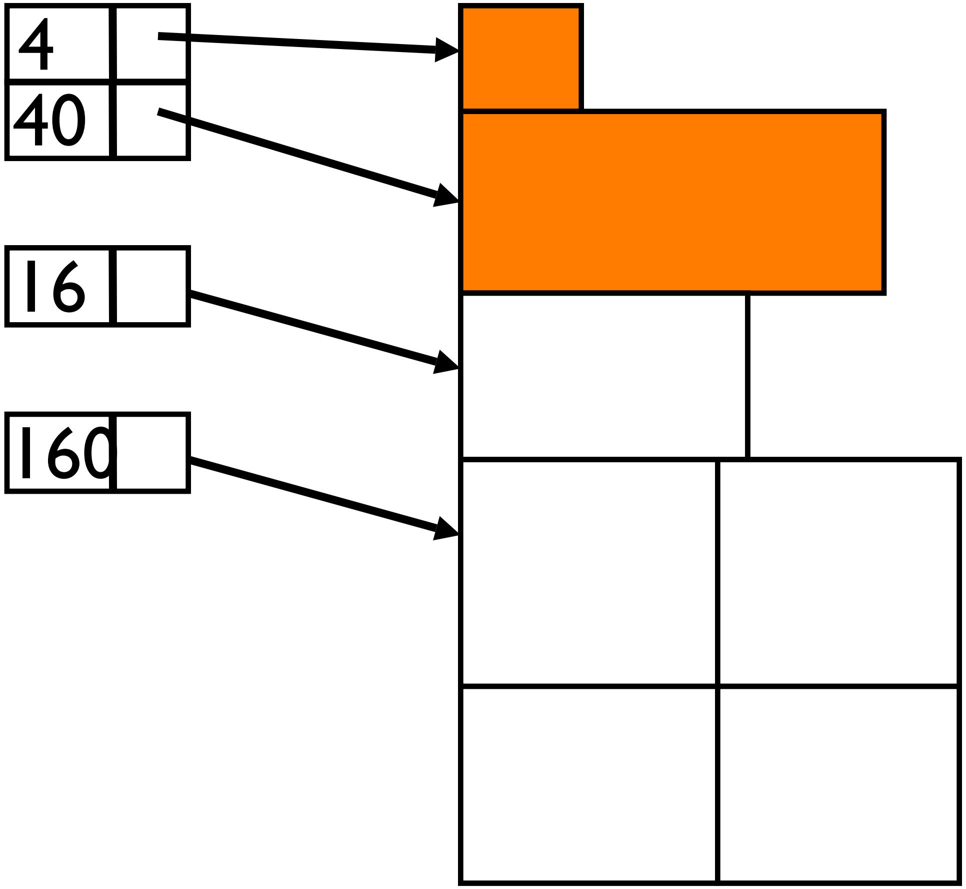
The new-delete mech.

```
int *p=new int;

int *a=new int[10];

Student *q=new
    Student();

Student *r=new
    Student[10];
```



The new-delete mech.

```
int *p=new int;

int *a=new int[10];

Student *q=new
    Student();

Student *r=new
    Student[10];

delete p;

a++;delete[] a;

delete q;

delete r;

delete[] r;
```



Tips for new and delete

- Don't use delete to free memory that new didn't allocate.
- Don't use delete to free the same block of memory twice in succession.
- Use delete [] if you used new [] to allocate an array.
- Use delete (no brackets) if you used new to allocate a single entity.
- It's safe to apply delete to the null pointer (nothing happens).

The string class

- You must add this at the head of you code
  - `#include <string>`
- Define variable of string like other types
  - `string str;`
- Initialize it w/ string contant
  - `string str = “Hello”;`
- Read and write string w/ cin/cout
  - `cin >> str;`
  - `cout << str;`

Assignment for string

```
char charr1[20];
char charr2[20] = “jaguar”;
string str1;
string str2 = “panther”;
carr1 = char2;   //  illegal
str1 = str2;   //  legal
```

Concatenation for string

- `string str3;`
- `str3 = str1 + str2;`
- `str1 += str2;`
- `str1 += “lalala”;`

Setting limits

- to keep the client programmer’s hands off members they shouldn't touch.
- to allow the library designer to change the internal workings of the structure without worrying about how it will affect the client programmer.

C++ access control

- The members of a class can be divided into some parts, each of the parts is marked as:
  - **public**
  - **private**
  - **protected**

public

- **public** means all member declarations that follow are available to everyone.
  - Example: Public.cpp

private

- The **private** keyword means that no one can access that member except inside function members of that type.
  - Example: Private.cpp

Friends

- to explicitly grant access to a function that isn’t a member of the structure
- The class itself controls which code has access to its members.
- Can declare a global function as a **friend**, as well as a member function of another class, or even an entire class, as a **friend**.
  - Example: Friend.cpp

class vs. struct

- **class** defaults to **private**, whereas **struct** defaults to **public**.

- Example: Class.cpp