

CMSC433, Spring 2001

Programming Language Technology
and Paradigms

1

C++

2

C++ without Classes

- Don't need to say "struct"
- New libraries
- function overloading
 - confusing link messages
- default parameters
- void *
 - C++ requires explicit conversion from void *

3

More C++ without Classes

- Dynamic initialization of globals
 - no way to control order between files
- References
 - Compiler treats as pointers
 - Programmer doesn't
 - Once initialized to reference an address, cannot be made to reference a different address
 - Used for pass-by-reference parameters

4

References

```
int x,y;
int *p = &x;
int &q = y;
q++; // increments y
*p++; // increments p, returns value of x
(*p)++; // increments x
q = x; // assigns value of x to y
p=&y; // makes p point to y;
```

5

new/delete

- use new A to allocate one A
- use new A[num] to allocate an array of num A's
- use delete to free one A
- use delete[] to free an array of A
- invokes constructors and destructors
 - for all elements of an array
 - delete[] must be given pointer to first element

6

Data Abstraction

- Avoid name clashes
- Hide implementation
- Override “standard” functionality

```

class IntArray{
public:
    IntArray();
    ~IntArray();
    void setSize(int value);
    int getSize();
    void setElem(int index, int value);
    int getElem(int index);
    ...}

void IA_init(IntArray *ia);
void IA_cleanup(IntArray * ia);
void IA_setSize(IntArray * ia, int value);
int IA_getSize(IntArray * ia);
void IA_setElem(IntArray * ia,
                int index, int value);
int IA_getElem(IntArray * ia, int index);
    
```

7

Int Array example

- Without OO:


```

void IA_init(IntArray *ia);
void IA_cleanup(IntArray * ia);
void IA_setSize(IntArray * ia, int value);
int IA_getSize(IntArray * ia);
void IA_setElem(IntArray * ia,
                int index, int value);
int IA_getElem(IntArray * ia, int index);
            
```
- With OO:


```

class IntArray{
public:
    IntArray();
    ~IntArray();
    void setSize(int value);
    int getSize();
    void setElem(int index,
                int value);
    int getElem(int index);
private:
    ...}
            
```

8

Access control

- A member can be public/protected/private
- An access specification controls access to all following members (until the next access specification)
- For classes, initial/default access is private
 - For structs, initial/default access is public
 - no other difference between classes and structs

9

Access levels

- public - anyone is allowed access
- private - only methods in the class allowed access
- protected - like private, except that subclasses can see inherited members

10

Protected access

- Class A {
 - private: int x;
 - protected: int y; }
 - Class B : public A {
 - static int foo(A a, B b) {
 - int i = a.x; // illegal
 - int j = a.y; // illegal
 - int k = b.x; // illegal
 - int n = b.y; // LEGAL
 - ... }

11

Friends

- A class X can declare a class Y or a function f as a friend
 - Gives Y or f access to all of X’s private and protected data
- Friendship is not transitive
 - If X declares Y as a friend, and Y declares Z as a friend, Z can’t see X’s private data

12

Hidden functions

- class A {
 A(); // void constructor
 A(const A &); // copy constructor
 A& operator=(const A &) // operator=
 ~A(); // destructor
}

13

void constructor

- Invoked whenever an instance is created without being initialized.
- If no constructor is provided, default public one is created
 - invokes void constructor on each base class and instance variable

14

copy constructor

- Invoked to create a new value from an old value
 - parameters passed by value
- If no copy constructor supplied, default public one created
 - invokes copy constructor for each base class and instance variable
 - almost never correct if you have pointers

15

operator=

- If no operator=() defined, defines default public one
 - does operator= on each base class and instance variable
- Watch for a = a
- A::operator=(const A&) usually returns A&
 - allows a = b = c;

16

Pointer obligations

- If you have a member variable that is a pointer, the default functions are almost certainly wrong
- For each pointer, decide if it is the unique pointer to an object
 - if not, need to control updates of that object and figure out when to free referenced object
 - if so, need to duplicate object rather than pointer

17

Pointer obligations

```
class IntArray {  
    int * rep;  
    int size;  
public:  
    IntArray(int sz) : rep(new int[sz]), size(sz) {};  
    IntArray(const IntArray &a)  
        : rep(new int[a.size]), size(a.size)  
        { memcpy(rep,a.rep,sizeof(int)*a.size); }  
    ~IntArray() {  
        delete [] rep;  
    }  
};
```

18

operator =

```
IntArray& operator=(const IntArray &a) {  
    if (this == &a) return *this;  
    if (size != a.size) {  
        delete [] rep;  
        rep = new int[a.size];  
        size = a.size;  
    }  
    memcpy(rep, a.rep, sizeof(int)*size);  
    return *this;  
}
```

19

One arg constructors

- A one argument constructor is also used for type conversion
 - A(int sz) allows an int to be used anywhere an A is expected
 - only one level of conversion will be required
 - if a B is required and an int is supplied, the system won't convert an int to an A, which is then converted to a B
 - Can mark constructor as `explicit` to avoid this feature

20

Subtyping

- if class D has class B as a public base type
 - a pointer to a D can be provided anywhere a pointer to a B is expected
 - a reference to a D can be provided anywhere a reference to a B is expected
- class D should fulfill B's public contract
 - Someone who expects a pointer to a B
 - and is given a pointer to a D
 - shouldn't be surprised

21

Virtual functions

- Without virtual functions, B's functionality will be invoked on objects pointed to by a B pointer
 - determined from type of pointer
- with virtual functions
 - dispatching based on run-time type of object pointed to
 - and compile-time type of arguments
 - must match argument types exactly to override

22

Non-virtual functions

- non-virtual functions have confusing semantics
 - function you get depends on type of pointer used to reference the object
- but virtual functions may be more expensive to invoke
- should probably make functions non-virtual only if never overridden
 - final in Java

23

Watch out !

- An array of D's can be given to someone
 - who expects an array of B's
- B's public contract may involve assigning another B to it
- What is the right thing to do
 - when a D is assigned a B?
 - when a B is assigned a D?

24

Operator= and subtyping

- struct B {
 B& virtual operator=(const B & b) { ... } };
- struct D : public B {
 D & virtual operator=(const D & d) { ... } };
- B b; D d; B* pbb = &b; B* pbd = &d;
 - b = d; // B::operator=(const B&)
 - d = b; // Illegal (why?)
 - *pbb = *pbd; // B::operator=(const B&)
 - *pbd = *pbb; // B::operator=(const B&)

25

Covariance of return types

- When you override a function, your return type can be a subtype of the method you are overriding
 - struct B {
 virtual B* getChild();
 }
 - struct D : public B {
 virtual D* getChild();
 }

26

Covariance

- Covariance of return types is OK
 - Somebody who expects a B*, is given a D*
 - invokes getChild() on it
 - Expects a B*, is given a D*
 - No surprise...
- Not OK for argument types
- Co-variance = (in the same way)(vary)
 - as opposed to contravariance
 - allowed in some OO languages for arguments, but not that useful

27

Argument types can't be covariant

- Consider if D::f overrode B::f in
 - struct B { virtual void f(B* p); }
 - struct D : public B { virtual void f(D* p); }
- Somebody who expects a B*, is given a D*
 - invokes f(new B()) on it
 - D::f is surprised!
 - expected a D*, got a B*

28

Function hiding

- Consider

```
class B {
  public: int f(int i);
};
class D : public B {
  public: int f(char *);
};
```
- f(int) not defined on a D

29

Function hiding rules

- Defining a function f in a derived class D
- hides any function named f with different arguments in a base class B
- Function can still be invoked
 - by using a pointer/reference of type B

30

operator= example

- virtual declaration didn't matter
 - arguments not identical
- Declaration of operator=(const D&)
 - hides operator=(const B&)
- To make it visible, re-declare
 - struct B {
 B& virtual operator=(const B & b) {...} };
 - struct D : public B {
 D & virtual operator=(const B & b) {...};
 D & virtual operator=(const D & d) {...};

31

similar problems elsewhere

- Copy constructor
 - overriding not an issue
 - watch out for omitting & for a reference parameter
- operator==
 - almost identical set of problems to operator=

32

WHY?

- Common situation:
 - defining a binary operator for a class B
 - defined as member function
 - when redefined for derived class D, should take a D as an argument
 - don't want to "accidentally" invoke the B version on a D

33

Object Hierarchy

- composition/aggregation
 - have components as instance variables
- derivation
 - have components as base classes

34

Engine vs. Car

- Should a Car have an Engine as a component or as a base type?

35

Animal class

- Say we have classes for Fish, Dog, Eagle, Whale and Hippo.
- Should they have a common base class?
- What methods should it support?
 - age()?
 - speed()?
 - fly()?
 - eat()?

36

Heterogeneous collections

- Will you ever need a collection that contains objects that all is-a/has-a/are-a A
 - If so, use is-a
- Will you need a collection of engines and cars?
- Will you need a collection of animals?

37

Casts in C++

- `static_cast`
- `dynamic_cast`
 - cast from base class ptr to derived class ptr
 - returns null if not instance of derived class
- `const_cast`
- `reinterpret_cast`
 - interpret bits

38

Casting between classes

- In C++ with multiple inheritance, casting between a ptr to a base type and a ptr to a derived type
 - may require adding/subtracting offset
 - may require run-time lookups
 - if you have virtual base classes
- Reusing the bit pattern likely to be badly wrong

39

`static_cast`

- For pointers, assumes correct, doesn't look at object
 - `B *b = static_cast<B *>(d);` // always OK
 - `D *d = static_cast<D *>(b);` // ?
- Also used for casting between integers, doubles, and out of `void *`

40

`dynamic_cast`

- Looks at object to determine:
 - if legal (returns null otherwise)
 - adjustment needed
- Generally used for safely casting from base type to derived type
- Requires that code be compiled with RTTI
 - Run time type information

41

`const_cast`

- Can only remove/add `const/volatile` declarations

42

reinterpret_cast

- interpret bits
- VERY dangerous, no checks
- No adjustments applied

43

Abstract classes

- An abstract class cannot be directly instantiated
- Although subclasses can
- In C++, any class with an abstract function is abstract
 - define a method to be abstract by setting it = 0

44

Constructor chaining

- For a constructor, you can give the arguments to the constructors for:
 - the base classes
 - the instance variables
- If nothing provided, void constructor used
- E.g. class B : public A {
 C c1, c2; D d1;
 public: B() : A(42), c1("hello"), c2("hi"),
 d1() { }

45

What does B() print?

- struct A {
 A() { g(); };
 virtual void f() { cout << "A"; };
 virtual void g() { f(); }
}
- class B : public A {
 B() { g(); };
 virtual void f() { cout << "B"; };
}

46

What happened?

- Constructor B() chains to constructor for A()
 - while doing constructor for A(), object is an A
- Then, perform void constructor for B
 - now, object is a B

47

Namespaces

- What if I've defined a List class
- And you've written a different List class
- Can my code and your code be used in the same program?
 - No - Can't have two different classes named the same
 - Namespaces to the rescue
- Similar to packages in Java

48

Using namespaces

- namespace Foo {
 class List { ... };
};
- namespace Bar {
 class List { ... };
};
- Foo::List queue = new Foo::List();

49

namespace Foo { ... };

- All new definitions are in namespace Foo
- Make all definitions in Foo available
- OK to open and close a namespace:
 namespace A { ... };
 namespace B { ... };
 namespace A { ... };

50

namespace needed to add definitions

- Can use :: to give define meaning of a name previously defined
- But not to define a new name
- void Foo::f(int x) { ... };
 – Foo::f(int) must have been previously declared

51

using namespace

- using namespace Foo;
 – makes all names from Foo available
 – until end of file or namespace
- using Foo::List;
 – makes List from namespace Foo available as List

52

Inheriting namespaces

- namespace Foo {
 using A;
 using B;
 class List { ... };
};
- using Foo {
 // Foo::List and all names from A
 // and B available
}

53

namespaces aren't databases

- You see X in namespace Foo only if the compiler sees X declared in Foo at some point earlier in the file.
 – doesn't effect need to carefully link header files

54

namespace aliases

- What if two developers both write code using the namespace Parser?
 - back to our old problem of name clashes
- Partial solution using namespace aliases;
- namespace Parser = Pughs_Parser;
- Makes long names bearable
 - still have to come up with unique long names
 - Java style: edu_umd_cs_pugh_Parser ?

55

Exceptions

- throw *exp*; -- throws an exception *exp*
- try { ... } catch (ExceptionType e) { ... }
 - if, during executing of try block, an exception is thrown that could be passed to a function taking an ExceptionType as an argument, the catch clause is invoked with e bound to value thrown

56

Exception example

- struct IndexOutOfBounds {};
- char String::getChar(int i) {
 if (i < 0 || i >= length)
 throw IndexOutOfBounds();
};
- try {
 c = s.getChar(42);
} catch (IndexOutOfBounds err) {
 cout << "Index out of bounds";
}

57

Exception hierarchy

- You can create a hierarchy of exceptions
 - Exception
 - IOException
 - BoundsException
- Using derived classes
 - catch a reference to the base class
 - if a derived class is thrown, you catch it
 - catch value of the base class
 - if derived class is thrown, copy constructor for the base class is invoked, using the obj thrown as arg

58

More exceptions

- If you aren't going to do anything with the exception you caught
 - omit variable name
 - just name parameters
- You can throw other values
 - ints, doubles
 - don't do this...

59

Exception declarations

- You can declare the set of exceptions a function might throw
 - not checked at compile time
- Checked at run-time
 - if you declare the exceptions you throw
 - and you throw something not on that this
 - function unexpected() is called
 - which by default calls terminate()
 - which by default calls abort()

60

Declaring exceptions

- Declare you might throw X or Y
`int f(int i) throw (X,Y) { ... };`
- Declare you don't throw any exceptions
`int g(int i) throw () { ... };`
- Declare that you don't know what exceptions you throw:
`int h(int i) { ... };`

61

Using destructors for final actions

- Sometimes, you want to ensure you take some action when leaving a scope
 - no matter how you leave the scope
 - exception, return, fall off the bottom
- Destructors get called no matter how you leave the scope
 - use them

62

File closer

- ```
class FileCloser {
 const FILE *f;
public:
 FileCloser(FILE *f) { this.f = f; };
 virtual ~FileCloser() { f.close(); }
}
```

63

### Monitor lock

- ```
class MonitorLock {
    Monitor & m;
public:
    MonitorLock(Monitor & mon) : m(mon)
        { m.lock(); }
    virtual ~MonitorLock()
        { m.unlock(); }
}
```

64