Outline of the Course:

- C++, imperative programming ☺

- OO analysis, UML the object model ☺

- C++ the object-based paradigm

- C++, the object-oriented paradigm

- Abstract data types

- Templates

- Miscellaneous topics (exceptions etc.)

The Language C++:

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

Bjarne Stroustrup   Walter Savitch,
The C++ Programming Language   Problem Solving with C++,
Addison Wesley 2000   Addison Wesley 2009.

Paul and Gail Anderson   Deitel & Deitel
Navigating C++   C++ - How to program
Prentice Hall 1998   Prentice Hall, 2007

Aims of this part of the 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

C++ the Object-based Paradigm

with many thanks to Jayshan Raghunandan

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

Assignment Project Exam Help

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

Add WeChat powcoder

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

Classes in C++

"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 memberFunctId( 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.

* Important for labs, but not important for exams/coursework.

An example: the class Date

```
┌─────────────────────────────┐
│ Date                        │
├─────────────────────────────┤
│ day: int                    │
│ month: int                  │
│ year: int                   │
├─────────────────────────────┤
│ next()                      │
│ nextDate():Date             │
│ equal(Date):bool            │
│ print()                     │
└─────────────────────────────┘
```

In this case we have three files: `Date.h`, `Date.cpp`, `main.cpp`.

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

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

    void Date::next(){
        day++ ;
        if (this->day==32)
                { day = 1; month++;};
        if ( (*this).month == 13 )
                { month=1; year++;};    }

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


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

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

int main(){
    Date today, aDay;
    today.day=30; today.month=10; today.year =
2007;
    today.print();
        // OUT: 30 / 10/ 2007
    aDay.day=15; aDay.month= 5; aDay.year = 1984;
    aDay.print();
        // OUT: 15 / 5/ 1984

    aDay = today;
     /* if ( aDay==today )
                std::cout << "same dates\n";*/
    // compiler error message:
    // class Date does not define  operator ==
```

```cpp
    if ( aDay.equal( today ))
    std::cout << "today and tomorrow are equal
\n";
       // OUT:  today and tomorrow are equal


    aDay.next();aDay.next();
    aDay.print();
                   // OUT:  1 / 11 / 2007

    today.print();
                   // OUT:  30 / 10 / 2007

    if ( aDay.equal( today ))
    std::cout << "aDay and tomorrow are equal \n";
                   // OUT:
                   }
```

Exercise Write function `Date nextDate()` which returns the date after the receiver without affecting the receiver. Test the function.

The previous example demonstrates

- 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.

- The body of a function defined in `C1` has access to all fields and methods defined in `C1`; eg `day` means the `day` field of receiver.

- Assignment (=) is defined for all objects (unless overridden by programmer – more later) – that separately.

- comparison (==) is not defined for objects - unless explicitly declared for specific class by programmer.

- **this** is an implicit argument to all member functions; it is a pointer to the object for which the function is invoked; `day` is a shorthand for `this->day`.

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

```cpp
class Date{
public:
        int year, month, day;
        void init(int, int, int);
        …  };
…
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 )
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

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

void Date::print() { /*as before */ …. };


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

The first solution was insufficient because

Therefore, C++ introduces ….

Constructors

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

```cpp
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>3000 )
```

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

        …

    int main(){
        Date today(30,1,2008), anotherDay(78,10,1986);
        // OUT: 78 illegal day
        today.print();        // OUT: 30 / 1 / 2008
        anotherDay.print();   // OUT: 1 / 10 / 1986

        Date* tomorrow = new Date(11,21,1996);
        // OUT: 21 illegal month
        tomorrow->print();    // OUT: 11 / 1 / 1996


      // Date someDay;
      // compiler error message:
      // no appropriate default constructor
available
      //   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 (destructors discussed later).

Dynamic Objects - pointers revisited

```cpp
#include <iostream>
#include "Date.h"
using namespace std;
int main(){
    Date *aDate, *bDate;
    aDate = new Date(5, 11, 1996);
    bDate = new Date(5, 11, 1996);
    aDate->print();   // OUT:    5 / 11 / 1996
    bDate->print();   // OUT:    5 / 11 / 1996

    if ( aDate == bDate )
          cout << "aDate, bDate are same \n";
    else cout << "aDate, bDate are not same\n";
            // OUT:  aDate, bDate are not same


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

```cpp
                   // OUT: aDate, bDate are equal

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

    bDate = aDate;
    aDate->print();      // OUT: 6 / 11 / 1996
    bDate->print();      // OUT: 6 / 11 / 1996

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

if ( aDate == bDate )
        cout << "aDate and bDate are same\n";
          // OUT: 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?

- 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?

One could have used functions instead:

```cpp
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

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.

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

```cpp
                   // OUT:      25 / 11 / 2001
tomorrow.print();
                   // OUT:      31 / 12 / 2002


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


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


rDate = tomorrow;
```

```
today.next();  rDate.next();
today.print();
            // OUT:        2 /   1 / 2003

rDate.print();
            // OUT:        2 /   1 / 2003

tomorrow.print();
            // OUT:       31 / 12 / 2002


tomorrow.next();
today.print();
            // OUT:        2 /   1 / 2003

rDate.print();
            // OUT:        2 /   1 / 2003

tomorrow.print();
            // OUT:        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:

```cpp
//in file Date.h
    class Date{
        public:
            int year, month, day;

            . . .

            int equal(Date&);
            };
```

because

Then, the implementation of class `Date` would be:

```cpp
//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:

```cpp
// 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

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

```cpp
// consolidates param passing, references and pointers
void f1(Date  aDate){ aDate.next(); }
void f2(Date* pDate){ pDate->next(); }
void f3(Date& rDate){ rDate.next(); }

int  main(){
    Date  d( 2, 12, 2222);
    Date* p = new Date( 15, 1, 1919);
    Date& r = d;

    d.print();                   // OUT:  2 / 12 / 2222
```

```
p->print();                    // OUT:  15 / 1 / 1919
 r.print();                    // OUT:  2 / 12 / 2222

f1(d);
 d.print();                    // OUT:  2 / 12 / 2222
f1(*p);
 p->print();                   // OUT:  15 / 1 / 1919
f1(r);
 r.print();                    // OUT:  2 / 12 / 2222

f2(&d);
    d.print();                 // OUT:  3 / 12 / 2222
f2(p);
 p->print();                   // OUT:  16 / 1 / 1919
f2(&r);
 r.print();                    // OUT:  4 / 12 / 2222

f3(d);
 d.print();                    // OUT:  5 / 12 / 2222
f3(*p);
```

```
    p->print();                    // OUT:  17 / 1 / 1919
    f3(r);
    r.print();                     // OUT:  6 / 12 / 2222

    d.print();                     // OUT:  6 / 12 / 2222
    p->print();                    // OUT:  17 / 1 / 1919
    r.print();                     // OUT:  6 / 12 / 2222
```

  }

The previous example demonstrates

- `&anObj` is the address of `anObj`, it has type `T*` if `anObj` has type `T`

- `*aPnter` derefences `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

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 aPntr` invokes the destructor for `*aPntr`.

```cpp
// in file Book.h
class Book{
public:
        char* title;
        int serialNumber;
        Book(char*);
        ~Book();   };

extern int bookCounter;

// in file Book.cpp
#include <iostream>
#include <string.h>
#include "Book.h"
using namespace std;

Book::Book(char *t)
{ title = new char [strlen(t)+1];
  strcpy(title,t); serialNumber = ++bookCounter;
  cout << " constr " << title
```

```cpp
                   << " at "<< serialNumber << endl;    }

  Book::~Book()
  { cout << " del " << title  << " from "
          << serialNumber <<"; BC = "
           << --bookCounter << endl;
      delete [] title; };


  // in file Book.cpp
  #include <iostream>
  #include <string.h>
  #include "Book.h"
  using namespace std;


  int main( ){
     Book b1("Seagull");
               // OUT:  Seagull at 1
        Book *b2, *b3 ;
               // OUT:
```

```
{ Book b4("Iron Mask");
        // OUT:  constr Iron Mask at 2
    b2 = new Book("Moonstone");
        // OUT:  constr Moonstone at 3
 Book *b5 = new Book("Tom Sawyer");
        // OUT:  constr Tom Sawyer at 4
 }      // OUT: del Iron Mask from 2; BC = 3
b2 = new Book("Polyanna");
        // OUT:  constr Polyanna at 4
b2 = new Book("Shogun");
        // OUT:  constr Shogun at 5
b3 = b2;
        // OUT:
delete b3;
        // OUT:  del Shogun from 5; BC = 4
cout << "Now " << bookCounter << " books left
\n";
return  1; }
        // OUT:  del Seagull from 1; BC = 3
```

So, we have memory leaks!

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 job to destroy declared (static or local) objects.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Motivating Private Members

Constructors alone cannot guarantee the integrity of data:

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

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

    Date myBirthday(29,2,1940);

    . . .

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

Private / Public / Protected Members

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*. Members are private by *default*.

Therefore

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

is a shorthand for

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

An Exercise in Private / Public Members

```cpp
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 << endl;};
    int compare(A z)
        { return (k==z.k)&&(j==z.j)&&
```

```cpp
                   (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) << endl;
//  compiler error: cannot access private member
    cout << "x.g(2)= " << x.g(2) << endl;
}


int main(void){
    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-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+privAnnotations` annotations. Does then `P'` compile? run?   If it runs, what result does it produce?

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Friends

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=l };
};
```

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

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

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

```
                    Book
      ─────────────────────────────
      serialNumber:int
      title:string
      counter:int
      ─────────────────────────────
      void printNumberBooks()
      void print()
```
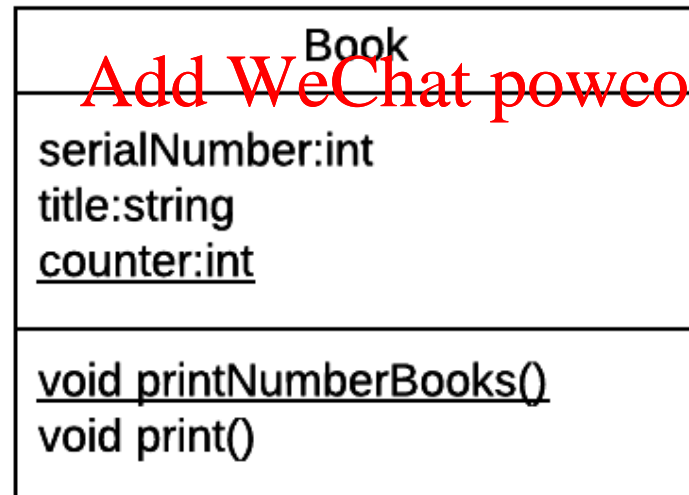
For instance, instead of a global variable `BookCounter` from earlier, use a static data member, `BookCounter`

```cpp
//file Book.h
class Book{
      char* title;
      static int counter;
      int serialNumber;
public:
      Book(char*);
      ~Book();
      static void printNumberBooks();}


//file Book.cpp
#include ….
int Book::counter = 0 ;
```

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


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


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


//in file main.h
#include ….
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

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

```cpp
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

```cpp
//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) << endl;    // OUT:    1
    cout << f(u,v) << endl;    // OUT:    2
    cout << f(x,u) << endl;    // OUT:    3
    cout << f(u,y) << endl;    // OUT:    4
    cout << anA.f(u) << endl;  // OUT:    5
    cout << aB.f(u) << endl;   // OUT:    6
    cout << aC.f(anA) << endl; // OUT:    7
    cout << aC.f(aB) << endl;  // OUT:    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)

Constants

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

```cpp
class A{
    const int i;
    int k;
 public:
    A(int l,int m);
    void f1(int l)
    void f2(int l) const;
};
```

```cpp
A::A(int l,int m):i(l){   k=m;  };
```

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

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


void h1(const A& anA, int l)
    { anA.f2(l); /* anA.f1(8); */ };
```

```cpp
void h2( A& anA, int l)
   { anA.f1(l); anA.f2(2*l); };

int main(void){
    const int constInt = 44;
   const A aConst(1,1);
   //   constInt = 33;
    //  const int anotherConst;
    //  aConst.f1(3);
    //  h2( aConst, 5);

   cout  << aConst<<endl;
   A anA(20,20);
   aConst.f2(3);

   anA.f1(5); anA.f2(5);
   h1( aConst, 5);

   h1( anA, 5); h2( anA, 5);
   return 0;}
```

The previous example demonstrates:

- value of constants never modified

- only constant functions may be called for constant objects
  (eg `aConst.f1(3)` illegal)

- constant arguments may be passed only if corresponding
  formal parameter a constant (eg `h2(aConst,5)` illegal)

- constant fields initialized in initialization part of
  constructors.

Given program `P` which compiles and runs successfully.

1) Take `P'=P-constAnnotations` annotations. Does then `P'`
compile?   run?   If it runs, what result does it produce?

2) take `P'=P+constAnnotations` annotations. Does then `P'`
compile?   run?   If it runs, what result does it produce?

Good C++ Programming Practise:
Give *as many* constant annotations as possible!

User Defined Operators

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.

```cpp
// file Compl.h
#include <iostream>
using namespace std;
class Compl{
    double re, im;
public:
    Compl (double,double);
    friend Compl operator+
        (const Compl&, const Compl&);
    friend Compl operator-
        (const Compl&, const Compl&);
    ...... // similar for operators -, /, *
    friend ostream& operator<<
        (ostream&, const Compl&);
};

// file Compl.cpp
#include "Complex.h"
Compl operator+  (const Compl&, const Compl&)
    { return Compl( z1.re+z2.re), z1.im+z2.im));}
```

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

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


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


// file main.cpp
#include "Complex.h"
int main(){
    Compl z1(2.2,4.4);   Compl z2(1.1,3.3);
    cout << "z1 =" << z1 << "z2 =" << z2 ;
        // OUT: z1 = ( 2.2, 4.4 )
        // OUT: z2 = ( 1.1,  3.3 )
    cout << (z1+z2);    // OUT:  ( 3.3,   7.7 )
    cout << (z1-z2);    // OUT:  ( 1.1,   1.1 )
    cout << (z1*z2);    // OUT:  ( 2.42,   14.52 )
    cout << (z1/z2);    // OUT:  ( 2,   1.33333 )
    cout << ((z1/z2)+(z1*z2))-z1;
// OUT:  ( 2.22,   11.4533 )
    return 1; }
```

User defined operators are syntactic sugar. They could be replaced
by   1)

or by 2)

Using 1)   the class `Compl` would be defined as


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

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

Using   the class `Compl` would be defined as

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



};
```

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

The advantage of operators is

Aside: Thus, knowing that << has been overloaded, as

```
ostream& operator<<₁ (ostream&, string)
ostream& operator<<₂ (ostream&, double)
ostream& operator<<  (ostream&, char)
ostream& operator<<₃ (ostream&, const Compl&)
```

we can analyze the expression

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

Operators summary

Operators 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.

NOTE The new ISO standard expects some operators to take constant reference parameters, e.g.:

```
friend Compl operator+ (const Compl&, const Compl&)
```

or

```
Compl operator+ (const Compl&)
```

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


A::A(int l) {i =l;}

A operator+ (const A& a1, const A& a2)
    { return A(a1.i+a2.i); };

A A::operator- (const A& anA) const
    { return A(i-anA.i); };
```

```cpp
A& operator++( A& a1)
    { a1.i=a1.i+4; return a1; };

A& A::operator--( )
    { i= i-4; return *this; };

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


int main(void) {
    A anA1(5), anA2(6);
    cout << "anA1= "<< anA1 << " anA2= "<< anA2 << endl;
            // OUT: anA1 = A(5)      anA2 = A(6)
    cout << "  anA1+anA2= " << anA1+anA2 << endl;
            // OUT: anA1+anA2 =  A(11)
    cout << "  anA1-anA2= " << anA1-anA2 << endl;
            // OUT: anA1-anA2 =  A(-1)
    cout << "  ++anA1= " << ++anA1  << endl;
            // OUT: ++anA1 =  A(9)
```

```
   cout << "  --anA1= " << --anA1  << endl;
         // OUT: -anA1 A(5)
return 0;}
```

In the above,   are global operators.

And   are member function operators.

The difference between global operators and member function
operators is

Why did we not declare the + operator as
```
   friend A operator+ (const A&, const A&) const;
```

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.

```cpp
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='c', 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); };
```

```cpp
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(void){
    A myA(5); cout<<" myA = " << myA << " \n";
                // OUT:   myA =  A(i=5)
    A anA; cout << " anA = " <<anA<< "\n";
                // OUT:   anA =  A(i=22)
    h(anA,3); // OUT:   h( A(i=22),  3);
    h(anA);   // OUT:   h( A(i=22), 88);
    h();      // OUT:   h( A(i=3), 88);

    anA.f(5); // OUT:   f(5);
    anA.f();  // OUT:   f(44);
```

```
anA.g(1,'f',5);  // OUT:    g(1,f,5);
anA.g(1,'f');    // OUT:    g(1,f,66);
anA.g(1);        // OUT:    g(1,b,66);
anA.g();         // OUT:    g(44,b,66);

anA.h2(5, 'g');  // OUT:    h2(5,g);
anA.h2(5);       // OUT:    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

Motivating constant and reference initializers

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

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


// 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"
int main(){
    Publisher AddisonWesley(23);
    Publisher OxfUniPress(24);
    Book AliceInWonderland(AddisonWesley,88);
    Book NavigatingCPP(AddisonWesley,7988); }
```

So far so good; but, "naive" 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;   } */
```

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 intialized 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:

```cpp
// 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;    }
```

Notice, that non-constant, non-reference data members may, but need not, also be treated in initializers. Therefore, following is legal

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

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:
```cpp
// 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!
};
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

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

So far so good; but ,again, "naïve" constructors would not compile, e.g.:

```
/* 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, i.e.

```cpp
// 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

```cpp
// 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 twice.

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.

Implicit Function Definitions and Calls
(with thanks to Tasos Maroudis, Tristan Allwood, David Cunningham, Jayshan Raghunandan, and Max Troy)

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

- `A(), ~A()` default constructor and destructor

- `A(const A&)` copy constructor

- `A& operator=(const A&)` copy assignment

The compiler implicitly inserts calls to these functions for:

- base classes

- nested classes

- parameter passing

- type conversions

- assignments

In the following, we make these calls explicit, by defining our own versions of the implicitly defined functions

```cpp
class A{
    int i;
public:
    A(int l=2);
    A(const A&);
//  A(A  par){i=par.i};    illegal!
//  A&(A& par){i=par.i};   illegal!
    A& operator=(const A&);
    friend ostream& operator<<
                    (ostream& o, const A&);
    ~A();
};


A::A(int l) {
    cout<< " A::A(int),"<<l<<","<<(int*)this
        << endl ;
    i =l;    }
```

```cpp
A::A(const A& a) {
      cout<<" A::A(const A&), , this: "
          << *this <<  " , a: " << a <<endl;
      i=a.i;   }

A::~A(){ cout <<" A::~A for " << *this << endl; }

A& A::operator= (const A& a)  {
     cout << " A::operator=(const A&) this: "
          << *this <<  " , a: " << a <<endl;
     i=a.i;
     return *this; }

ostream& operator<< (ostream& o, const A& a)
        {  return o <<   "   " <<
           (int*)&a  << ": A(" << a.i <<  ")\n" ;
}
```

Assignments, constructors and destructors

```
int main( ){
    A a1;
    // OUT: A::A(int) 2, be7fa0
    A a2(6);
    // OUT: A::A(int) 6, be7f98

    a1=a2;
    // OUT: A&::op=(const A&),
    //          this;    be7fa0: A(2),
    //          a:       be7f98: A(6)

    a1=A(9);
    // OUT:
        A::A(int) 9, be7fa8
        A&::op=(const A&), this:    be7fa0: A(6),
                            a:      be7fa8: A(9)
        A::~A for     be7fa8: A(9)
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

```
     return 0;}
// OUT
   A::~A for     be7f98: A(6)
   A::~A for     be7fa0: A(9)
```

The above example demonstrates

- implicit call to assignment operator

- implicit destructor call when object out of scope,

- implicit destructor call for temporary values.

What about parameter passing?

We define some more functions:

```cpp
void g(A& a)  {
      cout <<" g(A&), a: "<< a <<endl;
      };
void h(const A* a) {
      cout <<" h(A*), *a: "  <<*a <<endl;
      };
A ff( ){
          return A(16);
      }
A gg( ){
       A anA; anA = A(17); return anA;   }
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

And then execute the following code

```
A a1(4);
// OUT
  A::A(int)    4, 8113c0

f(a1);
// OUT
  A::A(const A&), this: 8113c8: A(-1082059784),
                     a:     8113c0: A(4)
  f(A), a:     8113c8: A(4)
  A::~A for    8113c8: A(4)

g(a1);
// OUT
  g(A&), a:     8113c0: A(4)

h(&a1);
// OUT
  h(A*), *a:    8113c0: A(4)
```

```
   a1 = ff();
// OUT
   A::A(int),  16,8113d0
   A&::op=(const A&), this:    8113c0: A(4),
                             a:    8113d0: A(16)
   A::~A for    8113d0: A(16)

a1 = gg();
// OUT
   A::A(int),    2,8113d8
   A::A(int),   17,81138c
   A&::op=(const A&), this: 8113d8: A(2),
                          a:    81138c: A(17)
   A::~A for  81138c: A(17)
   A&::op=(const A&), this: 8113c0: A(16),
                          a: 8113d8: A(17)
   A::~A for   8113d8: A(17)
```

The above example demonstrates

- implicit call of copy constructor when passing value parameters

What about nested objects?

We define the following class,

```cpp
class B{
    A* anAPointer;
    A anA;
public:
    B(){ cout << "B::B(), this: " << (int*)this
            << " \n" ; };
    B(int i){ cout << "B::B(int), this: "
            << (int*)this << "   \n" ;
            anA=A(i);  }
    B(char c):anA(5) { cout << "   B::B(char), this:"
                        << (int*)this << endl; }
    ~B( ) { cout << " B::~B(), for "<< (int*)this
            << "   \n" ; }
};
```

and execute

```cpp
int main(void){
    A anA1(5);
    // OUT
        A::A(int),  5, de89b8

    B aB;
    // OUT
        A::A(int),  2, de89b0
        B::B(int), this: de89ac

    B aB1(9);
    // OUT
      A::A(int),    2, de89a4
      B::B(int), this: de89a0
      A::A(int),    9, de895c
      A&::op=(const A&), this: de89a4: A(2),
                         a:     de895c: A(9)
      A::~A for de895c: A(9)
```

```
    aB2('z');
      // OUT
        A::A(int),  de8998
        B::B(char), de8994
   return 0;}
    // OUT
   B::~B()  for de8994
    A::~A for de8998: A(5)
```

```
    B::~B()for de89a0
    A::~A for   de89a4: A(2)
```

```
    B::~B()for  xxdede98?
```

```
    A::~A for 0xbfde89b0: A(2)

    A::~A for   0xbfde89b8: A(5)
```

The above example demonstrates

- implicit call of constructor /destructor of nested objects.

What about implicit conversions?

We define the following class,

```cpp
class C{
public:
    C(){  cout << "  C::C()\n"  ; };
    C(A& a){ cout <<"  C::C(A&), a: " <<a<<endl;};
};
```

and execute

```cpp
    A anA1;
    // OUT
     A::A(int), 2, f2daf4
    C    aC;
    // OUT
    C::C()
    aC = anA1;
    // OUT
    C::C(A&), a:    f2daf4: A(2)
```

The last four examples demonstrate

- implicit constructor calls upon value parameter passing

- implicit destructor calls when object out of scope

- no constructor calls for reference or pointer parameters

- implicit constructor calls for nested classes

- implicit constructor calls for nested classes

- implicit call of user defined assignment operators

- constructors may define implicit type conversions

Does all this matter?

- not that much, if you only use pointer or reference types,

- but you want to be aware of it

Self-Reference through `this`

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

```
this ->dataMember    <=>    dataMember

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

For example, Assignment Project Exam Help
```
class A{
    int i;        https://powcoder.com
public:
    A(int i){ Athisd->iWeChfat powcoder
    void f(){ this->i=3; this->f(); f(); }
             /* this =new A(4); */ };
}

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

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**->i(; illegal in n.

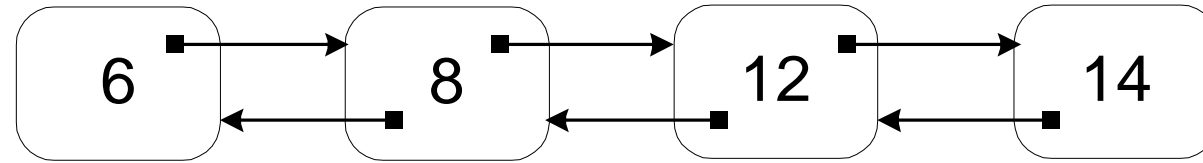- assignments to **this** illegal,
  e.g. in the body of f.

In the previous, 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:

Doubly Linked Lists

(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

```
DLinkList

insert(int)
insertAfter(int,int)
isEmpty(): bool
length(): int
remove(int)
printForwards()
printBackwards()
```

We shall take this opportunity to discuss

- use of **this** for the implementation of class `DLinkList`

- design of ADTs as apparent in this example

- test driven development

- breaking up program into header, body and main files

- lists

- mapping UML to C++ (roughly)

Test driven development: 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 in header files and in test function:

```cpp
// file DLinkList.h
#include <iostream>
using namespace std;
#include "DLink.h"


class DLinkList{
    DLink *firstLink;   // first link of chain
public:
    DLinkList();
        // 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

    bool 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 DLinkList&);
        // prints list forwards
    friend ostream&
        operator>>(ostream&, const DLinkList&);
}
```

And the main program, together with expected output

```cpp
// file DLLMain.cpp
#include "DLinkList.h"

int main(){
    DLinkList 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 << endl >> list ;
        // out 6 - ---
        // out 6 - ---

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

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

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

// removing an element from middle of list
list.remove(10);
cout <<  list << endl >> list ;
    // out 6 - 8 - 12 - 14 - ---
```

```cpp
        // out 14 - 12 - 8 - 6 - ---

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

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

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

// 9: removing from empty list
list.remove(8);   list.remove(100);
cout <<  list << endl >> list ;
```

```
        // out ---
        // out ---

return 1;}
```

Assignment Project Exam Help
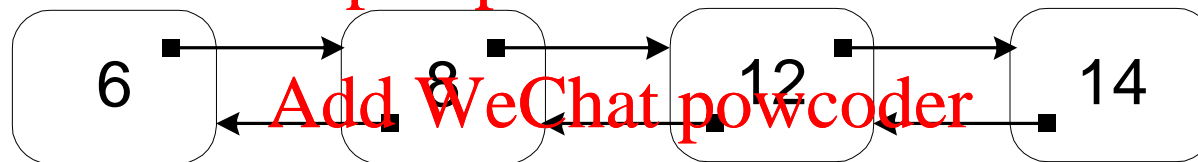
https://powcoder.com

Add WeChat powcoder

Representation of `DlinkList.`

We can now consider the represenation of class `DlinkList`.

The first question id the representation of its object.

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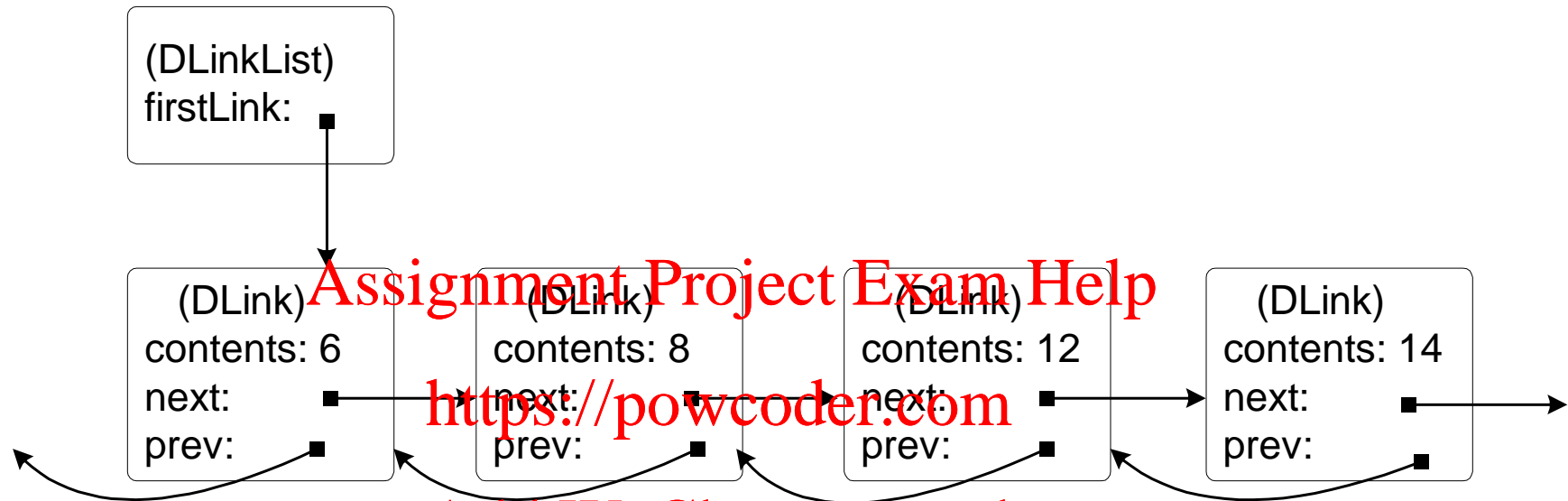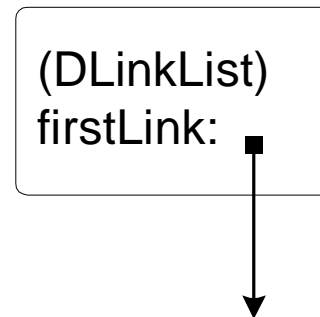

(Note: above is NOT an UML object diagram)

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 `DlinkList`) and the list components (of class `Dlink`)

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

(DLinkList)
firstLink:

(DLink)
contents: 6
next:
prev:

(DLink)
contents: 8
next:
prev:

(DLink)
contents: 12
next:
prev:

(DLink)
contents: 14
next:
prev:

whereas an empty doubly linked list will be represented as:

(DLinkList)
firstLink:

(where the two above are "almost" UML object diagrams)

Doubly linked lists are represented by a header (`DLinkList`) and a chain of links (`DLink`)
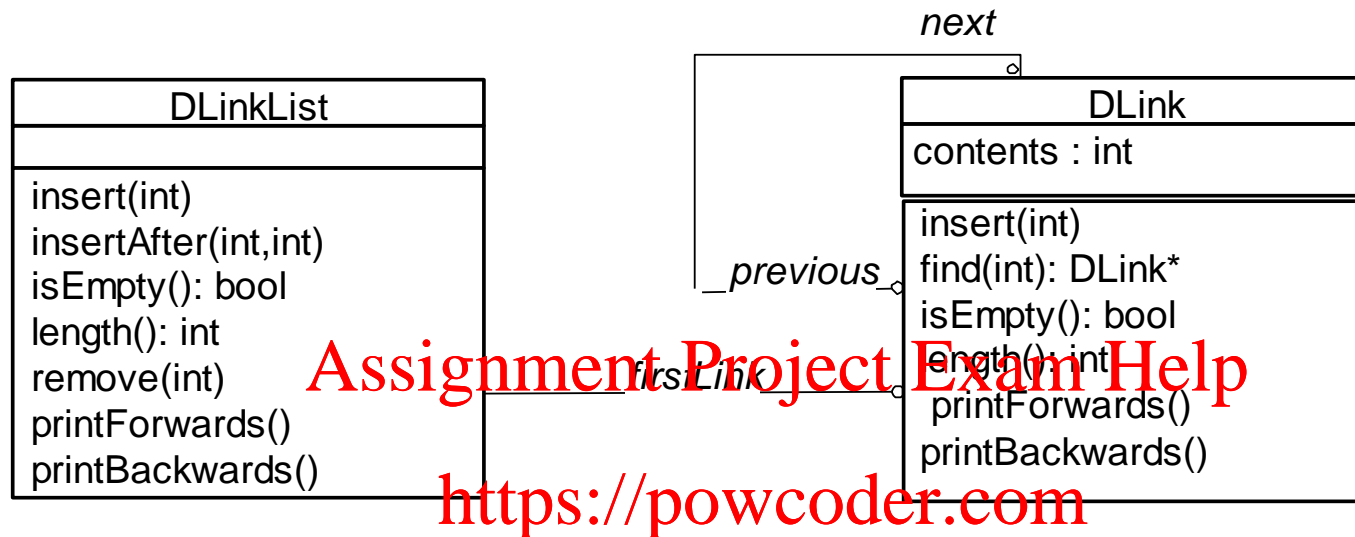
The above gives the following decomposition into files:

```
DlinkList.h, DlinkList.cpp,
Dlink.h, Dlink.cpp,
DLLMain.cpp.
```

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

Implementing the classes

// file DLink.h

```cpp
#include <iostream>
using namespace std;

class DLinkList;


  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,
```

```cpp
                // 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"
```

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

    friend class DLinkList; };
```

```cpp
DLink* const nil = 0; // the null DLink pointer
```

Notice
- all members are private,
- class DLinkList a friend,
- four friend operations,
- the null DLink pointer.

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

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

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

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

bool DLinkList::isEmpty() const
    { return  (firstLink==nil); }
```

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

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

void  DLinkList::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
```

```cpp
        { if (theLink==firstLink)
                // it is the first element
                firstLink=theLink->next;
            delete theLink; } }}


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

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

The member functions of `Dlink` are responsible for navigating forwards or backwards through the link structure, and are responsible for maintaining the forwars/backwards links when updating the link structure

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

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(); } }
```

```cpp
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::inserAfterThis(…),
  DLink::lastLink() and DLink::find(**int**)

- pointer structures

- "delegation" and object "collaboration"

- clever use of destructors

- recursive definitions

The Object Based Features in C++ - Summary

- classes

- data members

- member functions

- static members

- constructors, destructors

- this

- 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

The "Imperial 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 `Class`:

   4a) specify behaviour of each member function, especially for border cases

   4b) write a ~~static~~ `Class::testMe()` function which demonstrates behaviour of all member functions;

   4c) write member function bodies;

   4d) test using `Class::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 my be supplied by object.

Employee
salary: double
age: int
payrise(double)
birthday()

Assignment Project Exam Help

https://powcoder.com
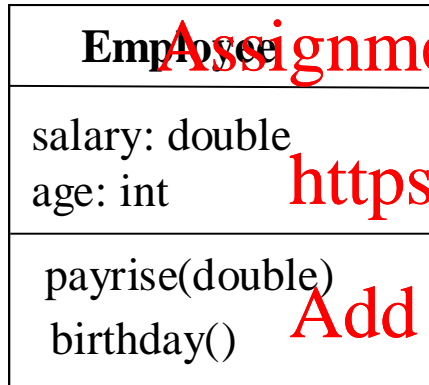
Add WeChat powcoder

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

attribute   -----> C++
operation   -----> C++

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.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Read on in the next part of the course: C++ the Object Oriented Paradigm