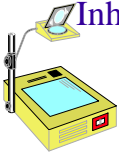




## Generalization/Specialization Inheritance & OO Design



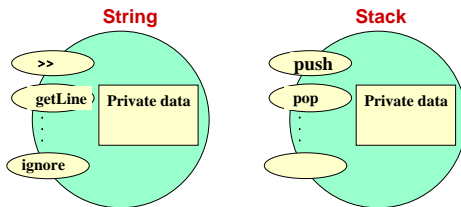
C++ & OO Programming

## What is OO?

- Inheritance is part of OO
- But OO is far more than just inheritance
- OO includes encapsulation & abstraction
- OO is how you organize your design
- Designing class hierarchies may be radically different
- inheritance can facilitate reuse, easy extension

2

## Encapsulation/Abstraction



3

## Inheritance issues

- public vs private inheritance
- virtual & non-virtual member functions
- purely virtual member functions
- abstract class
- must know:
  - how to use each feature
  - when to use

4

## Two rules

- public inheritance means: IS-A
- virtual function means: interface must be inherited

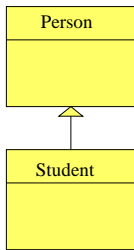
5

## Public Inheritance

- class D publicly inherits from class B
  - every object of type D is also an object of type B
  - but NOT vice-versa
  - B represents a more general concept than D
  - D represents a more specialized concept than B
  - anywhere a B can be used, a D can be used
  - However, if you need a D, a B will not work!

6

## public inheritance



Every student is a person, but not every person is a student.

C++ compiler enforces this!

7

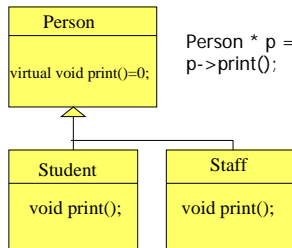
## is-a

```
class Person {
    void play(const Person &);
};
class Student : public Person {
    void study(const Student & s);
};
```

```
Person p;
Student s;
p.play();           // okay
s.play();           // no problem
s.study();          // yup!
p.study();          // error!
```

8

## Typical design



```
Person * p = new Student("Hank");
p->print();
```

9

## Passing parameters to base class constructor

```
class Person {
public:
    Person(const string & n) : name(n) {}
private:
    string name;
};

class Student {
public:
    Student(const string & n, float g) : Person(n), gpa(g) {}
private:
    float gpa;
};
```

10

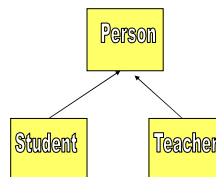
## Typical uses of inheritance

- Single inheritance – one base class
- attributes that describe all classes go in the base class
- attributes that are “special” go in derived class

11

## Software Reuse

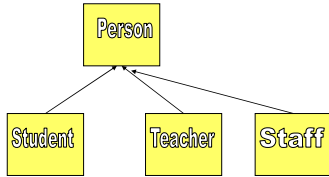
Derived classes reuse attributes in the base class.



12

## Easy to extend

Can add a new derived class, extending the framework, without changing any existing code: *extensibility*



13

## Invocation

- constructors
  - called bottom-up
  - execute top-down
- destructors
  - called top-down
  - execute bottom-up

14

## invocation versus execution?!

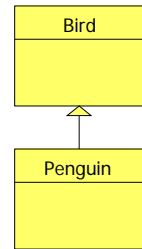
```
class Person {
public:
    Person(const string & n) : name(n) { cout << "base: construct\n"; }
    virtual ~Person() { cout << "base" << endl; }
private:
    string name;
};
class Student : public Person {
public:
    Student(const string & n, float g) : Person(n), gpa(g) {
        cout << "derived: construct" << endl; }
    }
    ~Student() { cout << "derived" << endl; }
private:
    float gpa;
};
Student x("Neo Anderson", 2.8);
```

15

## is-a doesn't always work right!

- fact: birds can fly
- fact: a penguin is a bird

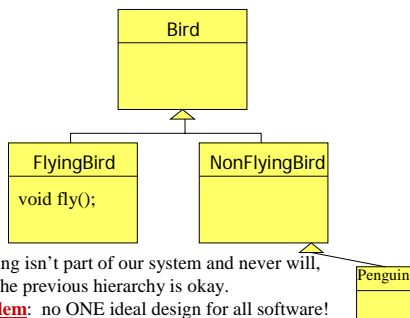
```
class Bird {
    virtual void fly();
};
class Penguin : public Bird {
};
```



**What's wrong with this?**

16

## One solution: Create a more "faithful" hierarchy



If flying isn't part of our system and never will, then the previous hierarchy is okay.  
**Problem:** no ONE ideal design for all software!

17

## Second Approach

**Redefine "fly" so that it generates a runtime error:**

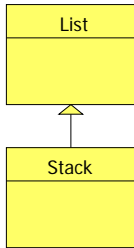
```
void error(const string & msg);

class Penguin : public Bird {
public:
    virtual void fly() { error("Penguins can't fly!"); }
};
```

**Better to detect errors at compile time, rather than at runtime.**

18

Another anti-intuitive example:  
a stack is-a list?

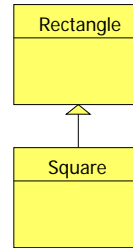


What's wrong with  
this picture?

compliments of  
Rumbaugh, et al.

19

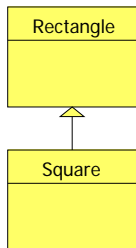
Another anti-intuitive example:  
Should Square inherit from Rectangle?



A square is-a rectangle?

20

Another anti-intuitive example:  
Should Square inherit from Rectangle?



```

class Rectangle {
public:
    virtual int getHeight();
    virtual void setHeight(int);
    virtual int getWidth();
    virtual void setWidth(int);
private:
    int height, width;
}
class Square : public Rectangle{};
    
```

**Problem:** something applicable to a rectangle,  
is not applicable to a square!

21

3 Kinds of functions in base class

```

class Shape {
public:
    virtual void draw() const = 0;
    virtual void error(const string &) const;
    int objectId() const;
};
class Rectangle : public Shape {};
class Circle : public Shape{};

Shape * s = new Shape; // error: Shape is abstract
Shape * r = new Rectangle; // okay
r->draw(); // calls draw in rectangle
    
```

22

virtual void draw() const = 0;  
a purely virtual function

- Derived classes inherit only the interface
- all derived classes must have a “draw”
- you can supply an implementation, but only way to call it is to specify the class:

```

class Shape {
public:
    virtual void draw() const = 0;
};
r->Shape::draw()
    
```

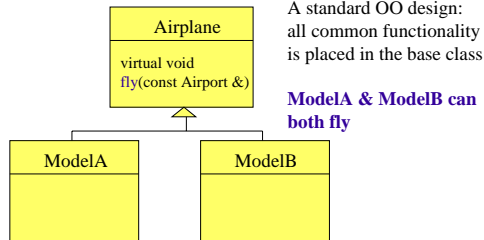
23

virtual void error(const string &);  
a simple virtual function

- derived classes inherit the interface and the implementation, which they may override
- In Shape example, every derived class gets **error()**, but may override
- a simple virtual function could be a problem if the designer later adds a derived class for which the inherited function doesn't apply!

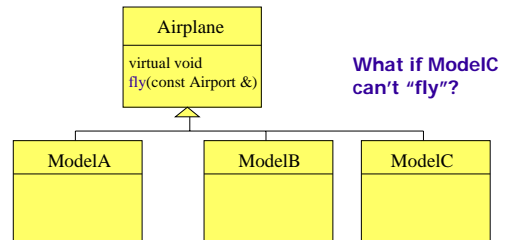
24

## simple virtual functions



25

## simple virtual functions



26

## int objectId() const; a non-virtual member function

- Since it's non-virtual, it's not supposed to behave differently in derived classes
- purpose: derived classes inherit an interface and a **mandatory** implementation
- Shape::objectId() means that every Shape object has a function that computes its id and it's always computed in the same way; it's **invariant over specialization**

27

## declaring functions in a base class

- the **three options**, pure virtual, simple virtual and non-virtual, allow the designer to specify meaning exactly
- the choice is a form of documentation

28

## Two mistakes

- declare all functions non-virtual in the base class
  - does not permit the derived class to specialize
  - it's okay to declare a class with all non-virtual functions, but just not as a base class
- declare all functions virtual
  - some base classes should have all virtual functions (e.g., the Protocol class)
  - however, some functions should not be overridden

29

## Why use virtual functions

- place the burden of knowing/choosing the object type on the compiler
- compiler constructs a vtbl for each class w/ virtual function
- one table per class; each class has ptr to vtbl
- vtbl holds pointers to all virtual functions
- vptrs init in constructor
- each virtual function is invoked thru its vptr

30

## Do virtual functions have performance penalty?

- vptr must be init in constructor
- virtual function is invoked via pointer indirection
- cannot inline virtual functions
- more later...

31

## advantage of virtual functions adapted from [Lip91]

```
class ZooAnimal {
public:
    virtual void draw() const = 0;
    int resolveType() const { return myType; }
private:
    enum animalTypes myType { BEAR, MONKEY };
};
class Bear : public ZooAnimal {
public:
    Bear(const string & name)
        : myName(name), myType(BEAR) {}
    void draw() { cout << "I'm a bear" << endl; }
};
```

32

## maintenance headache

```
void drawAllAnimals(ZooAnimal *pz) {
    for (ZooAnimal *p=pz; p; p = p->next) {
        switch (p->resolveType() ) {
            case BEAR : { ((Bear *)p)->draw(); break; }
            case MONKEY : { ((Monkey *)p)->draw(); break; }
            ... handle all other animals currently in the zoo
        }
    }
}
```

have to change the switch statement every time an animal arrives/leaves the zoo!

33

## Use virtual functions

```
void drawAllAnimals(ZooAnimal *pz) {
    for (ZooAnimal *p=pz; p; p = p->next) {
        p->draw();
    }
}
```

34

## RULE: Never redefine an inherited non-virtual function

```
class Person {
public:
    string getName() const { return name; }
private:
    string name;
};
class Student : public Person {};
```

```
Student stu;
Person * p = &stu;
Student * s = &stu;
p->getName();
s->getName();
```

**These two should behave the same!**

35

## Never redefine an inherited non-virtual function

```
class Person {
public:
    string getName() { return name; }
private:
    string name;
};
class Student : public Person { string getName(); };
```

```
Student stu;
Person * p = &stu;
Student * s = &stu;
p->getName();
s->getName();
```

**Now they behave differently!**  
Non-virtual functions are **statically** bound;  
virtual functions are **dynamically** bound

36

## non-virtual functions reflect: an invariant over specialization

- consider class D, derived from class B, with B::f() non-virtual
  - everything that is applicable to B objects, should be applicable to D objects (because every D isa B)
  - subclasses of B inherit the interface and the implementation of f()
  - if D redefines f(), there is a design contradiction!

37

## Polymorphism

Substitutability

38

## Polymorphism

- An easy concept, so often misunderstood
- in one word: **substitutability**
- a variable of type *base*, can be any object in the inheritance hierarchy!
- Thus, a variable of type *base* takes *many forms*!

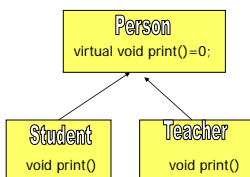
39

## virtual functions

- power polymorphism
- a virtual function is one that's called based on an object's type
- several related classes might declare a print, each different, but similar
- different sorts of objects know how to print themselves

40

## Polymorphism

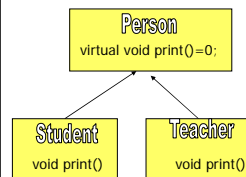


```
void print(Person * p) {
    p->print();
}
```

```
Person * p;
p = new Student(4.0);
print (p);
```

41

## a polymorphic list

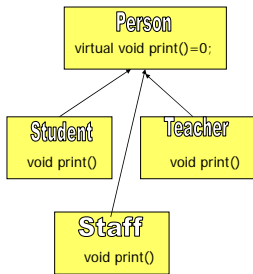


```
void print(const vector<Person*> p) {
    vector<Person*> ptr = p.begin();
    while (ptr != p.end()) {
        (*ptr)->print();
    }
}
```

```
vector<Person *> mylist;
mylist.push_back(new Student("Sam"));
mylist.push_back(new Teacher("Roy"));
mylist.push_back(new Student("Mary"));
print (mylist);
```

42

## Extend the list



```
void print(const vector<Person*> p) {
    vector<Person*> ptr = p.begin();
    while (ptr != p.end()) {
        (*ptr)->print();
    }
}

vector<Person *> mylist;
mylist.push_back(new Student("Sam"));
mylist.push_back(new Teacher("Roy"));
mylist.push_back(new Student("Mary"));
mylist.push_back(new Staff("Carol"));
print (mylist);
```

How much code gets changed?

43

## pure virtual function

- a placeholder that a base class uses
- requires that derived classes fill the hole
- creates an abstract base class (ABC)

44

## virtual destructors

- no objects are created from an ABC
- still have to provide for deletion of objects derived from ABC
- if destructor in base class is not virtual, the destructor in the derived class will not be called!

45

## memory leak?!

```
class Person {
public:
    Person(char * n) : name(new char[strlen(n)+1]) {
        strcpy(name, n);
    }
    ~Person() { delete [] name; }
private:
    char * name;
};

class Student : public Person {
public:
    Student(char * a) : address(new char[strlen(a)+1]) {
        strcpy(address, a);
    }
    ~Student() { delete [] address; }
private:
    char * address;
};
```

how to fill the leak?

46

## Virtual Streaming

How do you get overloaded output operator to work with inheritance

47

## Example of operator<<

```
class Student {
public:
    string & getName() const { return name; }
private:
    string name;
};

ostream & operator<<(ostream &, const Student &);
```

```
int main() {
    float x;
    Student stu;
    cout << x << endl;
    cout << stu << endl;
}
```

All output statements look the same!

```

class Person {
public:
    Person(const string & n) : name(n) {}
    const string & getName() const { return name; }
    virtual ostream & operator<<(ostream & out) { return (out << name); }
private:
    string name;
};
class Student : public Person {
public:
    Student(const string & n, float g) : Person(n), gpa(g) {}
    virtual ostream & operator<<(ostream & out) {
        return (out << getName() << "\t" << gpa);
    }
private:
    float gpa;
};
int main() {
    Student * ptr = new Student("Anakin", 1.5);
    (*ptr) << cout;
}

```

**This works!**

**Lack of symmetry here!**

```

class X {
public:
    virtual ostream & print( ostream & out ) const {
        // code to output an X
        return out;
    }
};
ostream & operator<<(ostream & out, const X & x) {
    return x.print(out);
}

class Y : public X {
    virtual ostream & print( ostream & out ) const {
        // code to output a Y
        return out;
    }
};

```

**Just one of these!**

**This is a pattern for virtual streaming**

50

```

class Person {
public:
    Person(const string & n) : name(n) {}
    virtual ~Person() {}
    const string & getName() const { return name; }
    virtual ostream & print(ostream & out) const = 0;
private:
    string name;
};
ostream & operator<<(ostream & out, const Person & p) { return p.print(out); }
class Student : public Person {
public:
    Student(const string & n, float g) : Person(n), gpa(g) {}
    virtual ostream & print(ostream & out) const { return out << getName() << '\t' << gpa; }
private:
    float gpa;
};
int main() {
    Person * stu = new Student("Anakin", 2.5);
    cout << *stu << endl;
}

```

**Symmetric with normal output**

51

## Avoid casting down the inheritance hierarchy

But first:  
C++ style casts

52

## Prefer C++ style casts

(type) expression

type(expression)

```

static_cast<type>(expression)
const_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)

```

53

## The C cast

- casts are considered, by some, as much of a pariah as a **goto!**
- C style casts permit the programmer to cast from almost any type to almost any type -- with unpredictable results
- C style casts are hard to find in a program (try finding them with grep!)  
(int) myvariable;

54

## C++ casts attempt to address the shortcomings of C-style casts

```
(type) expression;  
static_cast<type>(expression);
```

The C++ version is easier to find.  
both versions have same power and restrictions:  
-- can't cast away const  
-- can't cast an int into a struct  
-- can't cast a double into a pointer

```
int i, j;  
double result = ((double) i) / j;  
double result = static_cast<double>(i)/j;
```

55

## const\_cast<type>(expression)

Used to cast const-ness away!

```
class Student {  
public:  
    void updateGpa();  
};
```

```
const Student student_council_pres;  
update(const_cast<Student>(student_council_pres));
```

56

## casting down the inheritance hierarchy

dynamic\_cast<type>(expression)

57

## consider the following from Meyers

```
class Person{};
```

Protocol Class

```
class BankAccount {  
public:  
    virtual BankAccount(const Person * primaryOwner,  
        const Person * jointOwner);  
    virtual ~BankAccount();  
    virtual void makeDeposit(double) = 0;  
    virtual void makeWithdrawal(double) = 0;  
    virtual double balance() const = 0;  
};
```

58

## Motivating example

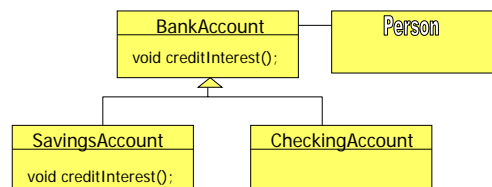
```
class Person{ };  
class BankAccount { };  
class SavingsAccount : public BankAccount ;  
class CheckingAccount : public BankAccount;
```

Let's assume that SavingsAccounts give interest,  
but CheckingAccounts do not!

Question: How to you credit interest to SavingsAccounts,  
but not to CheckingAccounts?

59

## Now suppose the checking account doesn't bear interest

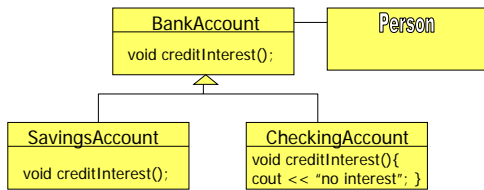


An non-interest bearing checking account!

What's the problem here?

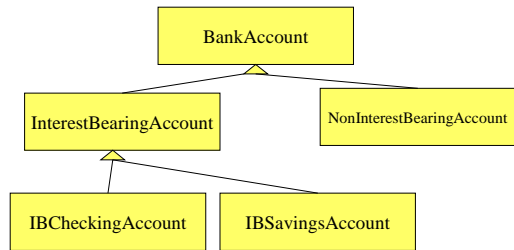
60

## One solution: put a dummy function in derived class:



61

## Extending the hierarchy



Problem with this one: you may need two lists!

62

## There are occasions when you must downcast

- Suppose the situation is beyond your control:
  - using a library to which you only have read access
  - you inherit a class hierarchy and you must modify it
  - `Dynamic_casting` is a better approach than raw-casting

63

## If you decide to downcast using dynamic cast, how does it work?

- when used on a pointer
  - if cast succeeds, i.e., the dynamic type of the pointer is same as cast type, a valid pointer of the expected type is returned
  - if the cast fails, the null pointer is returned

64

## dynamic\_cast example:

```

class BankAccount {};
class SavingsAccount : public BankAccount {
public: void creditInterest(); };
class CheckingAccount : public BankAccount {
public: void creditInterest(); };
...

for (list<BankAccount>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
    if (SavingsAccount * s = dynamic_cast<SavingsAccount*>(*p)) {
        s->creditInterest();
    }
}

```

65

## The cost of Virtual Functions

Scott Meyers  
*More Effective C++*  
 35 New Ways to Improve  
 your Programs & Designs

66

## Cost of virtual function calls

- when a virtual function is called
  - the executed code must correspond to the dynamic type of the object
  - the type of the pointer variable is immaterial
- To do this:
  - virtual tables (vtables)
  - virtual pointers (vptrs)

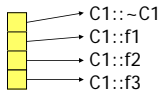
67

## vtbl

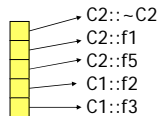
- usually an array of pointers to functions
- could be a linked list (in some compilers)
- each class has a vtbl
- difficult issue: where to put the vtbl so that all objects of the class can access

68

```
class C1 {
public:
    C1;
    virtual ~C1();
    virtual void f1();
    virtual int f2(char c) const;
    virtual void f3();
    void f4() const;
};
```



```
class C2 : public C1 {
    C2();
    virtual ~C2();
    virtual void f1();
    virtual void f5(char c);
};
```



69

## Need more

- vtbl by itself doesn't specify which function
- need virtual table pointer vptr to indicate which vtbl
- a hidden data member of each object
- init in constructor

70

## vptr

Data members  
for the object

object's vptr

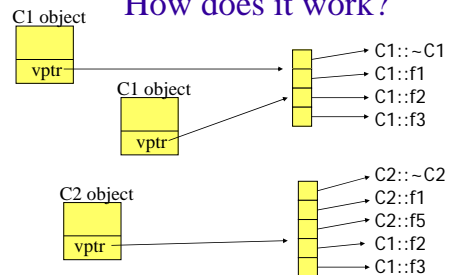
So far, cost of virtual functions:

- (1) one vtbl for each class
- (2) one vptr for each object

if class is small, vptr  
could double the size!

71

## How does it work?



```
void make_call(C1 * p) {
    p->f1();
}
```

72

## compiler generates code to:

- follow object's vptr to its vtbl
- find pointer in vtbl for this function (f1 in the example)
- invoke the function

`p->f1()` becomes

```
(*p->vptr[I])(); // call function pointed to by I-th  
                // entry in vtbl pointed to by p->vptr
```

73

## now the cost of virtual call is

- one vtbl for each class
- one vptr for each object
- calling a function through a pointer
- real cost is that they can't be inlined!

74

## reconsider the performance penalty

- cost of setting vptr in constructor is equivalent to initializing the type member
- cost of indirect function call is equivalent to the switch statement logic
- inability to inline a virtual function is its only penalty [Myers96, item 24]

For a case study, see:  
[www.brianmalloy.com/courses/cpp/examples/july19/](http://www.brianmalloy.com/courses/cpp/examples/july19/)

75

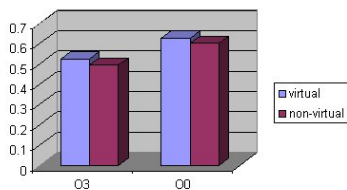
## timings for virtual functions

- Perform timings using a python script
- run it for 100 iterations
- Dell Precision workstation with 1.7 GHz Xeon processor with 500 M of Rambus, running RedHat Linux 7.1.
- Used gcc 2.96 to compile, with `-O2` optimizations, loop unrolling and inlining

76

## tale of the tape

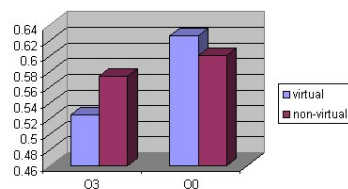
running time with inlining for non-virtual functions



77

## Using `-fno-inline`

running time with no inlining for non-virtual functions



78

## Private Inheritance

“Use private inheritance judiciously”,  
Item 42 in Meyers *Effective C++*

79

## Public Inheritance models IS-A

```
class Person { };  
class Student : public Person {};
```

```
void dance(const Person & p);  
void study(const Student & s);
```

```
Person p;  
Student s;  
dance(p);  
dance(s);  
study(s);  
study(p);
```

Which one won't compile?

80

## What does private inheritance model?

```
class Person { };  
class Student : private Person {};
```

```
void dance(const Person & p);  
void study(const Student & s);
```

```
Person p;  
Student s;  
dance(p);  
dance(s); // oops!
```

private inheritance does not mean IS-A

81

## Private inheritance means: *Implemented-in-terms-of*

- compilers will not convert a derived class object into a base class object
- members inherited from a derived class become private in the base class: even if they are public or protected in the base
- thus: you can make an implementation protected in the base and then inherit it in the derived class and users cannot get at the inherited implementation

82

## A motivating example for private inheritance

- templates are one of the most useful features of C++
- Consider a template stack that gets instantiated a dozen times, once for ints, once for floats, once for Students, etc
- you have a dozen instances of stack, each with its own copy of the code
- this is called **template induced code bloat!**

83

## Alternative: generic stack proposed by S. Meyers, 2<sup>nd</sup> edition

- Let's create our own generic stack using void\*
- we'll create the single generic stack, and then provide interfaces for each of the types of stacks

84

```

class GenericStack {
public:
  GenericStack() : topPtr(0) {}
  ~GenericStack(){ /* later */ }
  inline void push(void *object) {
    topPtr = new StackNode(object, topPtr); }
  inline void pop();
  inline void* top() const { return topPtr->data; }
  inline bool empty() const { return topPtr == 0; }
private:
  struct StackNode {
    StackNode(void *newData, StackNode *nextNode)
      : data(newData), next(nextNode) {}
    void* data;
    StackNode *next;
  };
  StackNode *topPtr;
};

```

Here's the generic stack!

85

Now we provide an interface for ints

```

class IntStack {
public:
  void push(int * intPtr) { s.push(intPtr); }
  int * pop() { return static_cast<int*>(s.pop()); }
  bool empty() const { return s.empty(); }
private:
  GenericStack s;
};

```

only int pointers can be pushed onto an IntStack!

However, users could just create an instance of GenericStack and get around the "type safety"!

86

Use private inheritance to preclude users from instantiating GenericStack!

```

class GenericStack {
protected:
  GenericStack();
  ~GenericStack();
  inline void push(void * object);
  inline void * pop();
  inline bool empty() const;
private:
  etc.
};
class IntStack : private GenericStack { };

GenericStack s; // error!

```

87

Don't want to have to type IntStack code to get an interface for each new type

```

template <class T>
class IntStack : private GenericStack {
public:
  void push(T * intPtr) { s.push(intPtr); }
  T * pop() { return static_cast<T*>(s.pop()); }
  bool empty() const { return s.empty(); }
private:
  GenericStack s;
};

```

compilers will generate as many interface classes as the users needs!

88

beauty of this approach

- because the interface class provides type safety, user type errors will be detected statically
- clients are unable to bypass the interface class
- interface functions are inline – no runtime cost; the generated code is the same as if programmers used GenericStack directly
- you pay for only one copy of the code for GenericStack
- maximally efficient, maximally type safe

89