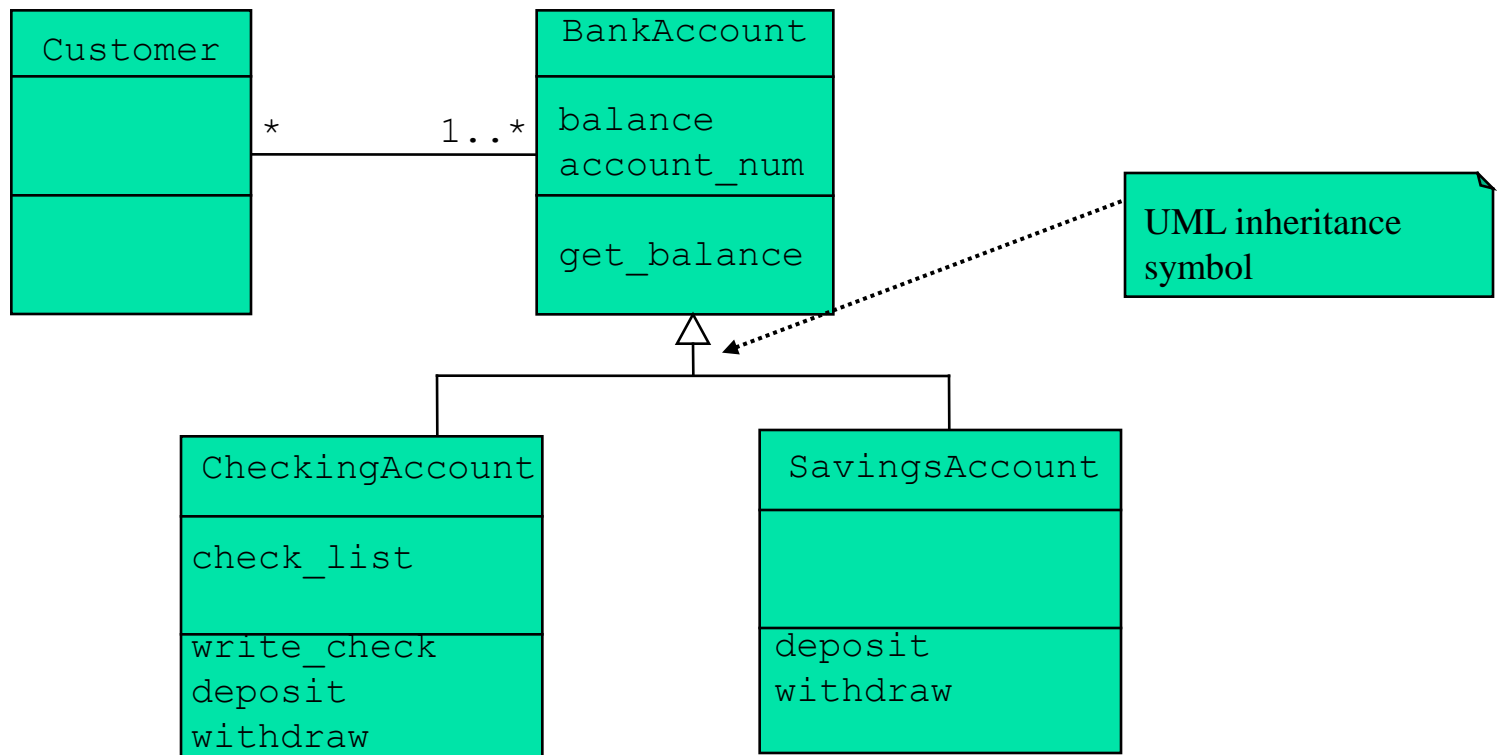


Inheritance

Inheritance in UML

- Inheritance models the object relation “is a kind of”
 - A SavingsAccount is a kind of BankAccount
 - Thus any attribute or method of a BankAccount also applies to a SavingsAccount



Inheritance in C++

Money would be typedef'd in a header file as a decimal type

```
// File: bankaccount.h
class BankAccount {
private:
    Money balance;
    int account_num;
public:
    Money get_balance() {
        return balance;
    }
    // other members,
    // e.g., constructor
};
```

public modifier needed to allow users of the *derived* class to access members of the *base* class

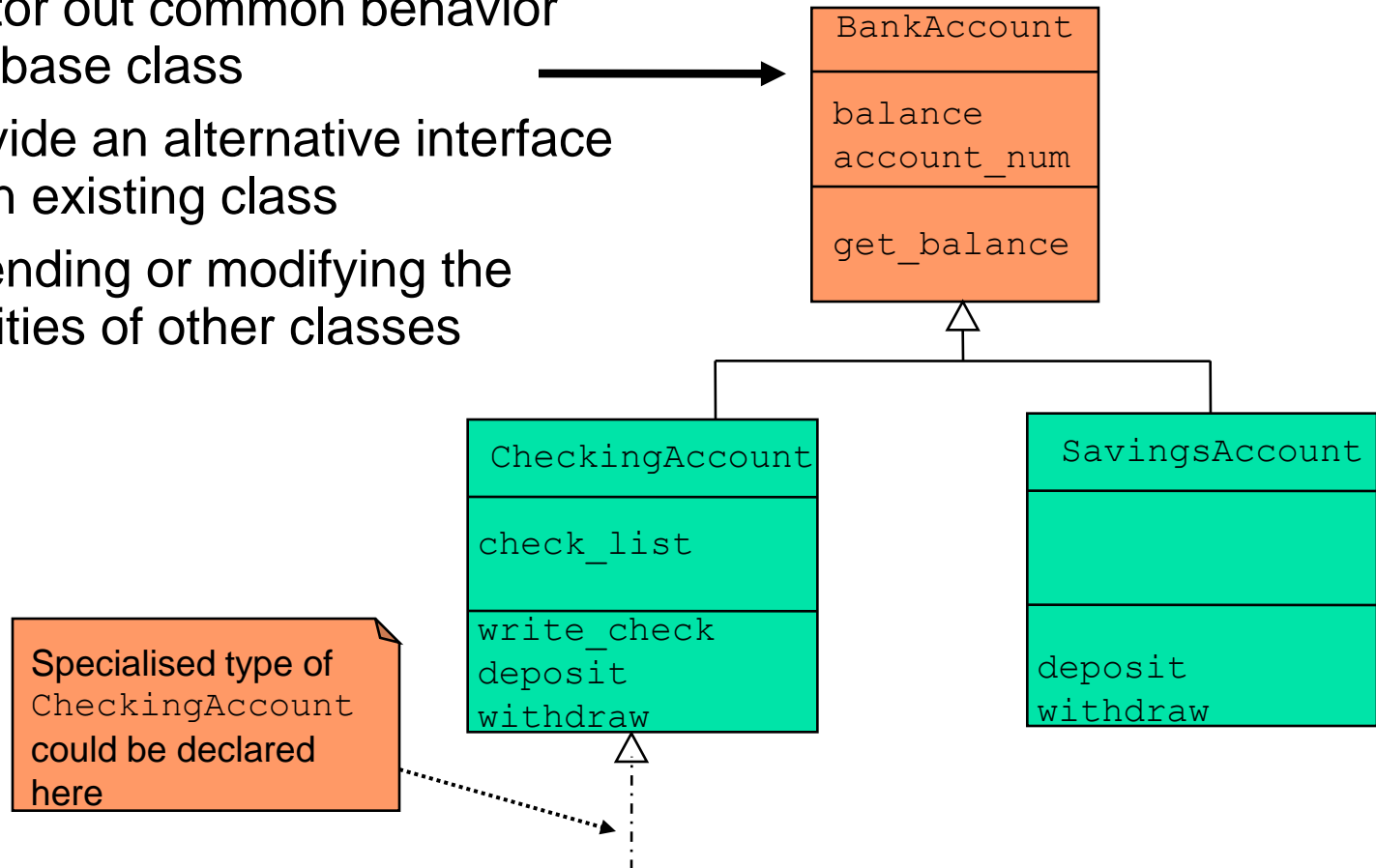
#include of file containing the base class definition needed. Double quotes are used as this is a user file, not a system file

```
#include "bankaccount.h"
...
class CheckingAccount : public BankAccount {
private:
    std::list<Check*> check_list;
public:
    void deposit(Money m) { ... }
    void withdraw(Money m) { ... }
    void write_check(Check* cp) { ... }
    // other members, e.g., constructor
};
```

Bankaccount.cpp

Derived Classes

- Derived classes are central to reusable software development; they provide the capability to
 - Factor out common behavior in a base class
 - Provide an alternative interface to an existing class
 - Extending or modifying the facilities of other classes

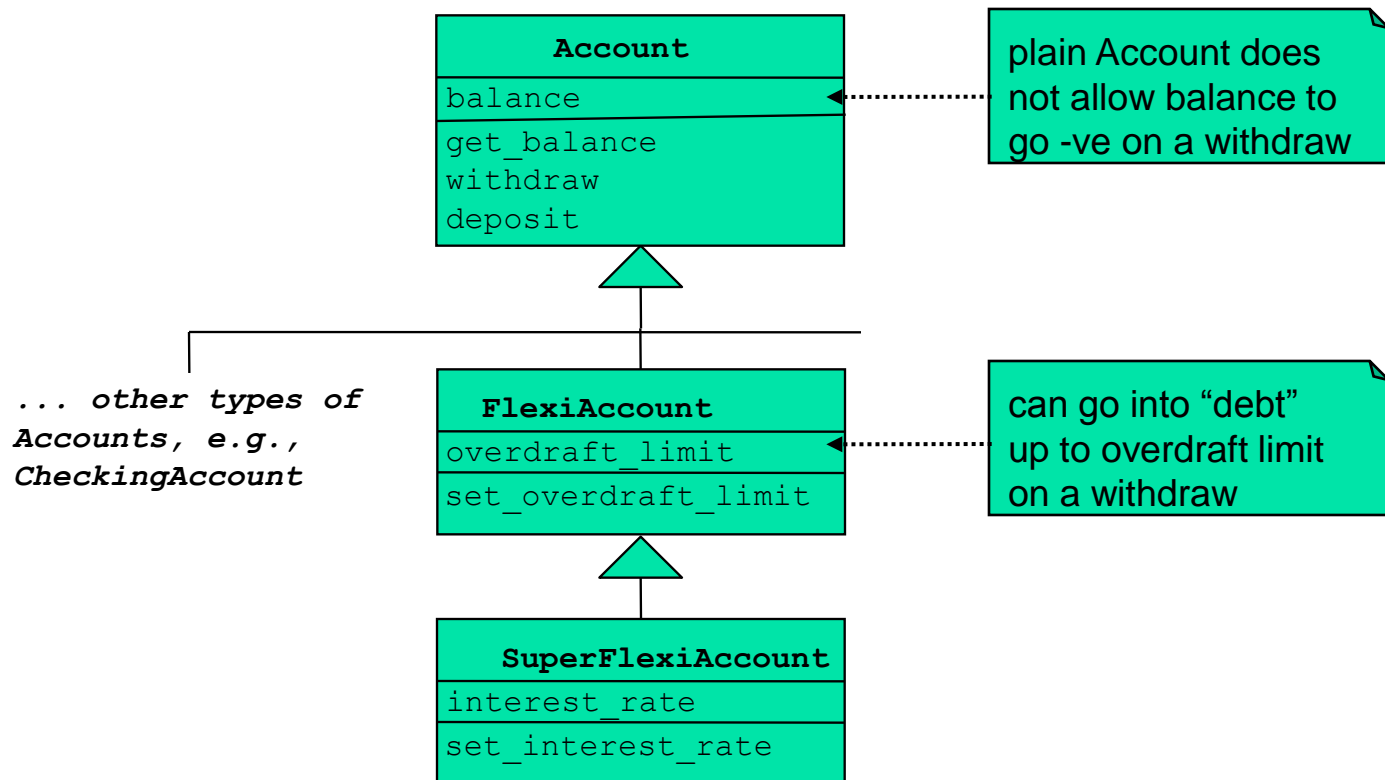


Inheritance Hierarchies

- Suppose that we were building a banking application, with various types of accounts and transactions
 - In our business analysis, for a particular bank, we will assume we have identified 3 types of account:
 - a plain savings Account
 - Supports deposit and withdraw, but no overdrafts or interest
 - Flexi Account
 - Supports deposit, withdraw, and overdrafts to a set limit (you can borrow money from the bank), but no interest
 - SuperFlexi Account
 - Supports deposit, withdraw, overdrafts, and credits interest for large balances
 - We will restrict our example to just these simple classes and properties
- How might these be related by inheritance?
- The “is a kind of” test can be used to determine if inheritance is appropriate

Inheritance Hierarchies (cont.)

- We might implement a skeleton of the hierarchy as follows



Inheritance Hierarchies (cont.)

```
// File "account1.h"
class Account
{
    int balance;
public:
    Account() : balance(0)
    int get_balance()
        { return balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

withdraw()
and
deposit() not
overridden
yet by
FlexiAccount

```
// File "flexiaccount1.h"
#include "account1.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
public:
    FlexiAccount()
        : overdraft_limit(0) {}
    bool set_overdraft_limit(int limit);
};
```

```
#include "flexiaccount1.h"
#include <iostream>
using namespace std;
int main() //main1.cpp
{
    Account a;
    cout << a.get_balance();

    FlexiAccount fa;
    fa.deposit(10);
    fa.withdraw(5);
    cout << fa.get_balance();

    fa.set_overdraft_limit(20);
    fa.withdraw(10);
    cout << fa.get_balance();
    ...
};
```

main() output
will be:
0 5 5 !!

Inheritance Hierarchies (cont.)

```
#include "account1.h"
bool Account::withdraw(int amount)
{
    if (amount <= 0 || amount > balance)
        return false;
    balance -= amount;
    return true;
}
bool Account::deposit(int amount)
{
    if (amount <= 0)
        return false;
    balance += amount;
    return true;
}
```

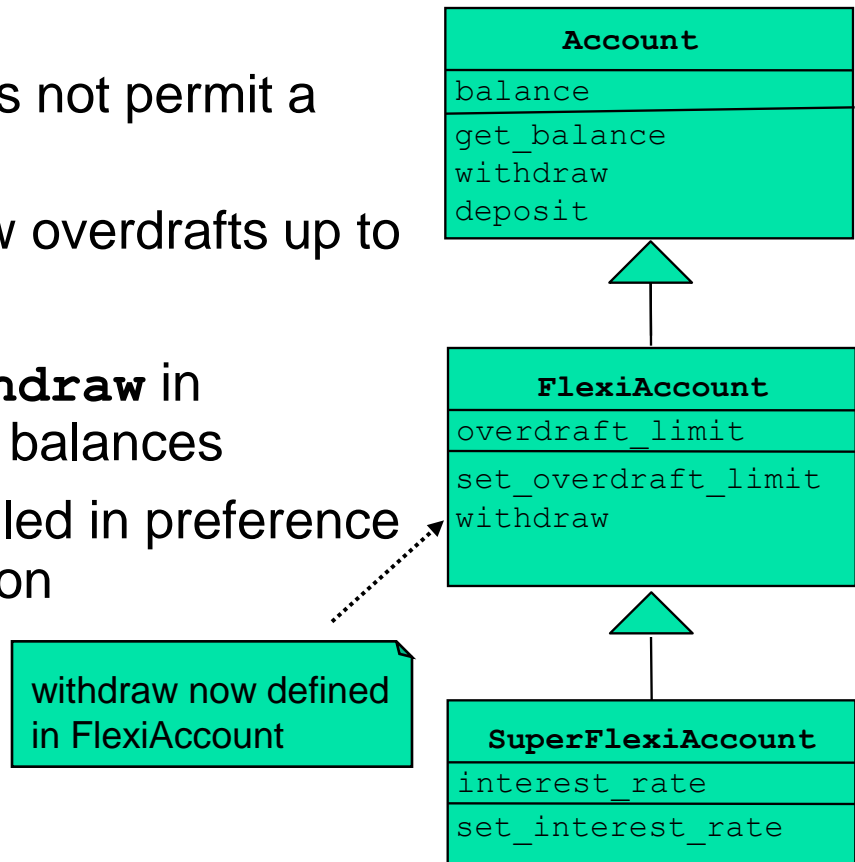
File account1.cpp:
implementation of
withdraw and deposit.
withdraw does not allow
-ve balance. Both
withdraw and deposit
return false if the
operation failed

```
#include "flexiaccount1.h"
bool FlexiAccount::set_overdraft_limit(int limit)
{
    if (limit < 0) return false;
    overdraft_limit = limit;
    return true;
}
```

File flexiaccount1.cpp :
implementation of
FlexiAccount
set_overdraft_limit
method

Inheritance Hierarchies (cont.)

- At present, the earlier `main()` program does not work properly, because **FlexiAccount** inherits **Account::withdraw()**
 - `Account::withdraw()` does not permit a negative balance
 - **FlexiAccount** wants to allow overdrafts up to `overdraft_limit`
- A simple solution is to define **withdraw** in **FlexiAccount** to allow negative balances
 - A class member function is called in preference to a base-class member function



Inheritance Hierarchies (cont.)

```
// File "flexiaccount2.h"
#include "account1.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
public:
    FlexiAccount() : overdraft_limit(0) { }
    bool set_overdraft_limit(int limit);
    bool withdraw(int amount);
};
```

Override base
member function
Allow withdraw of
amount up to a
balance of
- overdraft_limit

```
#include "flexiaccount2.h"
bool FlexiAccount::withdraw(int amount)
{
    if (amount <= 0 || amount > balance + overdraft_limit)
        return false;
    balance -= amount;
    return true;
}
```



What access errors occur in the member function? How can we fix this?
*Consider the access permissions of **Account::balance***

Inheritance Hierarchies (cont.)

- **balance** is private so **FlexiAccount::withdraw** cannot use it
- A quick fix is to make **balance** public, but this violates encapsulation
 - A better approach is public **get** and **set** methods

```
// File "account2.h"
class Account
{
    int balance;
public:
    Account() : balance(0) { }
    int get_balance() { return balance; }
    void set_balance(int new_balance) { balance = new_balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

balance private

two new member functions

```
#include "flexiaccount2.h"
bool FlexiAccount::withdraw(int amount)
{
    if (amount <= 0 || amount > get_balance() + overdraft_limit)
        return false;
    set_balance(get_balance() - amount);
    return true;
}
```

now use get and set methods

Inheritance Hierarchies (cont.)

- At first glance, it might seem that **get** and **set** methods are no different to a public data member
 - Any read or write can be implemented using **get** and **set**
- The key difference is that **get** and **set** encapsulate all updates and access
 - For example, if we needed to check for overflow of an account balance or check for high balances, the only change needed is to the **set** member

```
// File "account2.h"
class Account
{
    int balance;
public:
    Account() : balance(0) { }
    int get_balance() { return balance; }
    void set_balance(int new_balance) { balance = new_balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

put balance
checks here

deposit and withdraw
should call set_balance

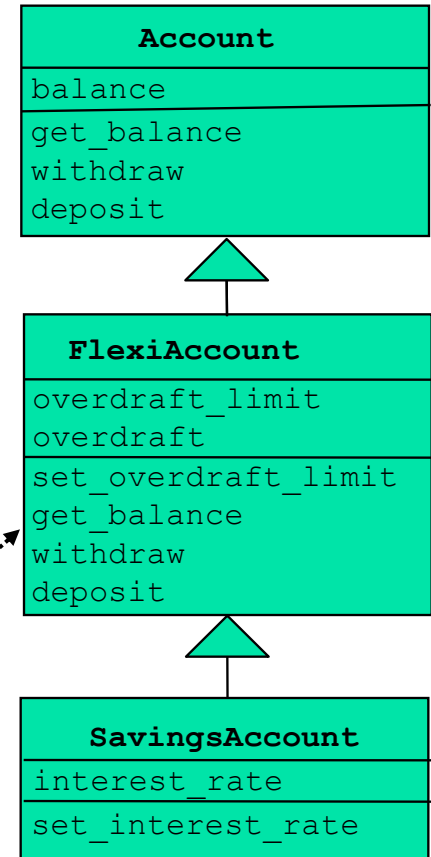
Inheritance Hierarchies (cont.)

- This approach still is unsatisfactory from a design and maintenance viewpoint
 - Anyone can change `Account::balance` via a `set_balance` call
The base class did not explicitly say that it was willing to let derived classes override the `withdraw` method
- A better approach is to maintain a separate **overdraft** balance in **FlexiAccount**
 - A `FlexiAccount` balance is the base `Account::balance` minus any overdraft amount
 - `Account::set_balance` is no longer needed
 - `FlexiAccount` now needs to redefine `deposit`, `withdraw`, and `get_balance`

Inheritance Hierarchies (cont.)

- **FlexiAccount::withdraw()** will take money from **Account::balance** by calling **Account::withdraw()** and/or add money from **FlexiAccount::overdraft**, up to **FlexiAccount::overdraft_limit**
- Similarly, **FlexiAccount::deposit()** will put money into **Account::balance** by calling **Account::deposit()** after paying off any **FlexiAccount::overdraft**

3 more member functions defined in FlexiAccount



Inheritance Hierarchies (cont.)

```
// File "flexiaccount3.h"
#include "account1.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
    int overdraft;
public:
    FlexiAccount() : overdraft_limit(0), overdraft(0) { }
    bool set_overdraft_limit(int limit);
    bool withdraw(int amount);
    bool deposit(int amount);
    int get_balance();
};
```

new data
member

override 3
base member
functions

Inheritance Hierarchies (cont.)

```
#include "flexiaccount3.h"
int FlexiAccount::get_balance()
{
    return Account::get_balance() - overdraft;
}
bool FlexiAccount::withdraw(int amount)
{
    if (amount < 0 || amount > (get_balance() + overdraft_limit))
        return false;
    int from_account = min(Account::get_balance(), amount);
    Account::withdraw(from_account);
    overdraft += amount - from_account;
    return true;
}
bool FlexiAccount::deposit(int amount)
{
    if (amount < 0) return false;
    int to_overdraft = min(amount, overdraft);
    Account::deposit(amount - to_overdraft);
    overdraft -= to_overdraft;
    return true;
}
```

call base member
function using its
qualified name

Withdraw amount
from either the
base Account
balance or
overdraft or both

Deposit adds to overdraft,
balance or both

Derived Class Constructors

- A derived class instance "is a" base class instance
 - So a base class default constructor is called automatically

```
// File "account3.h"
class Account
{
    int balance;
public:
    Account()
        : balance(0) { }
    int get_balance()
        { return balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

```
// File "flexiaccount3.h"
#include "account3.h"
class FlexiAccount : public
Account
{
    int overdraft_limit;
public:
    FlexiAccount()
        :
        overdraft_limit(0) { }
    ...
};
```

Implicit call
to base constructor
here.
We can have an
explicit call
for readability

Derived Class Constructors (cont.)

? What if the base class has no default constructor?

Is this
constructor
definition
legal?

```
// File "account4.h"
class Account
{
    int balance;
public:
    Account(int init_balance)
        : balance(init_balance) {}
    int get_balance()
        { return balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

```
// File "flexiaccount4.h"
#include "account4.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
public:
    FlexiAccount()
        : overdraft_limit(0) {}
    ...
};
```

Derived Class Constructors (cont.)

- If the base class has no default constructor
 - Then the derived class must call the constructor in the member initializer list

```
// File "account4.h"
class Account
{
    int balance;
public:
    Account(int init_balance)
        : balance(init_balance) { }
    int get_balance()
        { return balance; }
    bool withdraw(int amount);
    bool deposit(int amount);
};
```

Base class constructor explicitly called here, member initializations must be comma separated

```
// File "flexiaccount5.h"
#include "account4.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
public:
    FlexiAccount(int init_balance)
        : Account(init_balance),
          overdraft_limit(0) { }
    ...
};
```

Derived Class Constructors (cont.)

- C++ has a strict rule of initialization in "declaration order"
 - Base classes are initialized before any data members
 - And so base-base classes before base, recursively
 - Then data members, in the order they are declared in the class
 - NOT the member initializer order

```
// File "foo.h"
class Foo
{
    int x;
public:
    Foo(int init_x)
        : x(init_x) { }
    ...
};
```

Foo initialized first, then y, then z!
So y is undefined!
Never initialize members with other
members in a member initialization list

```
// File "bar.h"
#include "account4.h"
class Bar : public Foo
{
    int y;
    int z;
public:
    Bar(int init_x)
        : z(10),
          y(z),
          Foo(init_x)
    {}
    ...
};
```


Base-class member-access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	<p>public in derived class.</p> <p>Can be accessed directly by member functions, friend functions and nonmember functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
protected	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
private	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>

Fig. 12.16 | Summary of base-class member accessibility in a derived class.

Polymorphism

Type Compatibility

- A derived-class “is a kind of” base-class, but not conversely
 - So a derived-class instance can be assigned to a base-class instance, but not conversely
 - Also, base-class pointer can point to a derived-class instance, but not conversely

```
#include "superflexiaccount1.h"
int main()
{
    Account a;
    FlexiAccount fa;
    SuperFlexiAccount sa;
    a = fa;
    Account* aptr;
    aptr = &sa;

    fa = a;
    SuperFlexiAccount* sptr;
    sptr = &a;
    return 0;
}
```

This include also includes
flexiaccount.h and account.h

“member-wise copy”:
balance=oa.balance

Now *aptr* points to *sa*

Two compile errors

Type Compatibility (cont.)

- Base-class pointers are very useful, because they potentially allow us to access instances of any derived class
 - At runtime, we can choose which derived class to access

```
#include "superflexiaccount3.h"
#include <iostream>
using namespace std;
int main()
{
    Account* aptr;
    FlexiAccount fa;
    SuperFlexiAccount sa;
    ...
    aptr = &fa;
    ...
    aptr->deposit(50);
    cout << aptr->get_balance();
    return 0;
};
```

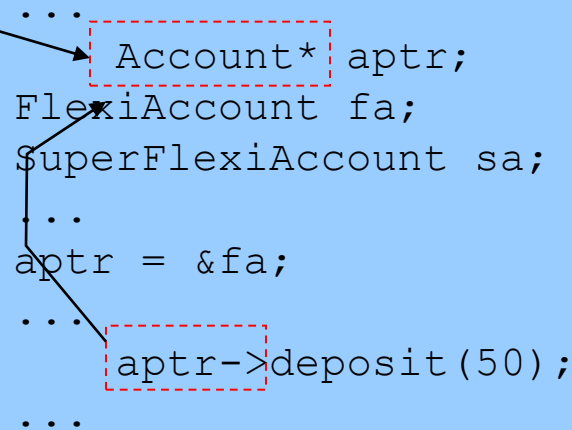
This include also includes flexiaccount.h and account.h

Or `aptr = &sa`, we still get calls of base class members

Call `Account::deposit()`, then `Account::get_balance()`, output: 50

Dynamic Binding

- In C++, binding is static by default, for run time efficiency
 - The type of the pointer (`aptr`) determines the function called
 - This is determined at compile time



The diagram illustrates static binding in C++. It shows a code snippet within a light blue box. The code is as follows:

```
...  
Account* aptr;  
FlexiAccount fa;  
SuperFlexiAccount sa;  
...  
aptr = &fa;  
...  
aptr->deposit(50);  
...
```

Annotations in the diagram include:

- A red dashed box around `Account*` in the first line.
- A red dashed box around `aptr->` in the last line.
- An arrow pointing from the text "The type of the pointer (`aptr`)" in the list above to the `Account*` box.
- An arrow pointing from the text "This is determined at compile time" in the list above to the `aptr = &fa;` line.

- In Java, binding is dynamic
 - The type of the object pointed to (`aptr->`) determines the function called
 - This is determined at run time
- To get dynamic binding in C++, functions must be declared as `virtual`

Dynamic Binding (cont.)

base class declares 3 functions virtual for dynamic binding

```
// File "account5.h"
class Account
{
    int balance;
public:
    Account() : balance(0) { }
    virtual int get_balance()
        { return balance; }
    virtual bool withdraw(int amount);
    virtual bool deposit(int amount);
};
```

```
// File "flexiaccount5.h"
#include "account5.h"
class FlexiAccount : public Account
{
    int overdraft_limit;
    int overdraft;
public:
    FlexiAccount() : overdraft_limit(0), overdraft(0) {}
    bool set_overdraft_limit(int limit);
    bool withdraw(int amount);
    bool deposit(int amount);
    int get_balance();
}
```

not virtual

3 derived classes declarations inherit **virtual** for dynamic binding; **virtual** keyword optional on their declarations

Dynamic Binding (cont.)

Includes base
FlexiAccount
and base
Account

Calls FlexiAccount::withdraw
as Account::withdraw is virtual
Output will be: -5

Calls FlexiAccount::withdraw
unless SuperFlexiAccount::withdraw
is defined
Output will be: -15

```
#include "superflexiaccount5.h"
#include <iostream>
using namespace std;
int main()
{
    Account* aptr;
    FlexiAccount fa(10);
    SuperFlexiAccount sfa(30);
    ...
    aptr = & fa;
    fa.set_overdraft_limit(10);
    aptr->withdraw(15);
    cout << aptr->get_balance();
    ...
    aptr = & sfa;
    sfa.set_overdraft_limit(20);
    aptr->withdraw(45);
    cout << aptr->get_balance();
    ...
};
```

Dynamic Binding (cont.)

- Derived classes inherit the virtual function definitions of their base class
- Thus, **SuperFlexiAccount** inherits **FlexiAccount**'s definition of **withdraw()**, but it can override this function if needed
 - A derived class always uses the “closest” **virtual** function definition in its parent hierarchy

	Overloaded functions	Virtual functions
Must have <i>same</i> name?	Yes	Yes
<i>Different</i> signatures?	Always	Never
Binding	Compile-time	Runtime
Scope	Same	Inherited

Dynamic Binding (cont.)

- Virtual functions are a powerful mechanism for writing adaptable applications

transfer works with “any” Account type and dynamically calls the right versions of deposit and withdraw

```
bool transfer(Account* from, Account* to, int amount)
{
    // call virtual functions withdraw and deposit
    if (!from->withdraw(amount)) return false;
    return to->deposit(amount);
}
```

- For example:

```
int main()
{
    Account a1(200);
    FlexiAccount fa(300);
    fa.set_overdraft_limit(500);
    transfer(&fa, &a1, 400);
    // call FlexiAccount::withdraw and Account::deposit
    return 0;
}
```

Interfaces

- Base classes are often defined with no intention of ever creating instances of them
 - **class** Account
 - Must specify what type of Account: Checking, Savings, ...
 - **class** GraphicObject
 - No object is “just” a GraphicObject, is it a Square or Circle, or ...
- Such base classes are very useful abstractions of common operations, which are `virtual` functions declared in the base class and defined by derived classes
 - `print()`, ...
 - `draw()`, `rotate()`, `move()`, `remove()`
- Operations in such base classes often have no implementation
 - `GraphicObject::draw()` cannot be defined without the derived type

Interfaces (cont.)

- C++ provides a better approach, *pure virtual* member functions
 - Just like Java **abstract** functions

```
class Account
{
    // data common to all Accounts
public:
    virtual void print() = 0; // pure virtual
    // = 0 means "no definition necessary"
    // other methods
};
```

- If any of a class's virtual functions are *pure virtual*, the class is an *abstract class* and instances of it cannot be created

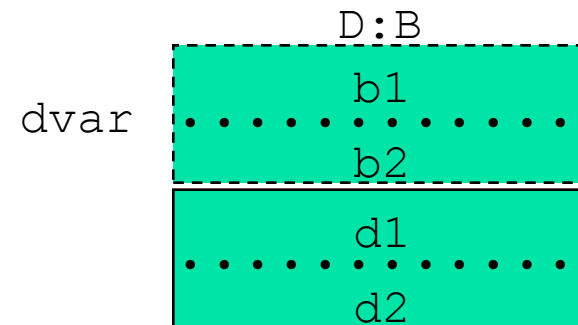
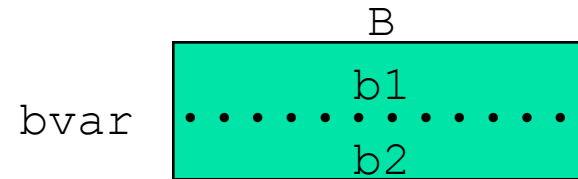
```
Account a;    // syntax error
Account* ap;  // pointers to abstract classes are legal
```

- Derived classes must supply definitions of all pure virtual functions; otherwise, they are also abstract classes

Runtime Type Identification

- By default, classes are just like structs
 - Containers of data, with no run time type information

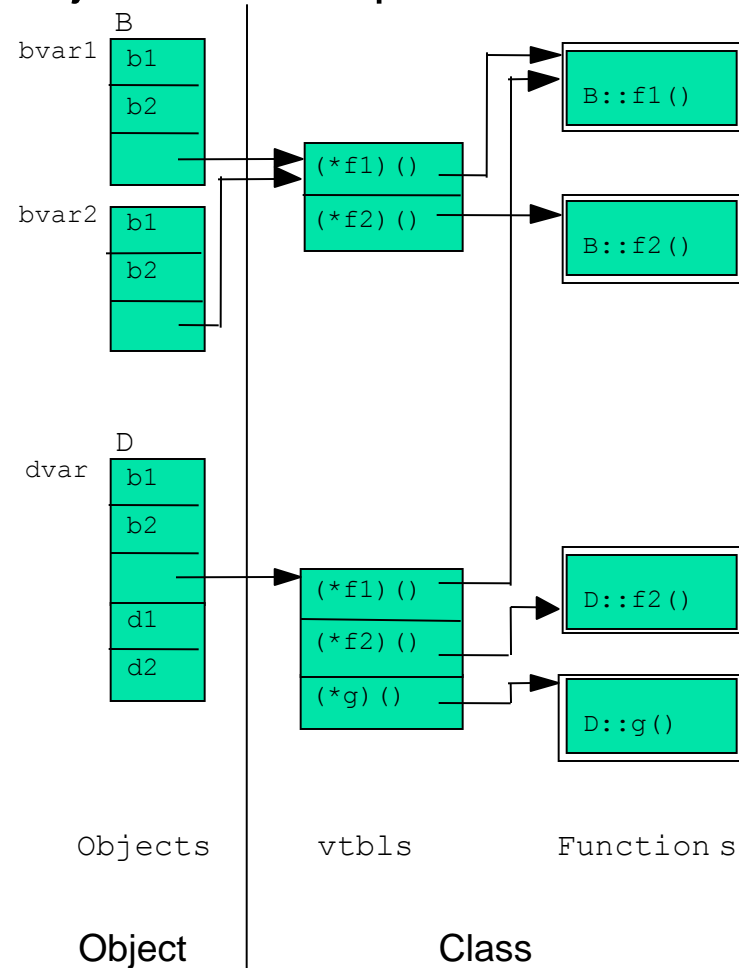
```
class B
{
    int b1, b2;
public:
    void f();
    B() {...}
};
class D: public B
{
    int d1, d2;
public:
    void g();
    D() {...}
};
B bvar;
D dvar;
```



Runtime Type Identification (cont.)

- But if they have virtual functions an object stores a pointer to them

```
class B
{
    int b1, b2;
public:
    virtual void f1();
    virtual void f2();
    B() {...}
};
class D : public B
{
    int d1, d2;
public:
    void f2();
    virtual void g();
    D() {...}
};
B  bvar1, bvar2;
D  dvar;
```



Runtime Type Identification (cont.)

- A base-class pointer can point to any derived-class object
 - If we know the actual class of the object, we can successfully *downcast* the pointer to that derived type

```
Account* aptr;  
FlexiAccount fa;  
...  
aptr = &fa;  
...  
((FlexiAccount*)aptr)->set_overdraft_limit(10);  
// how do we KNOW that ap is still pointing to  
// a FlexiAccount? What happens if it is not?
```

- Standard C++ now supports “safe” *downcasts* for *polymorphic* classes (classes with virtual functions)

Runtime Type Identification (cont.)

- `dynamic_cast<D*>(p)` will succeed if `p` points to an object of class `D` or a class publicly derived from `D`
 - Otherwise, the result is a null pointer

```
Account* aptr;  
FlexiAccount fa;  
...  
aptr = &fa;  
...  
FlexiAccount* fptr = dynamic_cast<OverdraftAccount*>(aptr);  
if (fptr != 0)  
    fptr->set_overdraft_limit(10);  
else    // error
```

Runtime Type Identification (cont.)



Unlike Java, C++ does not support "reflection"

- Obtaining full run time information about a class or object
 - Such as its data members or base classes
- There is no Object class that all objects derive from
- But the operator `typeid` returns information for objects with virtual functions

```
class type_info
{
public:
    // operators != and
    // == char* name();
private:
    ...
};
```

Output:
oa is of type
class FlexiAccount

```
#include <typeinfo>
Account a;
FlexiAccount fa;
...
if (typeid(a) == typeid(fa)) ...
...
cout << "oa is of type\n"
    << typeid(fa).name() << '\n';
...
```

Typeid returns an instance of `typeinfo`, a standard library class with an operator `==`

returns false

Multiple Inheritance

Multiple Inheritance

- In C++, it is possible to inherit from a number of different base classes.
- This is termed **multiple inheritance**.
- Each base class can be inherited using a different type of inheritance if desired.
- From Java, we are used to only having single inheritance, and all inheritance being public.
- Java supports the idea of multiple implementations of interfaces by a class.

Example

```
class Base1 {  
    // ..  
};  
  
class Base 2 {  
    // ..  
};
```

```
class Derived : public  
Base1, private Base2 {  
    // ..  
    public:  
        Derived() : Base1(),  
Base2() { }  
};
```

- Each base class has its inheritance type specified.
- Each base class constructor is placed in the initialiser list in the same order they are declared.

Explicit Scoping

- If a non-virtual base class is necessary (e.g. private inheritance for IS-IMPLEMENTED-IN-TERMS-OF), and access to a base class member is required, explicit scoping can be used.

```
void HighlyDerived::someFunction() const {  
    // ..  
    Derived1::baseClassFunction();  
    // the following operates on different  
    // base class, if Base is non-virtual  
    Derived2::baseClassFunction();}
```


Construction order

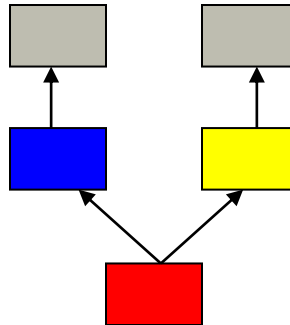
- Classes that inherit from multiple bases have their base classes constructed in the same order they are declared.
- An object is not considered created until *all* of its base class objects have also been created.

Destruction order

- First, the object's destructor is called, then the base class destructors in reverse order.
- An object is no longer valid once the object's destructor begins executing.

Common base classes

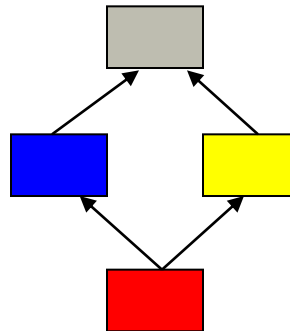
- Sometimes, you derive from two classes that both inherit from the same base class.



- This leads to two copies of data, and potential ambiguities. Which base class are you referring to?

Virtual base classes

- The usual method to get around this is to declare the base class as virtual. This leaves the following arrangement.



- A virtual base class must be initialised by the most-derived class in a hierarchy.

Virtual base class

```
class Derived1 : virtual public Base { };  
class Derived2 : virtual public Base { };  
  
class HighlyDerived : public Derived1,  
    public Derived2 {  
    // ..  
};
```

Diamond Inheritance Example

```
1 // Fig. 24.13: fig24_13.cpp
2 // Attempting to polymorphically call a function t
3 // multiply inherited from two base classes.
4 #include <iostream>
5 using namespace std;
6
7 // class Base definition
8 class Base
9 {
10 public:
11     virtual void print() const = 0; // pure vir
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18     // override print function
19     void print() const
20     {
21         cout << "DerivedOne\n";
22     } // end function print
23 }; // end class DerivedOne
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29     // override print function
30     void print() const
31     {
32         cout << "DerivedTwo\n";
33     } // end function print
34 }; // end class DerivedTwo
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
39 public:
40     // qualify which version of function print
41     void print() const
42     {
43         DerivedTwo::print();
44     } // end function print
45 }; // end class Multiple
46
47 int main()
48 {
49     Multiple both; // instantiate Multiple object
50     DerivedOne one; // instantiate DerivedOne object
51     DerivedTwo two; // instantiate DerivedTwo object
52     Base *array[ 3 ]; // create array of base-class pointers
53
54     array[ 0 ] = &both; // ERROR--ambiguous
55     array[ 1 ] = &one;
56     array[ 2 ] = &two;
57
58     // polymorphically invoke print
59     for ( int i = 0; i < 3; ++i )
60         array[ i ] -> print();
61 } // end main
```

Diamond Inheritance Example .. Cont..

```
1 // Fig. 24.14: fig24_14.cpp
2 // Using virtual base classes.
3 #include <iostream>
4 using namespace std;
5
6 // class Base definition
7 class Base
8 {
9 public:
10     virtual void print() const = 0; // pure virtual
11 }; // end class Base
12
13 // class DerivedOne definition
14 class DerivedOne : virtual public Base
15 {
16 public:
17     // override print function
18     void print() const
19     {
20         cout << "DerivedOne\n";
21     } // end function print
22 }; // end DerivedOne class
23
24 // class DerivedTwo definition
25 class DerivedTwo : virtual public Base
26 {
27 public:
28     // override print function
29     void print() const
30     {
31         cout << "DerivedTwo\n";
32     } // end function print
33 }; // end DerivedTwo class
```

```
34
35 // class Multiple definition
36 class Multiple : public DerivedOne, public DerivedTwo
37 {
38 public:
39     // qualify which version of function print
40     void print() const
41     {
42         DerivedTwo::print();
43     } // end function print
44 }; // end Multiple class
45
46 int main()
47 {
48     Multiple both; // instantiate Multiple object
49     DerivedOne one; // instantiate DerivedOne object
50
51     DerivedTwo two; // instantiate DerivedTwo object
52
53     // declare array of base-class pointers and initialize
54     // each element to a derived-class type
55     Base *array[ 3 ];
56     array[ 0 ] = &both;
57     array[ 1 ] = &one;
58     array[ 2 ] = &two;
59
60     // polymorphically invoke function print
61     for ( int i = 0; i < 3; ++i )
62         array[ i ]->print();
63 } // end main
```

When to use MI

- Multiple inheritance introduces a large number of problems, such as virtual base classes, construction order, explicit scoping and the like. When *would* you use it?
- Answer: Not often.

When to use MI

- Most systems can be modelled using single inheritance.
- MI should only be considered if base classes are protocol classes.
- Alternatively, if you are extending a library which you have not written, but the derived class needs to inherit from other classes as well, MI must be used.
- Use multiple inheritance when each inheritance would make sense in a single inheritance scenario.
- If these criteria are not met, perhaps reconsider your usage of multiple inheritance.

Operator Overloading

Operator Overloading

- We are used to using different operators on the built in types.

```
int c = a + b; // + operator
```

```
int d = ++c;   // prefix ++ operator
```

```
char e = d--;  // postfix -- operator
```

```
char f = -e;   // unary - operator
```

- It would often make sense to be able to perform these operations on classes that we have created.
- C++ allows this through **operator overloading**.

A compiler's view

- Let's assume we have a class of type T , that we are performing addition on:

```
T a, b, result;  
result = a + b;
```

- If the compiler sees this, it will attempt to resolve the addition first (operator precedence rules).
- The compiler now looks for a function that will add two T objects. It first looks for a member function

```
result = a.operator+(b);
```

A compiler's view

- If it doesn't find a member function, it tries a free function:

```
result = operator+(a, b);
```

- If it can't find any of those, it attempts to find a conversion constructor to convert either object to a form it knows.
- If that fails, the compiler will complain about a missing function declaration.

Operator Overloading

- This gives us a hint as to how to implement operator overloading.
 - As a member function. Not all operators can be made member functions.
 - As a free function. There are benefits and drawbacks to this approach.
- We will look at both approaches and when to use each. But first, some terminology.

Terminology

- **Binary operator** – an operator that works on two objects, such as addition or subtraction.
- **Unary operator** – an operator that works on one object, such as `!` (binary not), `~` (complement), or `++` and `--` (increment and decrement)
- **Ternary operator** – an operator that works on three objects. There is only one ternary operator, the conditional operator, `?:`

Terminology

- **Prefix** – Used to describe increment or decrementing. In prefix increment, the object is incremented, and the *new object* is returned.
- **Postfix** – In postfix increment, the object is incremented, and a copy of the *old object* is returned.

Member functions

- Let's try implementing a less-than operator for a class:

```
class Game {  
    private:  
        int points;  
    public:  
        bool operator<(const Game& rhs) const;  
};
```

Member functions

- Note that a comparison operator is logically const – it compares objects, not changes them. Likewise, the parameter is const.
- A comparison should return true or false – it's either less than, or not.

```
bool Game::operator<(const Game& rhs) const {  
    return (points < rhs.points);  
}
```

Free functions

- Now for the free function version. This time, the operator takes two parameters:

```
bool operator<(const Game& lhs, const Game& rhs) {  
    return (lhs.getPoints() < rhs.getPoints());  
}
```

Free functions

- Note that this is not a member function, so does not have the `Game::` scope resolution.
- As such, it needs to use accessor functions to do the comparison.
- Also, as this is no longer a member function, it is not immediately associated with the `Game` class – we have decreased cohesion.
- To limit the effect of this, declare the free function in the `game.h` file, and define it with `game.cpp`

Free functions

- Advantages:
 - Assume Game has a conversion constructor that takes an int.

`Game g2 = g + 1; // works, 1 is converted`

- If the binary `+` operator is a member function, the following won't work, despite it being logically the same as above.

`Game g2 = 1 + g; // ERROR!`

Free functions

- Advantages (cont)
 - If the binary `+` operator was a free function, the `1` would be automatically converted to a `Game` class, and the compiler would be happy.
 - This reason is often enough to make binary operators free functions.
 - Other advantages include readability (it *makes sense* that binary `+` takes *two* parameters).

Friends

- To circumvent the use of accessors in the free function, we could have declared the overloaded operator to be a **friend** of the Game class.
- Being a friend enables the function to have direct access to private and protected data members of the Game class.

Friends

- We can declare a function, or a even whole class, to be a friend of another class.
- Friendship can only be granted to another class – another class can't claim to be a friend to get around visibility controls.
- Friendship has the highest coupling of any relationship between classes, including inheritance.

Friends

```
class B { };  
  
class A {  
    // . . .  
    friend bool operator+(const A&, const A&);  
    friend class B;  
};  
  
void B::someFunction(const A& a) {  
    a.somePrivateDataMember = 2;  
}
```

Overloadable Operators

- We can overload any operator, with the exception of the scope resolution operator (`::`), the pointer to member function operator, and the conditional operator.
- We cannot overload these operators as they are either too important to change their meaning (such as scope resolution), or impossible to represent (such as conditional operator)

New operators

- We cannot define new operators. Imagine defining a \wedge operator to represent exponentiation (eg, 2^3).
- What associativity does the new operator have?

$a \wedge \wedge b \wedge \wedge c == (a \wedge \wedge b) \wedge \wedge c; \quad // \quad \text{OR maybe}$

$a \wedge \wedge b \wedge \wedge c == a \wedge \wedge (b \wedge \wedge c); \quad // \quad ?$

New operators

- We cannot specify this easily. Secondly, what precedence does the operator have?

$a^{b^c} == (a^b)^c; \quad // \text{ OR maybe}$

$a^{b^c} == a^{(b^c)}; \quad // ?$

- As we cannot specify these consistently, C++ prevents us from creating new operators.

Canonical forms

- There are a few ‘standard’ ways of implementing overloaded operators, termed **canonical forms**.
- These could be described as a set of rules for implementing types of operators.
- When defining a prefix or a postfix increment, always define *both* prefix and postfix.
- This ensures users can use both forms as necessary. It is supporting *expected behaviour*.
- Always define a postfix operator in terms of the prefix version.

Canonical pre- and postfix ++

```
class Game {  
    private:  
        int points;  
    public:  
        Game& operator++();  
        Game operator++(int);  
};
```

Canonical pre- and postfix ++

```
// prefix
```

```
Game& Game::operator++() {  
    ++points; // prefix too!  
    return *this;  
}
```

```
// postfix
```

```
Game Game::operator++(int) {  
    Game tmp = *this; // save temp value  
    ++(*this); // define postfix in terms of prefix  
    return tmp; // return tmp, making a copy  
}
```

Canonical form of operator=

- The assignment operator should always take a const reference, and return a reference to the current object. It should usually check for self-assignment.

```
Game& Game::operator=(const Game& rhs) {  
    if (&rhs != this) { // check for self-  
        assignment  
        points = rhs.points;  
    }  
    return *this;  
}
```


Conversion operators

- A conversion operator is similar to a conversion constructor.
- Let's try to convert a Game class to an integer.

```
class Game {  
    operator int() const;  
};
```

```
Game::operator int() const {  
    return points;  
}
```

Conversion operators

- Note that the conversion operator does not have a return type (similar to a constructor).
- Keep the use of conversion operators to a minimum – this will prevent hard to trace ambiguities and errors.
- Use conversion operators when converting a class to a builtin type (which do not have conversion constructors)