

Announcements

- **C project due Monday (April 5)**
- **Office hours in ARC**
 - **Thursday 2-5pm (Prof DeCarlo)**
 - **Friday 1-4pm (George Sakkis)**

1

C++ Class Example

```
#include <stdio.h>
#include <stream.h>
class vector
{
    int sz;
    int *v;
public:
    vector(int); constructor
    ~vector() {delete v;} destructor
    int size() const {return sz;} member
    int& elem(int i) const {return v[i];} functions
    int& operator[](int) const;
};
```

C++ allows overloading of operators:
e.g., subscript operator []

2

C++ Class Example, cont.

```
void error(char *s)    procedure
{ cout << s << "\n";
  exit(1);
}
vector::vector(int i)  constructor implementation
{ if (i <= 0) error("bad vector size");
  sz = i;
  v = new int[i];
}
int& vector::operator[](int i) const
{ if (i < 0 || i >= sz) error("index out of bounds");
  return v[i];
}
```

overloaded subscript operator code
does bounds check so is safe; elem()
is not safe.

3

Implementation Reuse through Inheritance

Continues same C++ program

```
class vec : public vector
{
  int high, low; //private members of class
public:
  vec(int, int);
  int& elem(int i)    { return vector::elem(i - low);}
  int& operator[](int);
};
vec::vec(int i, int j) : vector(j - i + 1)
{
  if (j < i) j = i;
  low = i;
  high = j;
}
int& vec::operator[](int i)
{
  if (i < low || i > high)
    error("index out of bounds for vec");
  return elem(i);
}
```

Explicit call to base
class constructor with args;
Initializes superclass

4

Multiple Constructors

```
class newvec : public vector
{
public:
    newvec(int s) : vector (s) {}
    newvec(const newvec &);
    ~newvec() {}
    void operator=(newvec &);
};
// implementation of second constructor
newvec::newvec(const newvec &a) : vector(a.size())
{
    for (int i = 0; i < a.size(); i++)
        elem(i) = a.elem(i);
}
```

overloading

Copy constructor

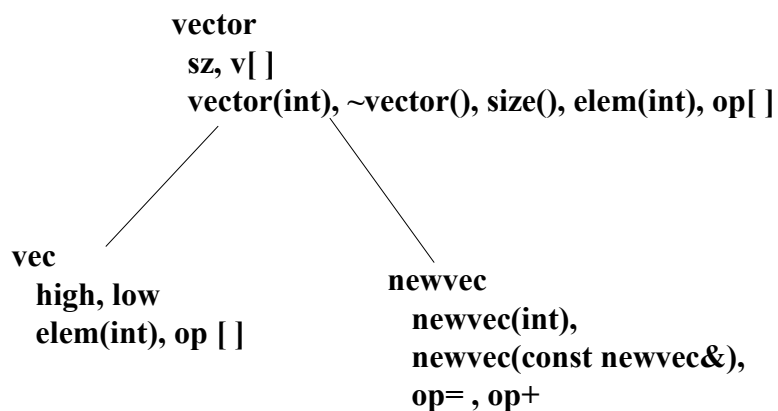
5

Using Overloaded Operators

```
// assignment operator
void newvec::operator=(newvec &a)
{
    int i;
    if (size() != a.size())
        error("bad vector size for =");
    for (i = 0; i < size(); i++) elem(i) = a.elem(i);
}
int main()
{
    int i;
    newvec v1(10);
    for (i = 0; i < 10; i++) v1[i] = i;
    newvec v2 = v1;
    newvec v3 = v1 + v2;
    ...
}
```

6

Hierarchy



7

Visibility in C++

- **For members and member functions**
 - Private - only visible within that class
 - Protected - only visible within that class and its derived classes
 - Public - visible to everyone
- **Public inheritance is used in most cases**

8

How is inheritance useful?

- **Reuse by inheriting implementation, modifying and/or extending it**
 - E.g., `vector` and `newvec` (inherit);
`vector` and `vec` (extending)
- **Allows enforcement of same public interface to objects in all subclasses (same method name and signature even if they do different things in their implementation)**
 - E.g., `elem()` in `vector` and `vec`

9

How is inheritance useful?

- **Can create superclasses by factoring out common data and actions from subclasses**
 - E.g., putting `size` in `vector`
 - Facilitates changes by localizing effects
 - Can avoid clerical errors if meaning of a same-named entity changes

10

Parametric polymorphism: templates in C++

```
template <class T>
class vector {
    int sz;
    T *v;
public:
    vector(int i) : sz(i) { v = new T[i]; };
    ~vector()          { delete v; }
    int size() const   { return sz; }
    T& elem(int i) const { return v[i]; }
};
...
vector<double> v(3);
v.elem(0) = 3.14;
```

11

Templates for polymorphic functions

```
// Square a value of type T
template <class T>
inline T sqr(const T &x)
{
    return x*x;
}

// Sign of a value of type T: one of {-1, 1}
template <class T>
inline T sgn(const T &x)
{
    return (x < T(0)) ? T(-1) : T(1);
}
```

12

Object Creation

Dynamic: `vector *v = new vector(10);`
`vec *w = new vec(10);`

Static: `vector a(10);`
`vec b(10);`

Differences - object created dynamically, with a pointer, in C++ are stored on heap; otherwise, they are stored on a stack frame

Be careful: `v = w;` `a = b;` are legal but with different effects. `v=w` makes `v` point to a `vec` object; `a=b` truncates the `vec` object data fields which don't belong to `vector` to fit in the stack storage!

Always create variable-sized objects using pointers in C++

13

Virtual Functions

- Dynamic binding only happens in C++ with functions declared to be *virtual*

```
Define void vector::printv() {..}  
void vec::printv() {..}
```

Declare `vector *v, vector *vv, vec *w`

Initialize `v, vv, w` to point to objects of their declared types

Execute

```
vv=w;  
v->printv();  
vv->printv();
```

Both calls will execute `vector::printv()`

Choice of function based on declared type of pointer
because methods not declared to be virtual

14

Virtual Fcns, Example

```
...
class A {
public:
    virtual void print() { cout << "inside A\n"; }
};
class B : public A {
public:
    void print()          { cout << "inside B\n"; }
};
main()
{
    A *pa = new A();
    B *pb = new B();
    pa->print(); // prints inside A
    pb->print(); // prints inside B
    *pa = *pb;
    pa->print(); // prints inside A
    pa = pb;
    pa->print(); // prints inside B
}
```

15

Abstract classes in C++

```
...
class A {
public:
    virtual void print() = 0;
};
class B : public A {
public:
    void print()          { cout << "inside B\n"; }
};
main()
{
    A *pa; // Note: cannot create an object of type A
    B *pb = new B();
    pb->print(); // prints inside B
    pa = pb;
    pa->print(); // prints inside B
}
```

16

Inheritance in C++

- **Use abstract class to create consistent interfaces for subclasses**
 - Promises an implementation for every non-abstract subclass (although this implementation can be inherited)
- **Have subtype polymorphism if never redefine inherited functions**
- **Code sharing and reuse**
- **Automatic propagation of changes to subclasses**
- **C++ has no equivalent to Java *final* which prevents subclass extension**
- **Benefits to class designer and users**

17

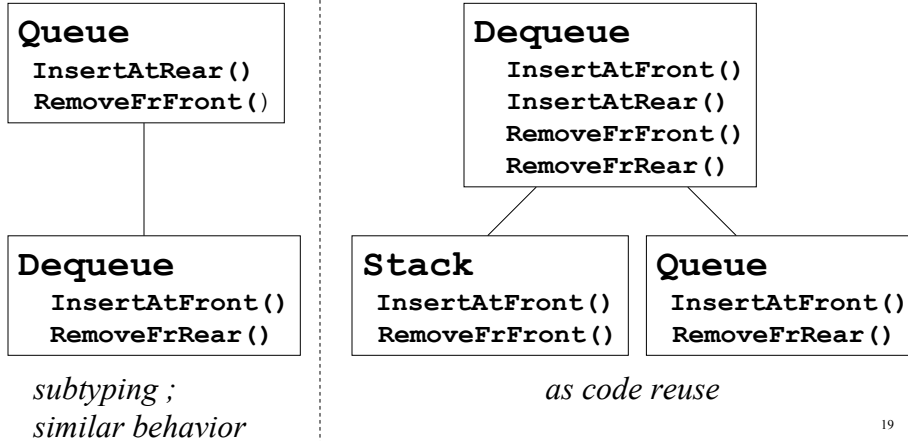
Inheritance

- **As subtyping**
 - Inheriting implementation and external specification
 - S is subtype of T if all operations on type T objects are meaningful on S objects; substitutability in behavior
- **As code reuse**
 - Inheriting only implementation; not necessarily an *is-a* relation
 - Building new components from old

18

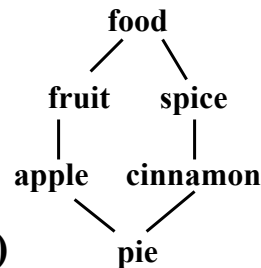
Examples

- Two ways to define Queue and Dequeue



Multiple inheritance

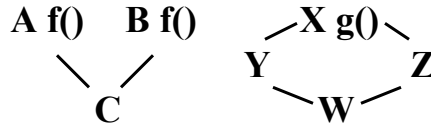
- Real world is multiple
- Interpretation depends on non-local structure (so it's not robust in the face of changes)
- Conflicts are *very* messy
- Often forbidden in industry because of these issues (exception: classes containing only virtual methods: interfaces)



Multiple inheritance conflict resolution

- **Problems:**

- Member clash
- Inheriting more than one copy of same member



- **Approaches:**

- Linearize hierarchy so only one parent is “closest”
- Throw an exception when same member is applied more than once due to duplicate paths
- Exclude some members to avoid problem (C++ virtual base classes)

21