



Outline of the Course:

- C++, imperative programming 
- OO analysis, UML the object model 
- C++ the object based paradigm
- C++, the object oriented paradigm
- templates
- exceptions

The Language C++:

C++ = C
+ strong typing
+ classes
+ inheritance
+ overloading
+ information hiding
+ exceptions
+ templates

Bjarne Stroustrup
The C++ Programming Language
Addison Wesley 1991, 1997, 2000

Walter Savitch,
Problem Solving with C++, Addison
Addison Wesley 1996, 2001

Paul and Gail Anderson
Navigating C++
Prentice Hall 1998

Deitel & Deitel
C++ - How to program
Prentice Hall, 1999

Aims of this course

- programming in the large using oo features
- understand the oo features in C++
- how to learn a programming language

How to work:

- read notes, think
- try out and modify examples from notes
- do tutorials and labs

Assesment

2 labs, and 2 coursework exercises

2 exam questions: format: UML diagram+main+bodies

C++ the Object Based Paradigm

Object: An object has **state**, exhibits some well-defined **behaviour**, and has a unique **identity**.

Class: A class describes a **set** of objects that share a common structure, and a common behaviour.

A single object is an **instance** of a class.

An example: the class Date

Date
day: int month: int year: int
next() nextDate():Date equal(Date):bool print()

Classes in C++

WS, ch 6.2

"A class is a user-defined type. A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects."

A *class* comprises:

- *data members*: (or fields) each object of the class has its own copy of the data members (local state)
- *member functions*: applicable to objects of the class

Data members describe the state of the objects. They have a type, and are declared as:

```
type dataMemberId
```

Member functions denote a service that objects offer to their clients. The *interface* to such a service is specified by its return type and formal parameter (s):

```
returnType memberFuncId( formalParams )
```

In particular, a function with **void** return type usually indicates a function which modifies the state of the receiver.

A C++ program is organised in*

- header files,
- implementation files,
- main program file.

In this case we have three files, namely

```
//in file Date.h
class Date{
    public:
        int year, month, day;
        void next();
        Date nextDate();
        int equal(Date);
        void print();
};
```

* Important for labs, but not important for exams/coursework. More in WS ch 8.2

```

//in file Date.cpp
#include <iostream>
#include "Date.h"

void Date::next()
{ day++ ;
if ( day==32 ) { day = 1; month++;};
if (month==13) {month=1; year++;}; }

Date Date::nextDate() {
    Date newDate;
    newDate = *this;
    newDate.next();
    return newDate; }

int Date::equal(Date aDate)
{ return ((day==aDate.day)
&& (month==aDate.month)
&& (year==aDate.year)); }

```

```

    void Date::print( )
        { cout << day << " / " << month << " / " << year << endl;}

// in file mainDate.cpp
#include <iostream>
#include "Date.h"

void main() {
    Date today, tomorrow;
    today.day=28; today.month=10;
        today.year = 1997;
    today.print();
        // output  28 / 10 / 1997
    tomorrow = today;
    tomorrow.print();
        // output  28 / 10 / 1997
/*      if ( tomorrow == today )cout << "same dates\n";*/
// compiler error message:
// class Date does not define operator ==

```

```
if ( tomorrow.equal( today ) )
    cout << "today and tomorrow are equal \n";
    // output  today and tomorrow are equal

tomorrow = today.nextDate();
today.print();
    // output  28 / 10 / 1997
tomorrow.print();
    // output  29 / 10 / 1997

tomorrow.next();
today.print();
    // output  28 / 10 / 1997
tomorrow.print();
    // output  30 / 10 / 1997
}
```

The previous example demonstrates

- assignment (=) is defined for objects; the semantics is
 - this definition may be overridden in specific class by the programmer.
- comparison (==) is not defined for objects - unless explicitly declared for specific class by programmer.
- the body of a function defined in C1 as `t1 f1 (params)` is indicated by `t1 C1::f1 (params)`.
- `anObj.f1 (...)` invokes the member function `f1` for `anObj`; the function has access to all members of the object.
- a pointer to the object for which a member function is invoked constitutes a hidden argument to the function; this argument can be explicitly referred to as **this**.

Constructors – Motivation

The problem: not all integer values make sense for the data members of a `Date` object. For example, 44/-7/77 is not an appropriate date, neither is 29/02/1997.

First Solution: provide an initialization function, as in

```
#include <iostream>
class date{
public:
    int year, month, day;
    void init(int, int, int);
    void print(); };

void date::init(int d, int m, int y)
{ if ( d<1 || d>31 )
  {cout<<d << " illegal day \n"; d=1;};
  if ( m<1 || m>12 )
  {cout <<m <<" illegal month \n"; m=1;};
  if (y<1900 || y>2000 )
```

```
{cout <<y <<" illegal year\n";y=2000;};  
day = d; month = m; year = y; };
```

```
void date::print() { /*as before */ ... };
```

```
void main( )  
{ date today, someday, wrongday;  
  today.init(30,1,1996); someday.init(43,1995,1);  
  // output illegal day, month, year  
  today.print();  
  // output 30 / 1 / 1996  
  someday.print();  
  // output 1 / 1 / 2000  
  wrongday.print();  
  // output 44467 / 78945 / -90678  
}
```

The first solution was insufficient because

Therefore, C++ introduces

Constructors

WS, ch 6.2

Constructors are special member functions aiming to control object creation. They look like member functions and have the same name as the class itself.

Constructors for the class Date

```
class Date{  
public:  
    int year, month, day;  
    Date(int, int, int);  
    void next();  
    Date nextDate();  
    bool equal(Date); }  

```

```
Date::Date(int d, int m, int y)  
    { if ( d<1 || d>31 )  
        {cout<<d << " illegal day \n"; d=1;};  
    }
```

```

if ( m<1 || m>12 )
{cout <<m <<" illegal month \n"; m=1;};
if (y<1900 || y>2000 )
{cout <<y <<" illegal year \n";y=2000;};
day = d; month = m; year = y; }
...

```

```

Date today(30,1,1996), anotherDay(78,0,-5);
// Date someDay;
// compiler error message:
// no appropriate default constructor available
today.print();
Date* tomorrow = new Date(11,11,1996);
// Date* aday = new Date;
// compiler error message:
// no appropriate default constructor available

```

The previous example demonstrates

- no return type, no return anObj in constructors
- if a class has a constructor, all objects of that class will be initialised

- if the constructor requires arguments, they must be supplied.

Constructors called whenever a new object is declared or created.

Good C++ Programming Practice

Every class should have (at least one) constructor, and (probably) a destructor

Dynamic Objects - pointers revisited

WS, ch 11.1

```
#include <iostream>
#include "Date.h"
void main() {
    Date *aDate, *bDate;
    aDate = new Date(5, 11, 1996);
    bDate = new Date(5, 11, 1996);
    aDate->print();
    // output    5 // 11 / 1996
```

```

bDate->print();
        // output    5 // 11 / 1996
if ( aDate == bDate )
        cout << "aDate, bDate are same \n";
else cout << "aDate, bDate are not same\n";
        // output    aDate, bDate are not same

if ( aDate->equal(*bDate))
        cout << "aDate, bDate are equal\n";
else cout << "aDate, bDate are not equal\n";
        // output    aDate, bDate are equal

(*aDate).next();
aDate->print();
        // output    6 // 11 / 1996
bDate->print();
        // output    5 // 11 / 1996

bDate = aDate;

```

```
aDate->print();
    // output    6 // 11 / 1996
bDate->print();
    // output    6 // 11 / 1996

aDate->next();
aDate->print();
    // output    7 // 11 / 1996
bDate->print();
    // output    7// 11 / 1996

if ( aDate == bDate )
    cout << "aDate and bDate are same\n";
    // output aDate and bDate are same
}
```

The previous example demonstrates

- pointers are declared through `ClassA* aPointer`
- `new ClassA` creates at run time a new object of `ClassA`
- `(*aPointer)` dereferences `aPointer`; it is of type `ClassA`
- `aPointer->aMember` is syntactic sugar for `(*aPointer).aMember`
- `ClassA` may not appear where `ClassA*` expected, nor may `ClassA*` appear where `ClassA` expected.
- assignment (`=`) for pointers is defined; its semantics is

it introduces *aliasing*

- comparison (`==`) for pointers is defined; its semantics is

Are classes necessary?

“philosophy”, WS page 326

- One could have used a group of variables instead:

```
void next(int& day, int& month, int& year)
{ day++ ;
  if ( day==32 ) { day = 1; month++;};
  if (month==13) {month=1; year++;}; }

int todayD, todayM, todayYear;
int tomorrowD, tomorrowM, tomorrowY;
...
next(todayD, todayM, todayYear)
...
```

This would lead to ...

- One could have used a **struct** instead (by definition, a **struct** is a class with all its members public).

However, ...

Are member functions necessary?

“philosophy”

One could have used functions instead:

```
class Date{
public:
    int year, month, day; };

void next(Date& aDate)
{ if ( aDate.day==31 ) { aDate.day = 1;
    if (aDate.month==12)
        {aDate.month=1; aDate.year++;};
    aDate.month++ }
    else { aDate.day++ } };
```

```
Date td;
td.day = 26; td.month = 11; td.year = 1996;
next( td );
```

But this would lead to:

References

WS, ch 4.2

References are alternative names for objects

- `X&` means reference to `X`
- A reference must be initialised
- Initialization is different from assignment.
- Assignment to a reference, or any operation invoked on the reference does not affect the reference; it affects the object referred to.

```
#include <iostream>
#include "Date.h"
void main() {
    Date today(25, 11, 2001);
    Date tomorrow(31, 12, 2002);
    today.print();
}
```

```

        // output    25 / 11 / 2001
tomorrow.print();
        // output    31 / 12 / 2002

Date &rDate = today;
// Date &r1Date; r1Date = today;
today.print();
        // output    25 / 11 / 2001
rDate.print();
        // output    25 / 11 / 2001

today.next(); rDate.next();
today.print();
        // output    27 / 11 / 2001
rDate.print();
        // output    27 / 11 / 2001
tomorrow.print();
        // output    31 / 12 / 2002

rDate = tomorrow;

```

```
today.next(); rDate.next();
today.print();
    // output      2 / 1 / 2003
rDate.print();
    // output      2 / 1 / 2003
tomorrow.print();
    // output      31 / 12 / 2002

tomorrow.next();
today.print();
    // output      2 / 1 / 2003
rDate.print();
    // output      2 / 1 / 2003
tomorrow.print();
    // output      1 / 1 / 2003
}
```

References are similar to pointers in that:

-

References are different from pointers in that:

-
-

The main use for references is in specifying arguments and return values for functions and operators.

For example, the `Date` class should have been declared as:

```
//in file Date.h
class Date{
    public:
        int year, month, day;
        . . .
        int equal(Date&);
};
```

because

Then, the implementation of class Date would be:

```
//in file Date.cpp
int Date::equal(Date& aDate)
    { return (aDate.day == day)
      && ( aDate.month == month ) )
      && ( aDate.year == year); }
```

Then, the use of class Date would be modified as follows:

```
// in file main.cpp
int main(){. . .
    Date today, tomorrow;
    . . .
    if (tomorrow.equal(today) )
        cout << "today and tomorrow are \n";
        . . . }
```

References, Pointers and Call by Value Parameter Passing

WS, ch 4.2, but in terms of integers

Parameter passing is by value, i.e. a copy of the actual parameter is passed to the function invocation; thus it remains unaffected

- actual parameters may be affected through the use of pointers or references to them

```
// consolidates param passing, references and pointers
```

```
void f1(Date aDate) { aDate.next(); }
```

```
void f2(Date* pDate) { pDate->next(); }
```

```
void f3(Date& rDate) { rDate.next(); }
```

```
void main() {
```

```
    Date d( 3, 3, 3333);
```

```
    Date* p = new Date( 25, 2, 2525);
```

```
    Date& r = d;
```

```

d.print(); // output 3 / 3 / 3333
r.print(); // output 3 / 3 / 3333
p->print(); // output 25 / 2 / 2525

f1(d);
d.print(); // output 3 / 3 / 3333
f1(r);
r.print(); // output 3 / 3 / 3333
f1(*p);
p->print(); // output 25 / 2 / 2525

f2(&d);
    d.print(); // output 4 / 3 / 3333
f2(p);
p->print(); // output 26 / 2 / 2525
f2(&r);
r.print(); // output 5 / 3 / 3333

f3(d);

```

```

d.print();           // output  6 / 3 / 3333
f3(*p);
p->print();          // output  27 / 2 / 2525
f3(r);

d.print();           // output  7 / 3 / 3333
r.print();           // output  7 / 3 / 3333
p->print();          // output  27 / 2 / 2525

}

```

The previous example demonstrates

- `&anObj` is the address of `anObj`, it has type `T*` if `anObj` has type `T`
- `*aPnter` dereferences `aPnter`, and it has type `T` if `aPnter` has type `T*`.
- The function `f1` never affects its actual parameter

- The functions `f2`, `f3` have very similar effect. They differ in
- One should declare parameters of type `T` as
 - `T&` when
 - `T*` when
 - `T` when
- Actually, the functions `f2`, `f3` do not modify their parameters either: `f2` does not affect `p`, it modifies the object pointed at by `p`; also, `f3` does not affect `r`, it modifies the object referenced by `r`.

Constructors and Destructors

WS, but in terms of arrays

Special member functions are:

- Constructors: *... as discussed earlier ...*
- Destructors: control object destruction; they reverse the effect of constructors, and release memory. They too look like member functions with the same name as the class, preceded by `~`.

Constructors called whenever a new object is declared or created.

Destructors called implicitly whenever an object gets out of scope.

They may also be called explicitly by `anObj.~ClassName`. Also, `delete aPtr` invokes the destructor for `*aPtr`.

```
// Constructors and Destructors
```

```
#include <iostream>
```

```
#include <string.h>
```

```
class Book{
```

```
public:
```

```
    char* title;
```

```
    int serialNumber;
```

```
    Book(char*);
```

```
    ~Book(); }
```

```
int BookCounter = 0 ;
```

```
Book::Book(char *t)
```

```
{ title = new char [strlen(t)+1];
```

```
  strcpy(title,t); serialNumber = ++BookCounter;
```

```
  cout << "      constr " << title
```

```
      << " at " << serialNumber << "\n"; }
```

```
Book::~~Book()
```

```

{ cout << "          del " << title << " from "
  << serialNumber <<"; BC = "
  << --BookCounter << " \n";
  delete title; };

```

```

int main(){
  Book b1("3 Musketeers");
      // output  constr 3 Musketeers at 1
  Book *b2, *b3 ;
      // output
  { // enter inner Block
    Book b4("Iron Mask");
      // output  constr Iron Mask at 2
    b2 = new Book("20 Years Later");
      // output  constr 20 Years Later at 3
    Book *b5 = new Book("Tom Sawyer");
      // output  constr Tom Sawyer at 4
  } // exit inner Block
      // output del Iron Mask from 2; BC = 3

```

```
b2 = new Book("Polyanna");  
    // output  constr Polyanna at 4  
b2 = new Book("Shogun");  
    // output  constr Shogun at 5  
b3 = b2;  
    // output  
delete b3;  
    // output  del Shogun from 5; BC = 4  
return 1; }  
    // output  del 3 Musketeers from 5; BC = 3
```

Good C++ Programming Practice

Every class should have (at least one) constructor, and (probably) a destructor

- Constructors should ensure the integrity of the objects produced
- It is the programmer's responsibility to destroy dynamically created objects
- It is the system's responsibility to destroy declared (static or local) objects.
-

Motivating Private Members

Constructors alone cannot guarantee the integrity of data:

```
class Date{  
public:  
    int year, month, day;  
    Date(int, int, int);  
    void next();  
    Date nextDate();  
    bool equal(Date); }  

```

```
// some other part of the program
```

```
Date myBirthday(29, 2, 1980);
```

```
. . .
```

```
    myBirthday.year++ // very bad!
```

Private / Public / Protected Members

WS, ch 6.1

Members may have one of three levels of program access:

- *public* may be accessed anywhere they are within scope.
- *protected*
- *private* may only be accessed by member functions of the class or by friends of the class.

Public members describe the *interface* of an ADT; private members describe its *implementation*.

```
class Date{  
    int year, month, day;  
public:  
    Date(int, int, int);  
    void next();  
    Date nextDate();  
    bool equal(Date);    }
```

An Exercise in Private / Public Members

```
class A{
    int j;
    int f(int y) { return ++k; };
public:
    int k;
    int g(int x) { return --j; };
    void print() { cout << "      j= "
        << j << ", k= " << k << "\n"; };
    int compare(A z)
        { return (k==z.k) && (j==z.j) &&
            (z.f(1)==g(5)); };
    A(int x, int y) { j=x; k=y; }; }

void f(A x) {
    // x.j++ ;
    // compiler error: cannot access private member
    x.k++;
    // cout << x.f(1) << "\n";
    // compiler error: cannot access private member
```

```

    cout << "x.g(2)= " << x.g(2) << "\n";
}

int main() {
    A a1(10,20), a2(30,40);
    a1.print(); f(a1);
    //      a1.f(3);
//  compiler error: cannot access private member

    a1.g(5);  a1.compare(a2);
    return 0; }

```

The previous example demonstrates:

- member functions may access all members of class, even of those that do not belong to the object currently executing the function, e.g. `z.f(1)` in `A::compare()`.
- other functions may only access public members.
- inline expansion of functions, e.g. `A::compare()`.

Good C++ Programming Practice

- Make private as many members as possible.
- Data members are usually private.
- Some member functions should be private too.
- Motto: disseminate information on a “need to know” basis.

Given program P which compiles and runs successfully, take $P' = P - \text{privAnnotations}$ annotations. Does then P' compile? run? If it runs, what result does it produce?

Given program P which compiles and runs successfully, take $P' = P + \text{privAnnotations}$ annotations. Does then P' compile? run? If it runs, what result does it produce?

Friends

WS, ch 8.1

A non-member function that is allowed access to the private part of a class is a *friend* of the class. A function is made a friend of a class by a friend declaration:

```
class C2;
class C1;

class C2{
    int j;
public:
    friend int f(C1&, C2&);
    int g(C1& );
    C2(int l) { j=1; };
};

class C1{
    int i;
public:
    C1(int l) { i=l; };
    friend int f(C1&, C2&);
    friend int C2::g(C1&);
};

int f(C1& aC1, C2& aC2) {return aC1.i*aC2.j;}
```

```
int C2::g(C1& aC1) {return aC1.i* j; }
```

```
int main()  
{ C1 a(7); C2 b(3);  
  ... f(a,b)+b.g(a)  
  return 0; }
```

The previous example demonstrates

- incomplete class declarations used for mutually recursive class definitions (e.g. `class C1; class C2;`).
- friends can be functions (e.g. `int f()`) or member functions from other classes (e.g. `C2::g(C1&)`).
- inline expansion of functions (e.g. `C2::C2(int)`).

If a class is declared a friend of another class, then all the member functions of the former are friends of the latter:

```
class C3;  
class C4;
```

```
class C3{  
    int j;  
public:  
    C3(int i) {j=i};  
    friend class C4;  
};
```

```
class C4{  
public:  
    int g(C3& aC3);  
    { return 2*aC3.j;};  
};
```

Types and Classes

```
class Cowboy{ void draw(); void move(Point); }
class Circle{ void draw(); void move(Point); }
Cowboy jim; Circle c;
// jim = c;
```

```
class A{ int x; };
int anInt, char aChar; A anA;
anInt = aChar;
// anInt = anA;
```

The above example demonstrates:

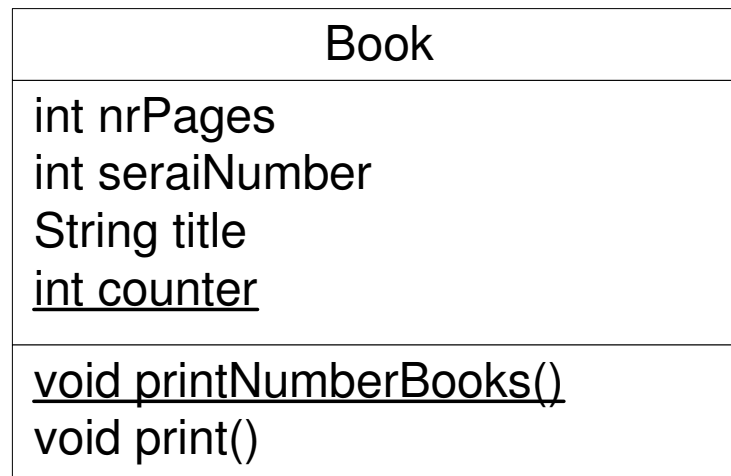
- Fundamental types can be mixed freely in assignment and expressions.
- Structure types are different, even if they have the same members.
- Structure types are different from fundamental types.

Static Members

WS, page 666, but for global variables

A data or function member of a class may be declared *static*. There is only one copy of a static data member, shared by all objects of the class in the program. A static member of class `c1` can be referred to as `c1::mem`, that is, independently of any object.

In UML class diagrams, attributes or operations that affect the class as a whole, are underlined.



For instance, instead of a global variable `BookCounter` from earlier, use a static data member, `BookCounter`

```
// Static Members, an Example
```

```
//file Book.h
```

```
#include <iostream>
```

```
#include <string.h>
```

```
class Book{
```

```
    char* title;
```

```
    static int BookCounter;
```

```
    int serialNumber;
```

```
public:
```

```
    Book (char* );
```

```
    ~Book ();
```

```
    static void printNumberBooks (); }
```

```

//file Book.cpp
#include "Book.h"
int Book::BookCounter = 0 ;

void Book::printNumberBooks()
{ cout << "== there are " << BookCounter
  << " Books \n"; }

Book::Book(char *t)
{ . . . ++Book::BookCounter . . . }

Book::~~Book() { . . . -BookCounter . . . };

//in file main.h
#include "Book.h"
int main() {
    // printNumberBooks();
    Book::printNumberBooks();
//    ... as earlier ...
    ... return 0; }

```

The previous example demonstrates

- static members accessible from member functions; also, from other classes or main program if public, or friends.
- static member function called by `ClassName::func(...)`;
- static variables may be accessed by `ClassName::varName`
- body of static member function indicated by
`resultType ClassName::func(. .) { ... }.`
- static data members have to be initialized by
`type ClassName::dataName = ...`
notice, that **static** is not repeated here
- static data members similar, but not identical to C static variables
- static members preferable to global variables or methods because:

Overloading

WS, ch. 3.6, and ch. 6.2

Different functions have typically different names, but for functions performing similar tasks on different types of objects it is best if they have the same name. When their argument types are different, the compiler can distinguish them anyway and choose the right function to call. For example:

```
class Date{
    int year, month, day;
public:
    Date( int, int, int );
    Date( int );
    Date(int, char, int );
    void next();
    void next(int); }
```

A program with overloading can be transformed into a program without overloading through

```
//exercise for overloaded functions
    int f(int i1, int i2) { return 1; };
    int f(char c1, char c2) { return 2; };
    int f(int i1, char c1) { return 3; };
    int f(char c1, int c2) { return 4; };
```

```
// The following are not overloaded
```

```
class A{
public:
    int f(char c) { return 5; }; };

class B{
public:
    int f(char c) { return 6; }; };
```

```
// The following are overloaded
```

```
class C{
public:
    int f(A x) { return 7; };
    int f(B x) { return 8; }; };
```

```

int main() {
    int x, y; char u, v; A anA; B aB; C aC;
    cout << f(x, y) << "\n"; // output 1
    cout << f(u, v) << "\n"; // output 2
    cout << f(x, u) << "\n"; // output 3
    cout << f(u, y) << "\n"; // output 4
    cout << anA.f(u) << "\n"; // output 5
    cout << aB.f(u) << "\n"; // output 6
    cout << aC.f(anA) << "\n"; // output 7
    cout << aC.f(aB) << "\n"; // output 8
}

```

The previous example demonstrates:

- member functions from same class with same identifier and different argument types overload each other.
- non-member functions with same identifier and different argument types overload each other.
- overloading resolution based on actual parameter types, ie at compile time (statically)

User Defined Operators

WS, ch 8.1

Programmers may define the meaning of operators when applied to objects of a specific class.

The operators are:

+	-	*	/	%	^	&	~	!
=	<	>	+=	-=	*=	/=	%=	
&=	=	<<	>>	>>=	<<=	==	!=	<
=	>=	&&		++	--	->*	,	->
[]	()							

new **delete**

For example, operators for complex numbers.

```
// file Complex.h
```

```
class Complex{
```

```
    double re, im;
```

```
public:
```

```
    Complex (double, double);
```

```
friend Complex operator+ (Complex, Complex);
```

```
friend Complex operator- (Complex, Complex);
```

```
friend Complex operator* (Complex, Complex );
```

```
friend Complex operator/ (Complex , Complex );
```

```
friend ostream&
```

```
operator<< (ostream&, Complex&);
```

```
};
```

```
// file Complex.cpp
```

```
Complex operator+(Complex z1, Complex z2)
```

```
{ return Complex( z1.re+z2.re, z1.im+z2.im);}
```

```
..... // similar for the operators -, /, *
```

```
ostream& operator<< (ostream& o, Complex& c)
{ o << "(r " << c.re << ", i " <<
  c.im << " )\n"; return o;};
```

```
Complex::Complex(double r, double i)
{re=r; im=i;};
```

```
// file main.cpp
```

```
int main()
{ Complex z1(2.2,4.4); Complex z2(1.1,3.3);
  cout << " z1 =" << z1 << " z2 =" << z2 ;
  // output z1 = z2 =

  cout << (z1+z2); // out (r 3.3, i 7.7)
  cout << (z1-z2); // out (r 1.1, i 1.1)
  cout << (z1*z2); // out (r 2.42, i 14.52)
  cout << (z1/z2); // out (r 2.0, i 1.3)
  cout << ((z1/z2)+(z1*z2))-z1;
  // out (r -2.08, i 8.82)

  return 0; }
```

User defined operators are syntactic sugar. They could be replaced by

1)

or by

2)

Using `Complex` the class `Complex` would be defined as

```
class Complex{  
    double re, im;  
    public:  
        Complex (double, double);  
  
};
```

and then $((z1/z2) + (z1 * z2)) - z1$ would be represented as

Using the class `Complex` would be defined as

```
class Complex{
    double re, im;
public:
    Complex (double, double);

};
```

and then $((z1/z2) + (z1 * z2)) - z1$ would be represented as

The advantage of operators is

We can now analyze the expression

```
cout << " z1+z2 =" << z1+z2 << "\n" ;
```

Operations may either be global operators or member functions. In the previous example, all operators were

Binary operators may be defined through global operators with two arguments, or through member functions with one argument.

Unary (prefix) operators may be defined through global operators with one argument, or through member functions without argument.

```
// demonstration of Operators
class A{
    int i;
public:
    A(int l);
    friend A operator+ (A, A);
    A operator- (A);
```



```

int main() {
    A anA1(5), anA2(6);
    cout << "anA1=" << anA1 << "anA2= " << anA2 << "\n";
        // output anA1 = A(i=5)      anA2 = A(i=6)
    cout << "  anA1+anA2= " << anA1+anA2 << "\n";
        // output anA1+anA2 =  A(i=11)
    cout << "  -anA2= " << -anA2 << "\n";
        // output -anA2 =  A(i=-6)
    cout << "  ++anA1= " << ++anA1  << "\n";
        // output ++anA1 =  A(i=9)
    cout << "  --anA1= " << --anA1  << "\n";
        // output --anA1  A(i=1)
    return 0; }

```

The difference between global operators and member function operators is .

Constants

WS, ch 8.1,

Adding the keyword `const` to the declaration of an entity makes it a constant rather than a variable. The value of constants may not be modified; therefore constants must be initialised.

Rationale for constants:

- for the programmer
- for the compiler

Constants restrict the way entities may be used. We may have :

- constant entities of fundamental types
- constant objects
- constant parameters
- constant result types
- constant data members
- constant member functions

```

// Demonstration of Constants
// file A.h
class A{
    const int i;
    int k;
public:
    A(int l,int m);
    friend ostream&
        operator<<(ostream& o, const A& anA);
    void f1(int l)
    void f2(int l) const;
};

// file A.cpp
A::A(int l,int m):i(l){ k=m; };

ostream& operator<<(ostream& o, const A& anA)
    { o <<" (i= "<<anA.i<<" , k= " <<anA.k <<" );
      return o; };

```

```

void A::f1(int l)
    { k=k+i+1; /* i=i-1; */ };

void A::f2(int l) const {
    int m = k+i+1;
    /* k=k+i+1; i=i-1; */};

// file Auxiliary.h
#include "A.h"
void h1(const A& anA, int l)
    { anA.f2(l); /* anA.f1(8); */ };
void h2( A& anA, int l)
    { anA.f1(l); anA.f2(2*l); };

// file main.cpp
#include "Auxiliary.h"
int main(){
    const int constInt = 44;
    const A aConst(1,1);
    // constInt = 33;

```

```

// const int anotherConst;
// aConst.f1(3);
// h2( aConst, 5);

cout << aConst<<"\n"; // output

A anA(20,20);
cout << anA<< " \n"; // output
aConst.f2(3);
cout << aConst << "\n"; // output

anA.f1(5); anA.f2(5);
cout << anA << "\n"; // output

h1( aConst, 5);
cout << aConst << "\n"; // output

h1( anA, 5); h2( anA, 5);
cout << anA << "\n"; // output
return 0; }

```

The previous example demonstrates:

- value of constants never modified
- only constant member functions may be called for constant objects (e.g. `aConst.f1(3)` illegal)
- constant arguments may be passed only if corresponding formal parameter a constant (e.g. `h2(aConst, 5)` illegal)

Given program P which compiles and runs successfully, take $P' = P - \text{constAnnotations}$ annotations. Does then P' compile? run? If it runs, what result does it produce?

Given program P which compiles and runs successfully, take $P' = P + \text{constAnnotations}$ annotations. Does then P' compile? run? If it runs, what result does it produce?

Default Arguments

For functions which require fewer arguments in the simplest, more specific case than in the general case, programmers may supply *default* arguments. The default arguments are supplied when the function is called without arguments in that position.

```
class A{
    int i;
public:
    A(int j=22) { i= j; };
    void f(int l=44) { cout << " f(" << l << " ); \n"; };
    void g(int l=44, char c='b', int m=66)
        { cout << " g(" << l << " , "
          << c << " , " << m << " ); \n"; };
    // void h1(int l=4, char c) { ... };
    void h2(int l, char c='d')
        { cout << " h2( " << l << " , " << c << " ); \n"; };
    friend ostream& operator<<( ostream& o, A& x); };
```

```
ostream& operator<<( ostream& o, A& x)
    { return o<<" A(i= " << x.i <<" ) "; };
```

```
A a1(3);
```

```
void h(A& x=a1, int l=88)
    { cout<<"      h( "<< x <<" , " << l <<" ) \n"; };
```

```
int main() {
    A myA(5); cout << "myA = " << myA << " \n";
           // output   myA =  A(i=5)
    A anA;  cout <<"anA = " <<anA<< " \n";
           // output   anA =  A(i=22)
    h(anA, 3); // output   h( A(i=22),  3);
    h(anA);   // output   h( A(i=22),  88);
    h();      // output   h( A(i=3),  88);

    anA.f(5); // output   f(5);
    anA.f();  // output   f(44);
```

```

anA.g(1, 'f', 5); // output    g(1, f, 5);
anA.g(1, 'f');   // output    g(1, f, 66);
anA.g(1);        // output    g(1, b, 66);
anA.g();         // output    g(44, b, 66);

anA.h2(5, 'g');  // output    h2(5, g);
anA.h2(5);       // output    h2(5, d);

return 0; }

```

The previous example demonstrates

- both member functions and global functions may have default arguments.
- any number of default arguments possible., parameters with default arguments have to follow those without,
e.g. `int f(int x=3, char c)` illegal)
- default arguments supplied when corresponding argument omitted from function call

Data member initializers

WS, but insufficient, for inheritance in p 817

Motivating constant and reference initializers

Constant and reference data members cannot be assigned in the bodies of the constructors. For example:

```
// in file Publisher.h
class Publisher{
    const int licenceNr;
    int numberBooks;
public:
    Publisher(int);
    void incrNumberPages(); };

// in file Book.h
#include "Publisher.h"
class Book{
    int soldSoFar;
    const int numberPages;
```

```

    Publisher& publishedBy;
public:
    Book(Publisher&, int); };

// in file Main.cpp
#include "Book.h"
void main() {
    Publisher AddisonWesley(23);
    Publisher OxfUniPress(24);
    Book AliceInWonderland(AddisonWesley, 88);
    Book NavigatingC++(AddisonWesley, 7988); }

```

So far so good; but, “naïve” constructors would not compile, eg:

```

/* in file Publisher.cpp
Publisher::Publisher(int l)
    {licenceNr=l; numberBooks=0}; */

```

is illegal, and causes a compiler error message:

```

'licenceNr' : must be initialized in constructor
            base/member initializer list
l-value specifies const object

```

and also:

```
/* in file Book.cpp
Book::Book(const Publisher& p,int nr)
    { publishedBy = p; numberPages= nr;
      soldSoFar = 0; p.incrNumberPages(); } */
```

is illegal, and causes compiler error messages:

```
'numberPages' : must be initialized
    in constructor base/member initializer list
'publishedBy' : must be initialized in
    constructor base/member initializer list
l-value specifies const object
```

Constant and reference initializers

Constant and reference data members must be initialized by data member initializers. Data member initializers appear between the method header and the method body, they are introduced through a “:” and they are separated through a “,”

For example:

```
// in file Publisher.cpp
#include "Publisher.h"
Publisher::Publisher(int l)
    :licenceNr(l) {numberBooks=0; };
```

```
// in file Book.cpp
#include "Book.h"
Book::Book(Publisher& p, int nr)
    :publishedBy(p), numberPages(nr)
    { soldSoFar=0; p.incrPages(); }
```

Notice, that non-constant, non-reference data members may also be treated in initializers, but need not, eg it is also legal to have

```
Book::Book(Publisher& p, int nr)
    :publishedBy(p), numberPages(nr), soldSoFar(0)
    { p.incrPages(); }
```

Motivating initializers for objects nested within other objects

Similarly, for objects contained within other objects just assignment in the bodies of the constructors is insufficient.

For example:

```
// in file Point.h
class Point{
    int x,y;
public:
    Point(int,int);
};
```

```
// in file Territory.h
#include "Point.h"
class Territory{
    Point sw, ne;
public:
    Territory(int,int,int,int);
    // Territory(Point,Point) would be better design!
};
```

```
// in file Main.cpp
. . .
Territory myTerritory(3,4,5,5);
```

So far so good; but ,again, “naïve” constructors would not compile, eg:

```
/* in file Territory.cpp
Territory::Territory(int k,int l,int m,int n)
{ sw = Point(k,l); ne = Point(m,n); } */
```

is illegal, and causes a compiler error message:

```
'Point' : no appropriate
           default constructor available
'Point' : no appropriate
           default constructor available
```

Initializers for objects nested within other objects

For all data members of class type, constructors have to be called in the data member initializer, ie

```
// in file Territory.cpp
#include "Territory.h"
Territory::Territory(int k, int l, int m, int n)
    :sw(k, l), ne(m, n) { }
```

Why have we been able to program without data member initializers?

In some cases, default constructors remove (or “mask”) the need for data member initializers. For example, with `Point` defined as

```
// in file Point.h
class Point{
    int x, y;
public:
    Point(int=3, int=4);    };
```

and Territory defined as before, the following is legal

```
Territory::Territory(int k,int l,int m,int n)
    { sw = Point(k,l);  ne = Point(m,n); }
```

With that constructor, the effect of

```
Territory myTerr(7,8,9,10);
```

is that the constructor `Point::Point(int,int)` is called times.

Furthermore,

```
Territory::Territory(int k,int l,int m,int n){}
```

is illegal/legal.

Such reliance on default constructors is an elegant solution/ not an elegant solution/depends on the particular case.

Self-Reference through **this**

WS, pages 880-882

this is an implicit pointer to the object currently executing a member function.

this ->dataMember <=> dataMember

this ->memFunc (...) <=> memFunc (...)

```
#include <iostream>
class A{
    int i;
public:
    A(int j){ i=j; };
    void f(){ this->i=3*this->i;
               /* this = new A(4); */ };
    void g(){ this->f(); this->f(); };
    friend ostream&
        operator<<( ostream& o, const A& x)
    { return o << " A(i= " << x.i << " ) "; };
```

```
}
```

```
void h(A& anA) {anA.f(); /* this->f(); */ };
```

```
void main( ) {  
    A anA(1);  
    cout << anA << "\n";  
        // output anA = A(i= 1 )  
  
    anA.f(); cout << anA << "\n";  
        // output anA = A(i= 1 )  
  
    anA.g(); cout << anA << "\n";  
        // output anA = A(i= 1 )  
  
    h(anA); cout << anA << "\n";  
        // output anA = A(i= 1 )  
}
```

The previous example demonstrates:

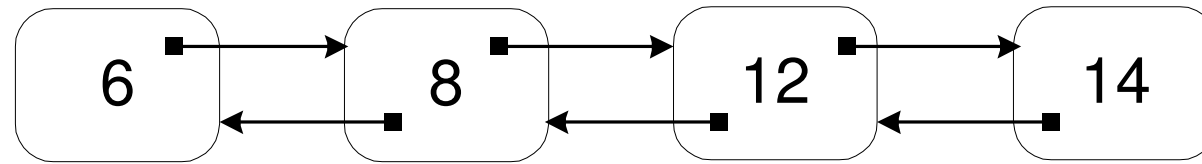
- `this`→memFnct (params) **equiv** memFnct (params),
e.g. `this`→f () **equivalent** to f () .
- `this`→dataMem **equivalent** to dataMem,
e.g. `this`→i **equivalent** to i .
- `this` only available inside a member function,
e.g. `this`→f () illegal in h .
- assignments to `this` illegal,
eg in the body of f .

In the previous examples the identifier `this` was dispensable. But in the case of

- `functCall (.., this, ..)`
- `... = this ;`

the identifier `this` is indispensable.

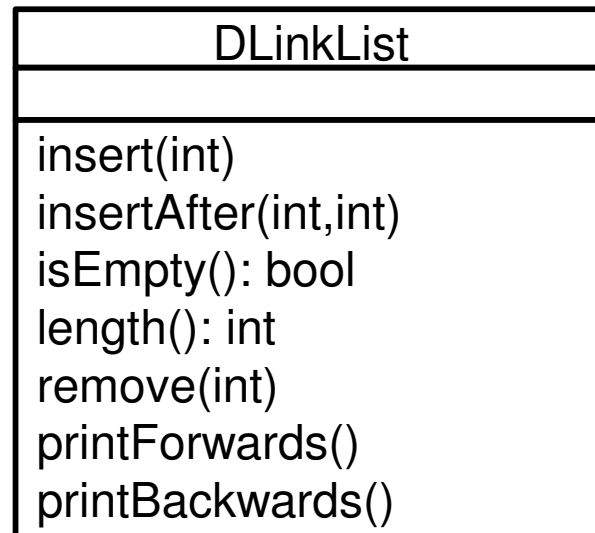
An example, where **this** is indispensable, is a doubly linked list:



(Note: above is NOT an UML object diagram)

Doubly linked lists

- contain a sequence of numbers
- may insert a number at the end or after another number
- may be printed forwards or backwards



We shall discuss

- use of **this** for the implementation of class `DLinkedList`
- design of ADTs as apparent in this example
- breaking up program into header, body and main files
- lists
- mapping UML to C++ (roughly)

At this point, it is advisable to consider possible cases, and when there is a leeway, decide the expected behaviour. This can be reflected in a “test plan” for the particular class.

For example, indicating the possibilities in italics, and the choices by strike-through:

- 1) are empty lists possible? *yes*, ~~*no*~~
- 2) a list with one element

- 3) a list with several elements
- 4) add an element into an empty list
- 5) add an element into a nonempty list
- 6) insert element after another that exists in list
- 7) insert element after a non-existing: *void, error, insert-at-end*
- 8) remove an element that is in the list
- 9) remove an element that is not in the list: *void, error*
- 10) remove an element from the empty list: *void, error*

These decisions will be reflected both in the header file and in the test function:

```
// file DLinkedList.h
#include <iostream>
const int nil = 0; // the null pointer
class DLink;
```

```

class DLinkedList{
    DLink *firstLink;
        // points to first link of chain
public:
    DLinkedList();
        // 1: creates new, empty doubly linked list
        // constructor reflects decision on 1)

void insert(int x);
        // inserts a new link with x at the
        // end of the list

void insertAfter(int x, int y);
        // inserts new link with x in list
        // after y;
        // 7: y not in list, then nothing inserted

int isEmpty() const;
        // true if receiver is an empty list

```

```
int length() const;  
    // number of elements in the list  
  
void remove(int x);  
    // removes element containing x  
    // 9, 10: does nothing if x not in list  
  
friend ostream&  
    operator<<(ostream&, const DLinkedList&);  
    // prints list forwards  
friend ostream&  
    operator>>(ostream&, const DLinkedList&);
```

And the main program, together with the expected output:

```
// file DLLMain.cpp  
#include "DLinkedList.h"  
  
int main(){  
    DLinkedList list;
```

```

// 1: printing an empty list
cout << list ;
    // out ---

// 2: inserting the first element
// 2: printing list with one element
list.insert(6);
cout << list << "\n" >> list ;
    // out 6 - ---
    // out 6 - ---

// 3: inserting 8, 12, 14
list.insert(8); list.insert(12); list.insert(14);
cout << list << "\n" >> list ;
    // out 6 - 8 - 12 - 14 - ---
    // out 14 - 12 - 8 - 6 - ---

// 6: inserting 10 after 8
list.insertAfter(10, 8);
cout << list << "\n" >> list ;
    // out 6 - 8 - 10 - 12 - 14 - ---
    // out 14 - 12 - 10 - 8 - 6 - ---

```

```

// 7: inserting 100 after 500
list.insertAfter(100, 500);
cout << list << "\n" >> list ;
    // out 6 - 8 - 10 - 12 - 14 - ---
    // out 14 - 12 - 10 - 8 - 6 - ---

// 9: removing an element that is not in list
list.remove(500);
cout << list << "\n" >> list ;
    // out 6 - 8 - 10 - 12 - 14 - ---
    // out 14 - 12 - 10 - 8 - 6 - ---

// removing an element that is in middle of list
list.remove(10);
cout << list << "\n" >> list ;
    // out 6 - 8 - 12 - 14 - ---
    // out 14 - 12 - 8 - 6 - ---

// 8: removing the first element
list.remove(6);
cout << list << "\n" >> list ;
    // out 8 - 12 - 14 - ---

```

```

    // out 14 - 12 - 8 - ---
// 8: removing the last element
list.remove(14);
cout << list << "\n" >> list ;
    // out 8 - 12 - ---
    // out 12 - 8 - ---

// 8: removing all elements
list.remove(8); list.remove(12);
cout << list << "\n" >> list ;
    // out ---
    // out ---

// 9: removing from empty list
list.remove(8); list.remove(100);
cout << list << "\n" >> list ;
    // out ---
    // out ---

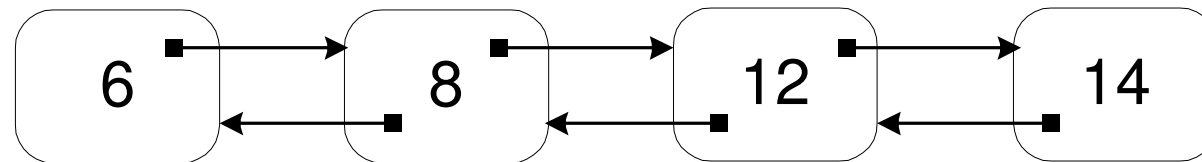
return 1;}

```

We can now consider the implementation of class `DLinkedList`.

The first question is the representation of its object.

Obviously, a non-empty list will be represented through doubly chained objects, as in

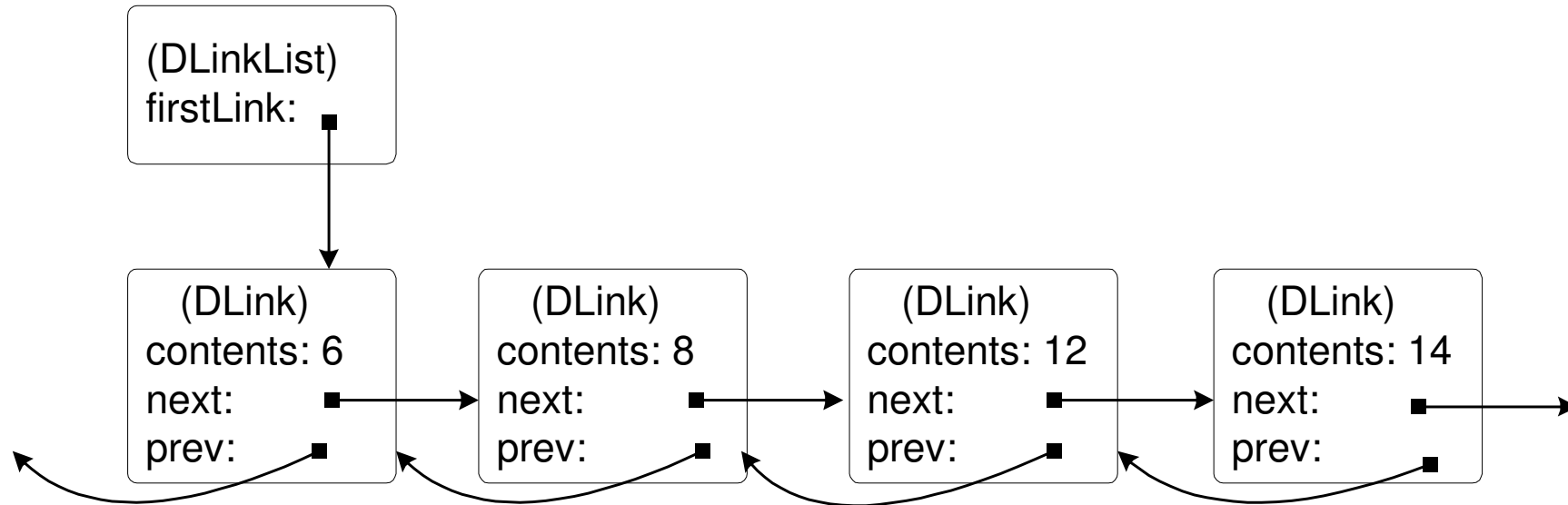


(Note: above is NOT an UML object diagram)

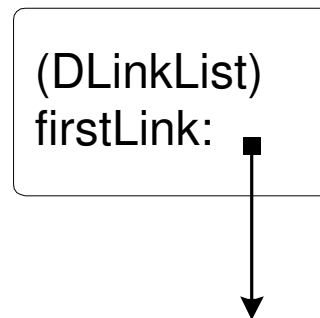
But what about the representation of the empty list? In C, it could be represented as the null pointer, but in C++ it can't (why?).

The standard solution is to distinguish between the list object (of class `DLinkedList`) and the list components (of class `DLink`)

Thus, a doubly linked list with 6, 8, 12 and 14 will be represented as:

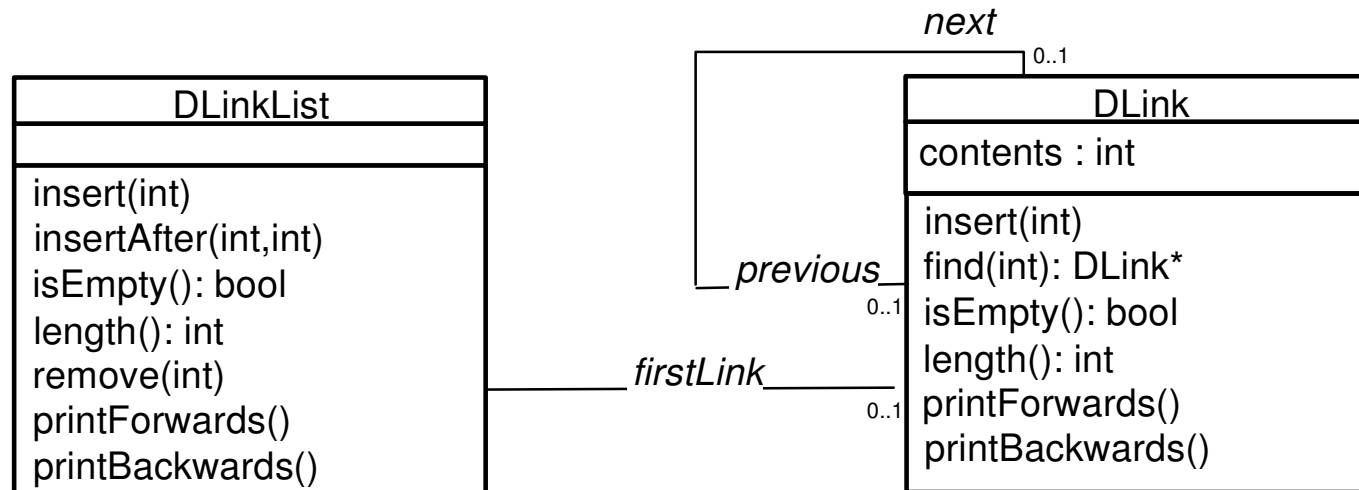


whereas an empty doubly linked list will be represented as:



(where the two above are “almost” UML object diagrams)

Doubly linked lists are represented by a header (DLinkedList) and a chain of links (DLink)



The above gives the following decomposition into files:

```
DLinkedList.h, DLinkedList.cpp,  
DLink.h, DLink.cpp,  
DLLMain.cpp.
```

The association `firstLink` was represented as a pointer to a `DLink` in `DLinkedList.h`. We shall also represent the associations `next` and `previous` as pointer to `DLinks`. Thus:

```

// file DLink.h
#include "DLinkedList.h"
class DLink{           // the link
    int contents;
        // the integer in the link

DLink *next, *prev;
        // pointers to the next and previous links
        // of the chain, possibly nil

DLink(int);
~DLink();

void insertAfterThis(int);
        // creates a new link containing x,
        // and inserts after current link

DLink* lastLink();
        // returns the last link in the chain

DLink* find(int x);
        // returns pointer to link with contents x
        // or nil, if no such link found

```

```

int length() const;
    // returns number of links in the chain

friend ostream&
    operator<<(ostream&, const DLink&);
    // prints list "forwards"
friend ostream&
    operator>>(ostream&, const DLink&);
    // prints list "backwards"
friend ostream&
    operator<<(ostream&, const DLinkList&);
    // prints list "forwards"
friend ostream&
    operator>>(ostream&, const DLinkList&);
    // prints list "backwards"

friend class DLinkList; };

```

- Notice
- all members are private, and
 - four friend declarations!

In general, the member functions of `DLinkedList` check whether the list is empty, and if not, they delegate the request to `firstLink` or `lastLink()` (ie the class `DLink`)

```
// file DLinkedList.cpp
#include "DLink.h" #include "DLinkedList.h"

DLinkedList::DLinkedList() { firstLink = nil; };

void DLinkedList::insert(int x)
    { if (firstLink==nil) // empty list
        { firstLink = new DLink(x); }
      else { (firstLink->
              lastLink())->insertAfterThis(x); }; }

int DLinkedList::isEmpty() { return (firstLink==nil); }

int DLinkedList::length() const
    { if (firstLink==nil) { return 0; }
      else { return firstLink->length(); }; } }
```

```

void DLinkedList::insertAfter(int x, int y)
    { if (firstLink==nil) { return; } // empty list
      else
        { DLink* theLink = firstLink->find(y);
          if (theLink ==nil) // y not in list
            { return; } // 7: do nothing
          else { theLink->insertAfterThis(x); } } }

```

```

void DLinkedList::remove(int x)
    { if (firstLink==nil) // empty list
      { return; } // 10: do nothing
      else
        { DLink* theLink=firstLink->find(x);
          if (theLink ==nil) // not found
            { return; } // 9: do nothing
          else { if (theLink==firstLink)
                // it is the first element
                firstLink=theLink->next;
              delete theLink; } }}

```

```
ostream& operator<<(ostream& o, const DLinkedList& l)
    {if (l.firstLink==nil) { o << " ---\n";}
    else { o << (*l.firstLink) ; }; return o;}
```

```
ostream& operator>>(ostream& o, const DLinkedList& l)
    {if (l.firstLink==nil) { o << " ---\n"; }
    else {o >> (l.firstLink->lastLink());};
    return o; }
```

```
// file DLink.cpp
```

```
#include "DLink.h" #include "DLink.cpp"
```

```
DLink::DLink(int x)
    { next=nil; prev=nil; contents=x;}
```

```
DLink::~~DLink()
    { DLink* oldPrev= prev; DLink* oldNext= next;
      if (!oldPrev==nil) oldPrev->next = oldNext;
      if (!oldNext==nil) oldNext->prev = oldPrev; }
```

```
DLink* DLink::lastLink()
{ if (next==nil) { return this; }
  else {return next->lastLink(); } }
```

```
void DLink::insertAfterThis(int x)
{ DLink* oldNextLink = next;
  DLink* newLink= new DLink(x);
  next = newLink;
  newLink->prev = this;
  if (oldNextLink!=nil)
    oldNextLink->prev = newLink;
  newLink->next = oldNextLink; }
```

```
ostream& operator<<(ostream& o, const DLink& l)
{ o << l.contents << " - ";
  if (l.next==nil){ o<< "---\n"; return o; }
  else { o << (*(l.next)) ; return o; };}
```

```
ostream& operator>>(ostream& o, const DLink& l)
{ o << l.contents << " - ";
```

```
if (l.prev==nil) { o<< "---\n"; return o;}  
else { o >> (*(l.prev)) ; return o; };
```

```
DLink* DLink::find(int x)  
{ if ( contents == x) { return this;}  
  else { if (next == nil) { return nil; }  
        else { return next->find(x);}; } }
```

```
int DLink::length() const  
{ if ( next== nil) { return 1;}  
  else { return 1 + next->length();}; }
```

This example demonstrates (among other things):

- use of **this** in body of `DLink::insertAfterThis(...)`,
`DLink::lastLink()` and `DLink::find(int)`
- pointer structures

Implicit Function Definitions and Calls

For any user-defined class `A`, unless explicitly defined by the programmer, the compiler generates:

- `A()` default constructor – if no constructor was declared by programmer for `A`
- `A(const A&)` a copy constructor
- `A& A::operator=(const A&)` assignment operator
- `//A(const A)` illegal

The compiler implicitly calls to these functions for:

- base classes
- nested classes
- parameter passing
- type conversions
- assignments

```

class A{
    int i;
    public:
        A(int l=3);
        A(const A&);
//  A(A par){i=par.i};    illegal!
//  A&(A& par){i=par.i};  illegal!
        A& operator= ( A anA);
        A operator- ( A& anA);
        A operator+ ( A anA);
        friend ostream& operator<<(ostream& o, const A&);
};

```

```

A::A(int l=3)
    { cout<<" A::A(int) call, l= " <<l<< "\n" ;
      i =l;}
A::A(const A& par)
    { cout<<" A::A(A&) call with " <<par <<"\n";
      i=par.i;}

```

```

A& A::operator= ( A anA)
    { cout << "  A&::op=(A) with " <<anA<<"\n";
      i=anA.i; return *this; }

A  A::operator- ( A&  anA)
    { cout << "  A::op-(A&)  with " <<anA<<"\n";
      return A::A(i-anA.i); };

A  A::operator+ ( A anA)
    { cout << "  A::op+(A)  call with " <<anA<<"\n";
      return A::A(i+anA.i); }

ostream& operator<< (ostream& o, const A& anA)
    { return o << "A(i= " << anA.i << ")"; };

void f(A par)
    { cout << "  f(A)  with " << par <<"\n"; };

void g(A& par)
    { cout <<"  g(A&)  with " << par <<"\n"; };

void h(const A* par)
    { cout <<"  h(A*)  with: "  <<*par <<"\n"; };

```

```

class B{
    A anA;
    A* anAPointer;
public:
    B(){ cout << "    B::B()  \n"    ; };
    // equivalent to
    //B():anA(3){ cout << "    B::B()  \n";};
    void f(const A x)
        { cout << "    B::f(A)    with " <<x<< " \n";};
    void g(const A& x)
        { cout << "    B::g(A&)  with " <<x<< " \n";};
    void h(const A* x)
        { cout << "    B::h(A*)  with " <<x<<" \n"    ;};
};

class C{
public:
    C(){ cout << "    C::C() \n"    ; };
    C(A& x){ cout <<"    C::C(A&)  with" <<x<<" \n";};
};

```

```

int main() {
    A anA1(6); // out A::A(int) call, l= 6
    A anA2(4); // out A::A(int) call, l= 4
    anA1 = anA2-(anA1+anA2);
        // out A::A(A&) call with A(i=4)
        // out A::op+(A) call with A(i=4)
        // out A::A(int) call, l= 10
        // out A::op-A(A&) call, A(i=10)
        // out A::A(int) call, l= -6
        // out A&::op=(A) call with A(i=-6)

    A anA3(3); // out A::A(int) call, l=3
    f(anA3); // out A::A(A&) call with A(i=3)
        // out f(A) called with A(i=3)
    g(anA3); // out g(A&) call with A(i=3)
    h(&anA1); // out h(A*) call with A(i=-6)

    B aB; // out A::A(int) call l=3, B::B()
    aB.f(anA1); // out A::A(A&) call with A(i=-6)
        // out B::f(A) call with A(i=-6)
}

```

```
aB.g(anA1); // out B::g(A&) call with A(i=-6)
aB.h(&anA1); // out B::h(A*) call with 0xbffffbb4

// aB = anA1;
aC; // out C::C()
aC = anA1; // out C::C(A&) call with A(i=-6)
return 0; }
```

The above example demonstrates

- implicit constructor calls upon value parameter passing
- no constructor calls for reference or pointer parameters
- implicit constructor calls for nested classes
- mysterious compiler errors when constructors unavailable
- implicit call of user defined assignment operators
- constructors may define implicit type conversions

The Object Based Features in C++ - Summary

- classes
- data members
- member functions
- static members
- constructors, destructors
- private/public/protected members
- name type equivalence for composite types
- function overloading, default parameters, constants
- operator overloading
- use of pointers, references
- member functions vs global functions
- this

The “IC Conv MSc C++ Programming Life Cycle”

- 1) Read problem description,
- 2) Draw the UML object model class diagram,
- 3) “Design” C++ program (i.e. headers, main function, compile main)
- 4) For each class `Classi`:
 - 4a) specify behaviour of each member function, especially for border cases
 - 4b) write a static `Classi::testMe()` function which demonstrates behaviour of all member functions;
 - 4c) write member function bodies;
 - 4d) test using `Classi::testMe()`.
- 5) Integrate and test.
- 6) If specification not met, continue at point 1.

Border cases are the “unexpected” cases. For example, asking the length of an empty list, receiving a negative input where a positive was expected.

NEVER jeopardise good design in order to get the program to run.

From UML to C++

UML class diagram =

- classes with attributes and operations
- associations between classes
- inheritance structure

UML is a high level description: it is concerned with the modelling of some situation, and not with how to store/represent the information

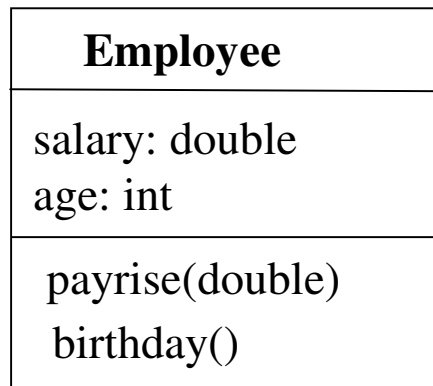
C++ is a "middle" level description: among other things, it is concerned with how to store/represent the information

UML Classes, Attributes and Operations

class: group of objects with similar properties.

attributes: data values held by objects in a class; they have fundamental type.

operations: services that may be supplied by object.



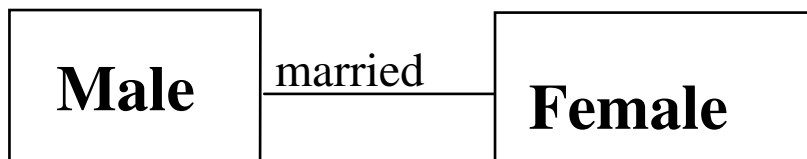
In general: UML class -----> C++

attribute -----> C++

operation -----> C++

UML Associations

Association: group of connections between objects with common structure/semantics (relationship)



1st Possibility

```
class Male{
public:
```

```
};
```

```
class Female{
public:
```

```
};
```

2nd Possibility

```
class Male{
public:
```

```
};
```

```
class Female{
public:
```

```
};
```

3rd Possibility

```
class Male{  
public:
```

```
};
```

```
class Female{  
public:
```

```
};
```

4th Possibility

```
class Male{  
public:
```

```
};
```

```
class Female{  
public:
```

```
};
```

One should prefer

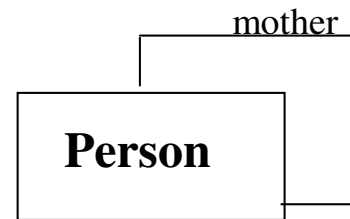
1st possibility, when

2nd possibility, when

3rd possibility, when

4th possibility, when

Associations possible between objects of same class:



1st Possibility

```
class Person{  
public:
```

```
} ;
```

2nd Possibility

```
class Person{  
public:
```

```
};
```

3rd Possibility

```
class Person{  
public:
```

```
};
```

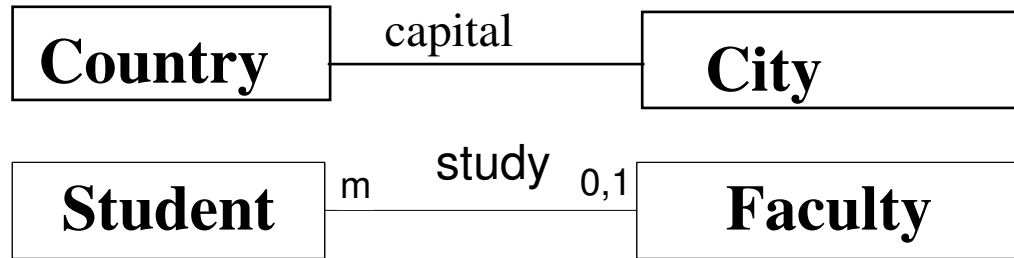
One should prefer

1st possibility, when

2nd possibility, when

3rd possibility, when

Multiplicity of Associations



```
class Student{
public:

};
```

```
class Student{
public:

};
```

1st Possibility

```
class Faculty{
public:

};
```

2nd Possibility

```
class Faculty{
public:

};
```

3rd Possibility

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

```
class Faculty{  
public:  
  
};
```

4th Possibility

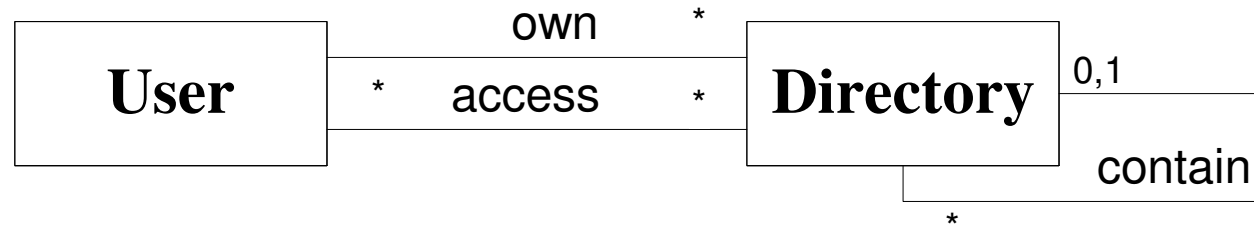
```
class Student{  
public:  
  
};
```

```
class Faculty{  
public:  
  
};
```

One should prefer

- 1st possibility, when
- 2nd possibility, when
- 3rd possibility, when
- 4th possibility, when

Multiple Associations



1st Possibility

```
class User{
public:
```

```
};
```

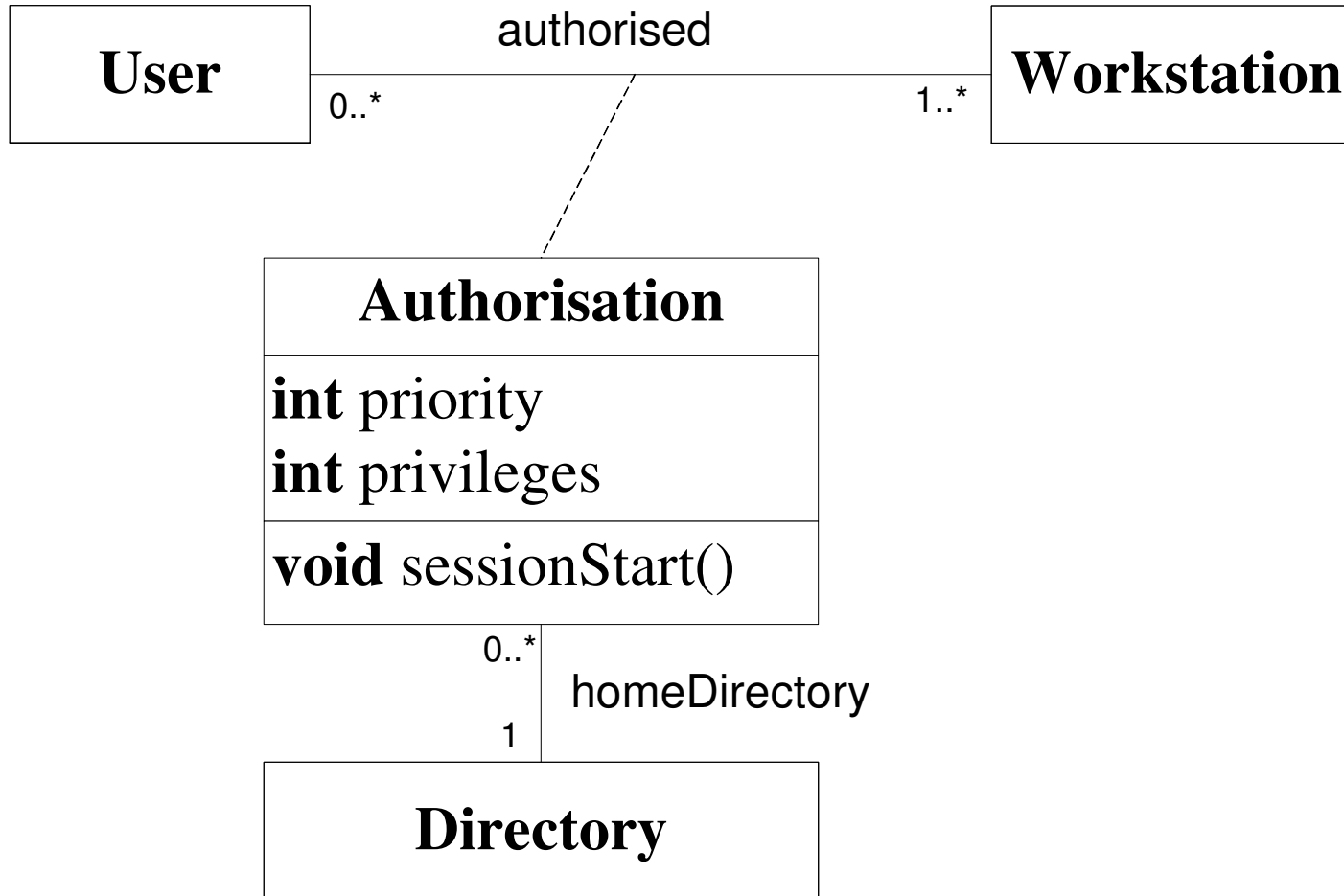
```
class Directory{
public:
```

```
};
```

2nd ,, 3rd , 4th Possibility ...

Associations as Classes

Associations may carry additional properties



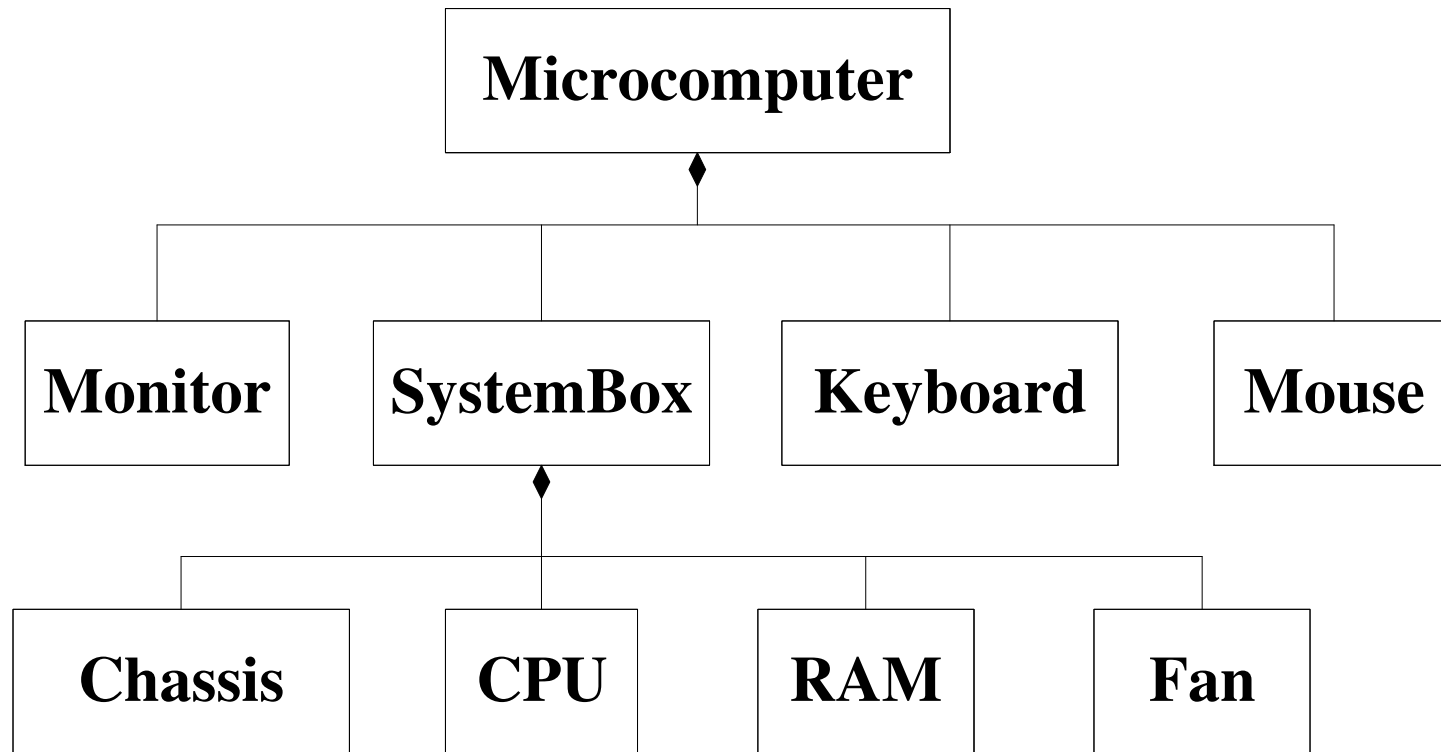
1st Possibility

<pre>class U{ public: };</pre>	<pre>class A{ public: };</pre>	<pre>class D{ public: };</pre>	<pre>class W{ public: };</pre>
---	---	---	---

2nd Possibility

<pre>class U{ public: };</pre>	<pre>class A{ public: };</pre>	<pre>class D{ public: };</pre>	<pre>class W{ public: };</pre>
---	---	---	---

Aggregation/Composition is the special "consists of" relation



In general, aggregation is represented in C++ by

Remember:

- *First* the UML object model class diagram, *then* the C++ design.
- The C++ design may contain more functions/data members than the UML class diagram.
- The C++ design may contain more classes than the UML class diagram.
- The UML class diagram may not express all aspects of specification.
- Usually, the following mapping can be applied:
 - classes
 - superclasses
 - attributes
 - operations
 - associations
 - aggregations

The trickiest decision is probably how to represent associations.

But what about superclasses?

Read on in C++ the Object Oriented Paradigm