

Advanced C++

Lorenzo Natale (lorenzo.natale@iit.it)

September/October 2007, Italian Institute of Technology, Morego Genova

what is advanced?

Advanced C++

Lorenzo Natale (lorenzo.natale@iit.it)

September/October 2007, Italian Institute of Technology, Morego Genova

Schedule:

Tue 18/9 9:30-12:30

Thu 20/9 9:30-12:30

Tue 2/10 9:30-12:30

Thu 4/10 9:30-12:30

Program

- Introduction to Object Oriented Programming
- Classes
 - constructors, destructors
 - members, static and const members
 - operators, overload
- Inheritance, polymorphism
- Templates
- Overview of the Standard Template Library
- Maybe some advanced topics

Writing code is easy!

But...

- Does anybody write GOOD code?
- Programming languages are like spoken languages, it is not so difficult to make sentences, but it is very difficult to write well
- Even worse, the wrong assumption when writing code is that what is important is to make it work...

What is GOOD code?

- The whole point is to... avoid bugs
- Simplicity
- Readability
- Modularity/Separation → code reuse, be “smart lazy”
- Layering → e.g. GUIs
- Efficiency
- Elegance → code can be beautiful to look at
- Manage complexity, reduce bugs, improve flexibility, ease maintainability and facilitate team work

and finally...

GOOD code is a matter of taste...

Programming Paradigms

- Procedural programming:

“Decide which procedures you want; use the best algorithms you can find”

- focus on processing-algorithms

- usually supported by languages by providing facilities for creating functions, passing arguments to them and return results

- Example: `double sqrt(double v)`

sqrt.h

```
double sqrt(double v);
```

```
#include "sqrt.h"
```

```
int main()
```

```
{
```

```
    double root2=sqrt(2);
```

```
}
```

main.cpp

```
#include "sqrt.h"
```

```
double sqrt(double v)
```

```
{
```

```
    ... //code to compute sqrt...
```

```
}
```

sqrt.cpp

Programming Paradigms

- Modular programming, emphasis in the design of programs has shifted from design of procedures toward organization of data:

“Decide which modules you want; partition the program so that data is hidden within modules”

-Example: implementation of a Stack

Example: stack

Example: better stack

Programming Paradigms

- User-defined types (or abstract data type):

“Decide which types you want; provide a full set of operations for each type”

-Example: implementation of the Stack class

C++ stack

More on OOP, concepts:

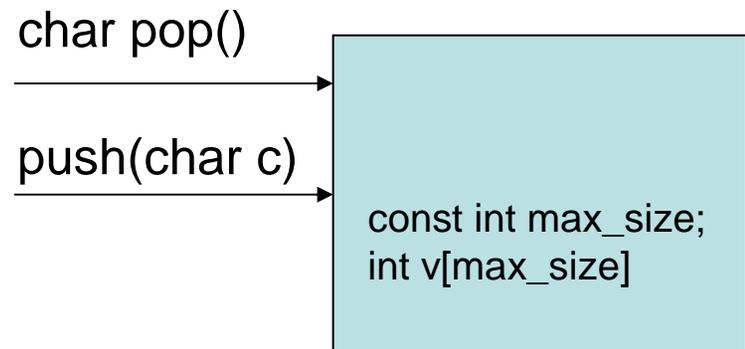
encapsulation

inheritance

polymorphism

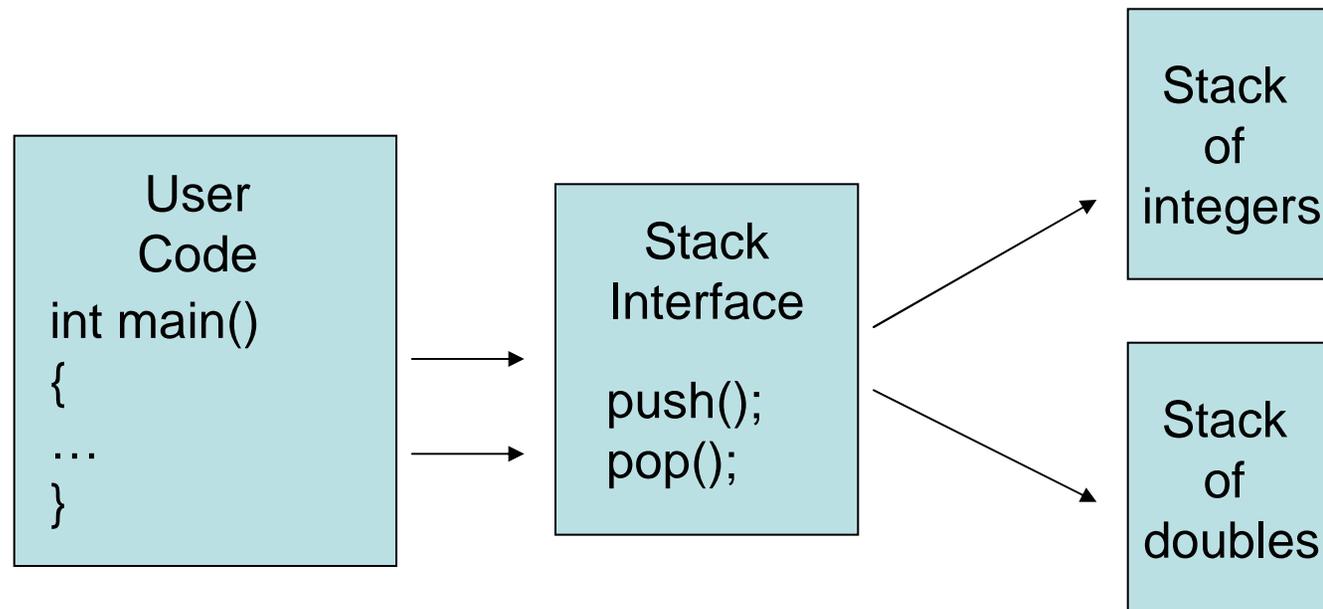
Encapsulation

- Put together data and code (methods) required to manipulate it → objects
- Objects are “black boxes”; users have access to an object only through a subset of methods and variables it declares as public (interface)

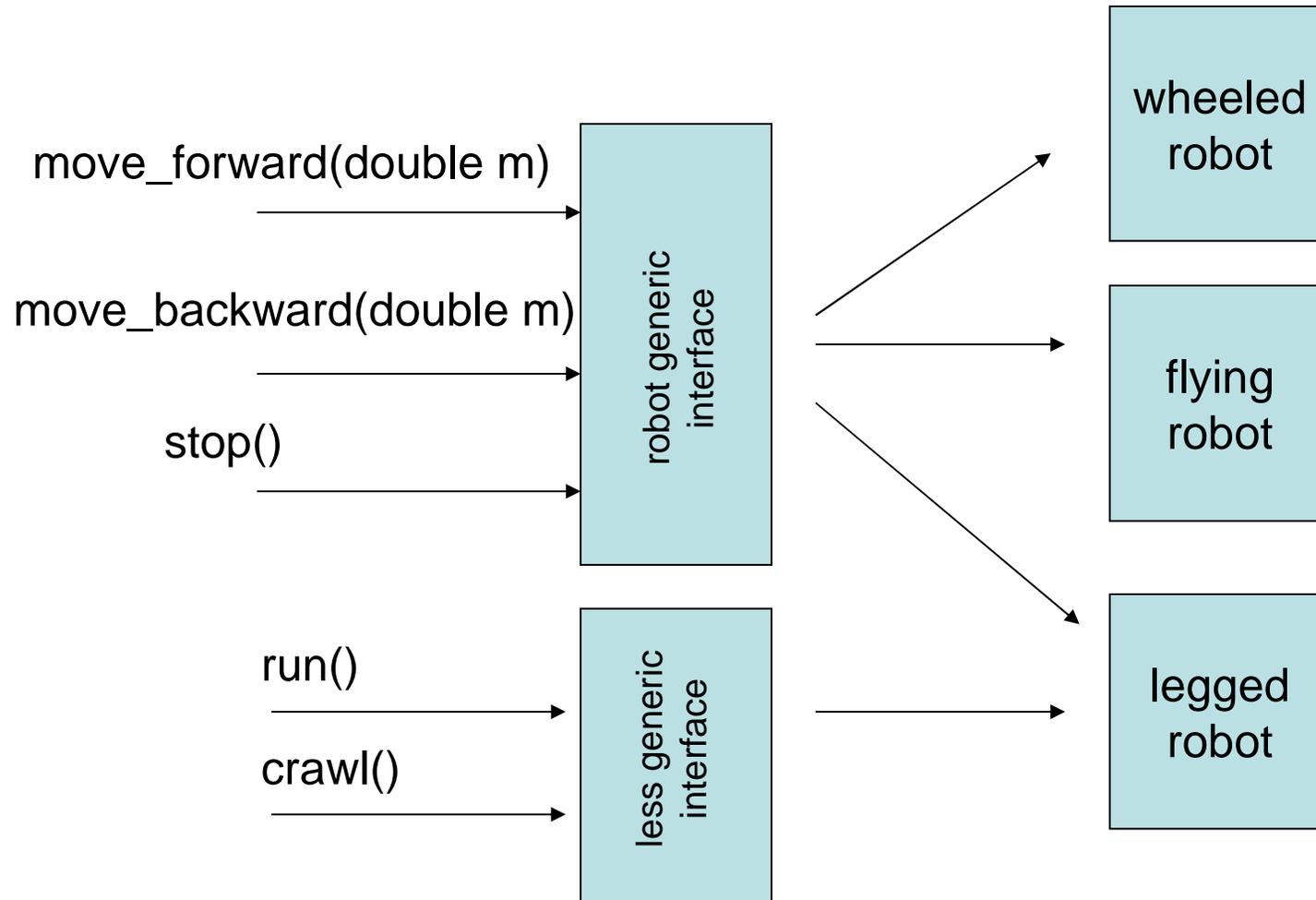


Polymorphism

- Extends the concept of *interface*
- Interfaces can be made separated entities
- Through polymorphism an interface provides access to different *implementations*

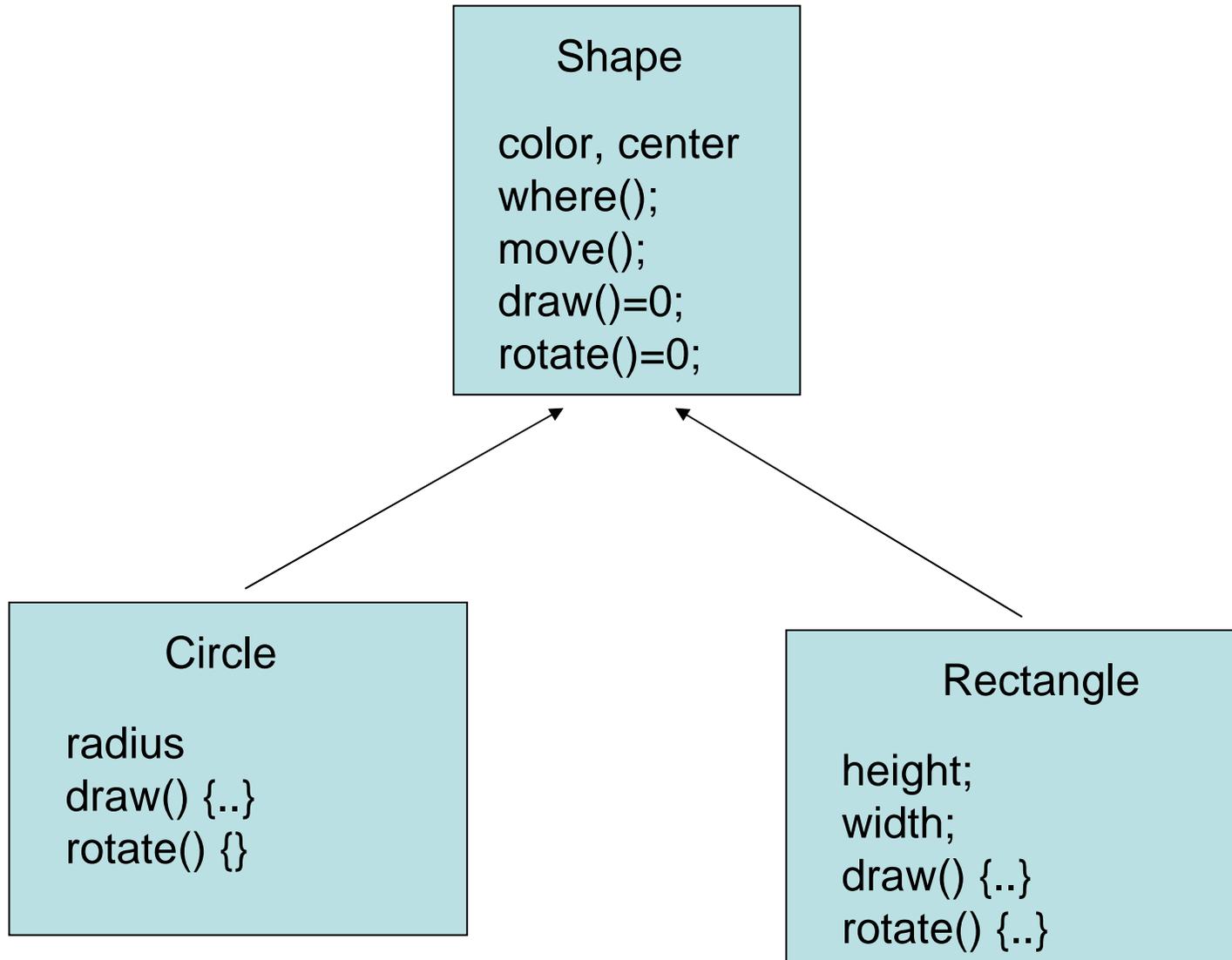


Polymorphism can be powerful



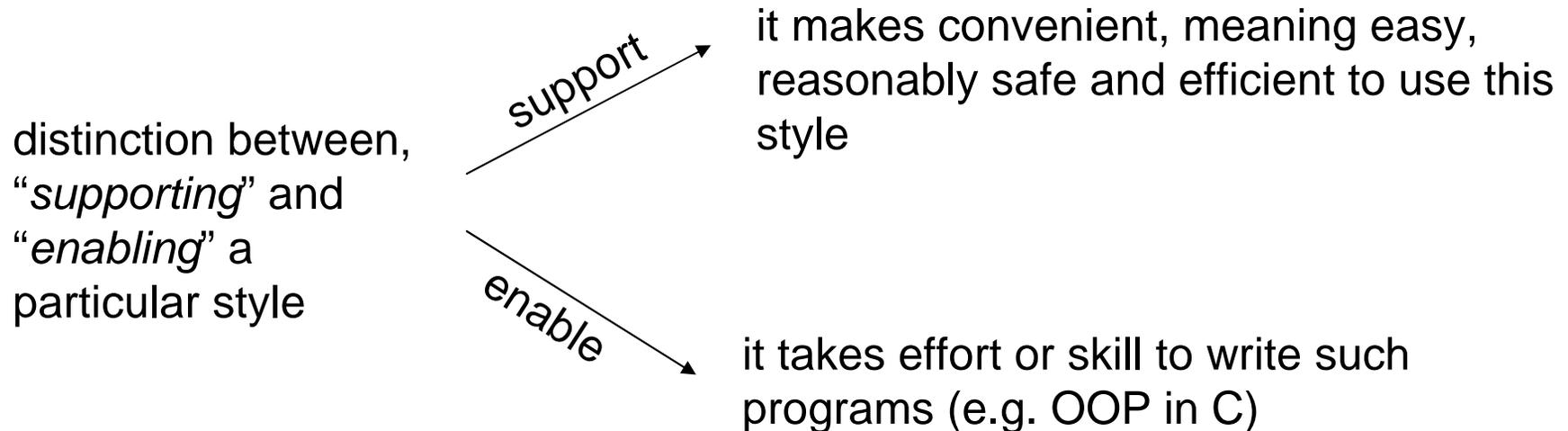
Inheritance

- Through *inheritance* an object acquires properties of another object
- Allows to organize information in a hierarchical way, by creating a hierarchy of classes and subclasses (parents and children)
- Subclasses need only define properties that make them unique within their “group”
- Other properties are acquired (derived) through inheritance



“Decide which classes you want; provide a full set of operations for class; make commonality explicit by using inheritance”

C++ *supports* Object Oriented Programming



Also C++ provides compile-time/run-time checks for type checking, ambiguity detection and additional OO libraries

C++, Historical Notes

- Extension of C, originally developed by Bjarne Stroustrup in 1979
- Borrows notion of class from Simula67 (Dahl 1970) and operators from Algol68 (Woodward 1974)
- Initially called C, with classes
- Became C++ in 1983
- First revision in 1985
- Started Standardization by ANSI/ISO, in 1990
- First draft in 1995, ratified in 1998
- Minor revisions in 2003

Classes

- A class is a user-defined type
- It groups data and functions to manipulate it
- It is similar to a *struct*, actually in C++ a struct is a kind of class
- Example: define the concept of *Date*

```
struct Date {  
    int day, month, year;  
    void init_date(int d, int m, int y); //initialize  
    void add_year(int n); //add n years...  
    void add_month(int n);  
    void add_day(int n);  
};
```

```
struct Date {
    int day, month, year;
    void init(int d, int m, int y); //initialize
    void add_year(int n); //add n years...
    void add_month(int n);
    void add_day(int n);
};
```

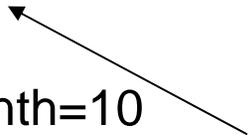
```
int main()
{
    Date today;
    today.init(18,9,2007)
;

    Date tomorrow;
    tomorrow=today;
    tomorrow.add_day(1

);

    tomorrow.month=10
0;
1
```

```
void Date::init(int dd, int mm, int yy)
{
    day=dd;
    month=mm;
    year=yy;
}
```



violation! can we prevent this?

Access Control

- The previous declaration of Date does not protect the representation of Date (e.g. d, g, and y) from external access
- Use *class*

```
class Date {  
    int d, m, y;  
  
public:  
    void init_date(int dd, int mm, int yy); //initialize  
    void add_year(int n); //add n years...  
    void add_month(int n);  
    void add_day(int n);  
};
```

private part of Date

public part of Date

- Access to *private* members (be it data or functions) is allowed only from members of the class

```
int main()
{
    Date today;
    today.init(18,9,2007);

    Date tomorrow;
    tomorrow=today;
    tomorrow.add_day(1);

    tomorrow.m=100; // error cannot access private member
}
```

- Safe, access to data is forced through the public interface (prevent writing 100 in month)
- Other details are hidden to the external world and can be changed

- More in general:

```
class class_name {  
  data and private functions  
  public:  
  data and public functions  
  private:  
  data and public functions  
  ...  
  ...  
};
```

- members can be *public*, *private* and *protected*, this affects inheritance rules...

Constructors

- The use of the function *init* is tedious and error-prone, a programmer could forget to initialize an object with *undefined* results
- We can “force” initialization by declaring a function with the explicit purpose of initializing the object (*constructor*)
- A constructor has the same name of the class and *does not return any value (not even void)*

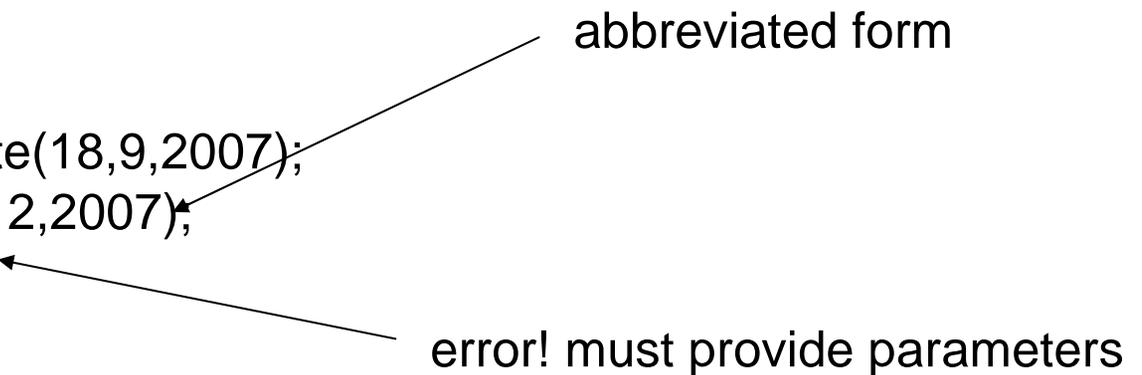
```
class Date {  
public:  
    Date(int d, int m, int y);    // constructor  
};  
Date::Date(int dd, int mm, int yy)  
{  
    day=dd;  
    month=mm;  
    year=yy;  
}
```

- A constructor is called whenever a object of a class is created
- If the constructor requires arguments, these arguments *must* be supplied

```
int main()
{
    Date today=Date(18,9,2007);
    Date xmas(25,12,2007);
    Date tomorrow;
}
```

abbreviated form

error! must provide parameters



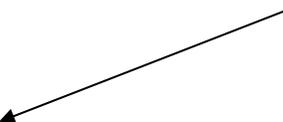
Overload

- It is useful (and typical) to provide several ways of initializing a class object

```
class Date {  
public:  
    Date(int d, int m, int y);    // constructor  
    Date(int d, int m);          // day, month, today's year  
    Date(int d);                 // today's month and year  
    Date();                      // default date, today  
    Date(const char *date);      // from a string representation  
};
```

```
int main()  
{  
    Date today=Date(18);  
    Date xmas("Dec 25, 2007");  
    Date now;  
}
```

the compiler decides
which constructor to call (from
the parameters)



Overload: the same principle applied to functions

```
struct complex {
    double Re;
    double Im;
};

// compute the square of a scalar (double)
double square(double a) {
    return a*a;
}

// compute the square of a complex
complex square(complex a) {
    complex tmp;
    tmp.Re = a.Re*a.Re - a.Im*a.Im;
    tmp.Im = 2*a.Re*a.Im;
    return tmp;
}
```

..but

- Overloaded functions must have different parameters
- It is not possible to have functions with same parameters but different return type

```
double myFunction();  
int myFunction();
```

error!



Constructors with one parameter

If an object has a constructor with a single parameter

```
class Number {  
    int a;  
public:  
    Number (int j)  
    {a=j;}  
  
    int geta()  
    {return a;}  
};
```

```
int main()  
{  
    Number n=10; // eq: n=Number(10);  
}
```

Default parameters

- Constructor overload might lead to proliferations of functions
- The use of default parameters can alleviate this problem

```
class Date {
public:
    Date(int d=0, int m=0, int y=0);    // constructor
    ...
};

Date::Date(int d, int m, int y)
{
    if (d!=0)
        day=d;
    else
        day=...; //get today's day and write it
    ....
}
```

- Of course in this case we are lucky because 0,0,0 are not valid numbers for day, month and year, so we do not clash with possible values

Consider this other example:

```
class Rectangle
{
    double a,b;
public:
    Rectangle(double i=0, double j=0);           // default's parameters
    double area() {
        return a*b;
    }
};

Rectangle::Rectangle (int i, int j) {
    a = i;
    b = j;
}
```

```
Rectangle a;           // equivalent to Rectangle a(0,0)
Rectangle b(2);       // equivalent to Rectangle b(2,0);
Rectangle c(2,2);
double ab= b.area(); // ab is 0
double ac = c.area(); // ac is 4
```

Notes on default parameters

- Limitations:
 - Default parameters must be at the end of the list
 - There must be no ambiguity with other overloaded versions of the same function (e.g. Rectangle::Rectangle())
- Default parameters go in the declaration
- Obviously not limited to constructors, e.g.:

```
void printError(const char *string = 0);
```

```
void printError(const char *string)
{
    if (string == 0)
        printf("Error\n");
    else
        printf("Error: %s\n", string);
}
```

Array of objects

- It is possible to create arrays of objects
- Useful to manage sets of variables of the same type

Example:

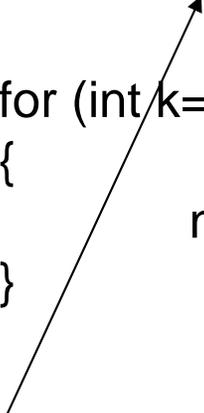
```
class Number {  
    int i;  
public:  
    void set(int j) {i=j;}  
    int get() {return i;}  
};
```

```
int main()  
{  
    Number ns[9];  
  
    for (int k=0; k<9;k++)  
    {  
        ns[k].set(10);  
    }  
}
```

What if the constructor requires a parameter?

```
class Number {
    int i;
public:
    Number(int k)
    {
        i=k;
    }
    void set(int j) {i=j;}
    int get() {return i;}
};
```

```
int main()
{
    Number ns[9];
    for (int k=0; k<9;k++)
    {
        ns[k].set(10);
    }
}
```



error, constructor requires parameter

```
int main()
{
    Number ns[9]={1,2,3, ...};

    for (int k=0; k<9;k++)
    {
        ns[k].set(10);
    }
}
```

Pointers to Objects

- As for other variables, it is possible to define pointers to objects

```
int main()
{
    Number a(10);
    Number *p;

    p=&a;

    p->set(11); //now a contains 11
    printf("%d", p->get()); //prints 11
}
```

- Arithmetic of pointers works for pointers to objects

```
int main()
{
    Number ns[9]={1,2,3, ...};
    Number *p;
    p=ns; //point to beginning of array

    for (int k=0; k<9;k++)
    {
        p->set(10);
        p++;
    }
}
```

- Members are like any other variables, so we can get a pointer to them

```
int main()
{
    Number ns;
    int *p=&ns.i;    //i must be public

    *p=0; //writes 0 in ns.i
}
```

- Must have good reasons to do this, violates data hiding/encapsulation principle

Destructor

- A class's *destructor* is a function that is executed when the object is destroyed (it goes out of scope)
- It has the same name as the class prefixed by ~
- It does not return any value (not even void)

```
class Employee {
    const int SIZE=80;
    char *name;           // private
    double salary;       // private
public:
    Employee ();
    ~ Employee ();
    void setName(char *n);
    void setSalary(double s)
        {salary=s};
}
```

```
Employee :: Employee(){
    name = new char [SIZE];
}
Employee::~Employee() {
    delete [] name;
}
```

```
void Employee::setName(const char *n)
{
```

```
    strncpy(name,n, SIZE)
    name[SIZE-1]='\0';
```

```
int main()
{
```

```
    Employee foo;
```

```
{
```

```
    Employee bar;
```

```
}
```

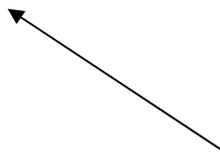
```
    Employee baz;
    f();
```

```
}
```

```
int f()
{
}
}
```

```
Employee bop;
```

{ } here define a new scope



- To recap:
 - destructors and constructors are invoked when objects instantiated and destroyed (the object go out of scope or a delete is called)
 - destructors follow order inverse w.r.t. the order of constructors
- Important: it is not possible to know the order of construction of global objects declared in different files

Static

- In C a static variable is initialized only the first time it is used

```
void loop()
{
    static int times=0;
    times++;
    printf("Called %d times\n", times);
}
```

```
int main()
{
    for(int k=0; k<100;k++)
    {
        loop();
    }
}
```

- In C++ this is still true; in addition members of a class can be static, this applies to both data and functions
- When a variable is static within a class, there exists only once copy shared among all objects of the same class

```
class Shared {  
static int a;
```

public:

```
void set(int i) {a=i};  
}
```

Shared::a exists before X,Y

```
int Shared::a=0; //definition of "a" (only once)
```

```
int main()  
{
```

```
Shared X, Y;
```

X.a is 1

```
X.set(1);
```

Ya is 2, but also X.a is 2

```
Y.set(2)
```

```
}
```

- Static function can only access static variables
- Useful to initialize static variables

```
class Shared {  
    static int a;  
  
public:  
    static void init(int k){ a=j;}  
}
```

```
int Shared::a=0; //definition of "a" (only once)
```

```
int main()  
{  
    Shared::a=99;  
    Shared::init(99);  
    Shared X, Y;  
    X.set(1);  
    Y.set(2)  
}
```

error, a is private

ok, init is public, notice that at this point there are no instances of Shared, but we can have access to a through Shared::init()

to tell the compiler we want to use the init() defined in Shared we used Shared::init()

- Why static members?
- They are useful to avoid global variables
- Counting the number of instances of a given class
- Sharing resources that are “unique”

Example 1: counter

```
class Counter {
public:
    static int count;
    Counter() {count++};
    ~Counter() {count--};
    void show()
    {
        printf("Counter%d", count);
    }
};
int Counter::count=0;

int main()
{
    Counter c1;
    c1.show();
    f();
    Counter c2;
    c2.show();
}
```

```
void f()
{
    Counter tmp;
    tmp.show();
}
```

Example 2: RobotShared

This

- When a member function is executed, a pointer to the “calling” object is automatically passed to it
- This pointer is called “this”
- In fact “this” is the mechanism used by C++ to implement member functions; member functions are like normal functions, that receive a “hidden” parameter, a pointer to the calling object

```
class Complex
{
    double re;
    double im;
public:
    Complex (double r, double i)
    {
        re=r;
        im=i;
    }
    double getRe()
    { return re; }
};
```

```
class Complex
{
    double re;
    double im;
public:
    Complex (double r, double i)
    {
        this->re=r;
        this->im=i;
    }
    double getRe()
    { return this->re; }
};
```

Operators

- Operators in C++ are like other functions and can be overloaded
- This is powerful, it allows to have user-defined types behave like built-in ones

```
class Complex {  
    double re;  
    double im;  
public:  
    Complex (double r = 0.0, double i = 0.0) ;  
};
```

```
main()  
{  
    Complex C;  
    Complex A(0,4);  
    Complex B(-1);  
    C = A - B; // ??  
}
```

Operators as member functions

- General form:
return_type class_name::operator#(parameters)
{...}

- Example “operator -”

```
class Complex {  
    double re;  
    double im;  
  
public:  
    Complex (double r = 0.0, double i = 0.0) ;  
    Complex operator- (Complex b);  
};
```

“-” is equivalent to a function called operator -

```
Complex Complex::operator-(Complex b)  
{  
    Complex temp;  
    temp.re = re - b.re;           // sum real part  
    temp.im = im - b.im;         // sum img part  
    return temp;  
}
```

return object, useful to assign the object C=A-B

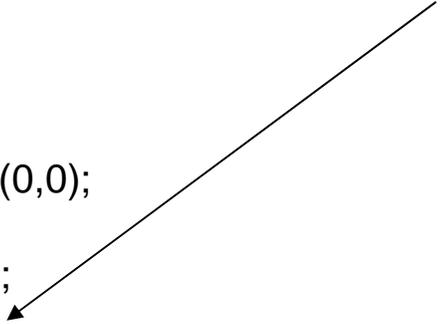
- Another example, operator '++'

```

Complex Complex::operator++ ()
{
    re++;
    im++;
    return *this;           // return itself
}

int main()
{
    Complex A(0,0);
    ++A;
    Complex C;
    C=++A;
}

```

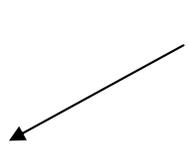


- What about A++ ?

```

Complex Complex::operator++ (int x)
{
    Complex tmp=*this;     //save for later
    re++;
    im++;
    return tmp;           // return prev copy
}

```



dummy int, to be ignored (always 0)

- More operators, abbreviated form:

```
Complex Complex::operator+= (Complex b)
{
    re=re+b.re
    im=im+b.im;
    return *this;
}
```

- Copy assignment operator:

```
Complex Complex::operator= (Complex b)
{
    re=b.re
    im=b.im;
    return *this;
}
```

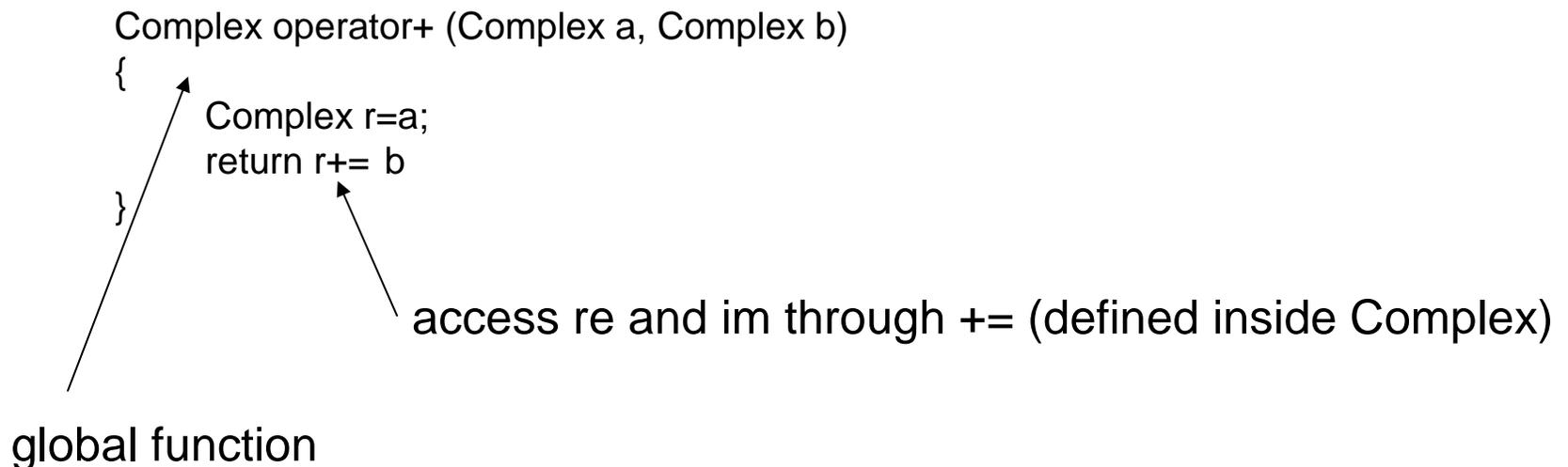
allows concatenation A=B=C



- An assignment operator like this is generated for free by the compiler (bitwise copy), but in some cases this is not enough...

Nonmember Operators

- It is better to minimize the number of functions that directly manipulate the representation of an object
- Operators can also be define outside of a class



drawback: less efficient in general, cannot access to members
(here Complex::re and Complex::im)

Mixed-mode arithmetic

- Nonmember operators are more flexible

Complex d=2+b ?

2 here is an integer, cannot overload operator +

```
Complex Complex::operator+= (double a)
{
    re+=a;
    return *this;
}
```

← += member operator

```
Complex operator+ (double a, Complex b)
{
    Complex tmp=b;
    return b+=a;
}
```

← + nonmember operator

Overload of operators: limitations

- cannot change the order of precedence of operators
- cannot change the number of parameters required (of course we can ignore some of them)
- cannot have default parameters
- we can overload most of the operators “+”, “-“, “++”, “—“, “*“, “->“, “>“, “<“, “==“, etc..., but *not* “.” “..” “.*” and “?”
- common sense: don't change the meaning of the operator (i.e. don't write operator + so that it does a subtraction)

Copying, passing objects to functions

```
class Vector
{
    double *buffer;
    int length;
public:
    Vector(int l=0, double v=0.0);
    ~Vector();
    double get(int k);
    int getLength();
}
```

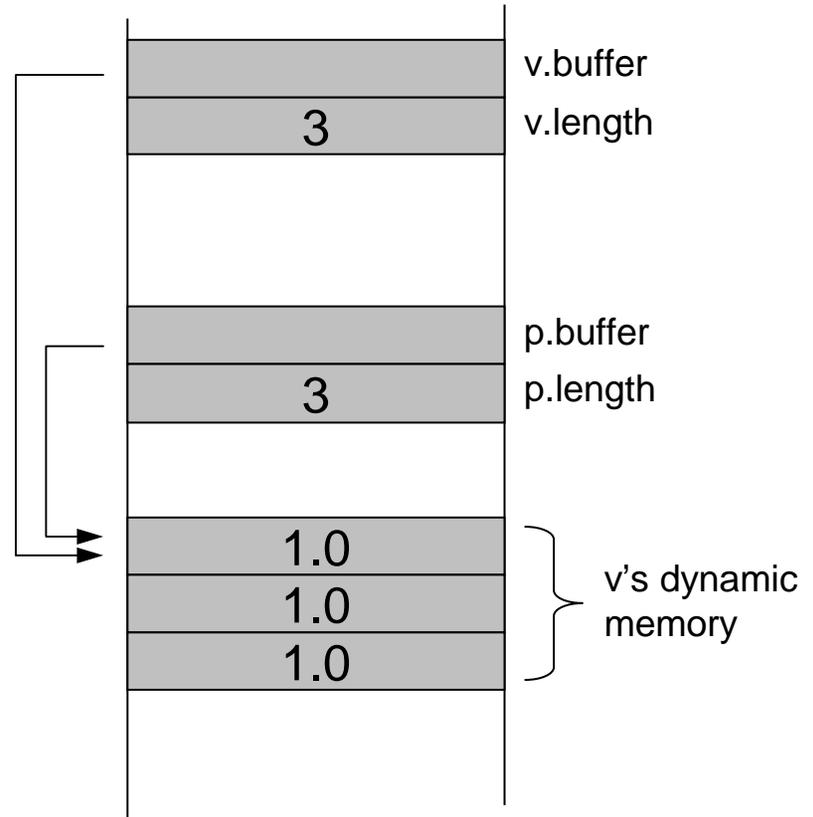
```
Vector::Vector(int l=0, double v=0.0):
buffer(0),
length(0)
{
    if (l>0) {
        length=l;
        buffer=new double [l];
        for(int k=0;k<l;k++)
            buffer[k]=v;
    }
}
```

```
Vector::~~Vector()
{
    if (buffer!=0)
        delete [] buffer;
}
```

```
void printVector(Vector p)
{
    for(int k=0; k<p.getLength(); k++)
        printf("%lf\n", p.get(k));
}
```

```
int main()
{
    vector v(3, 1.0)
    printVector(v);
}
```

disaster!



- Solution 1, use pointers

```
void printVector(Vector *p)
{
    for(int k=0; k<p->getLength(); k++)
        printf("%lf\n", p->get(k));
}
```

only the pointer is duplicated, ~Vector() is not invoked at the end of the function

```
int main()
{
    vector v(3, 1.0)
    printVector(&v);
}
```

Drawbacks:

- printVector can now modify v through p
- tedious, we have to use & and ->
- difficult to overload operators for pointers

- Solution 2, C++ allows to use reference to pass objects

```
void printVector(Vector &p)
{
    for(int k=0; k<p.getLength(); k++)
        printf("%lf\n", p.get(k));
}
```

```
int main()
{
    vector v(3, 1.0)
    printVector(v);
}
```

we say v is passed *by reference* (as opposed to *by value*)

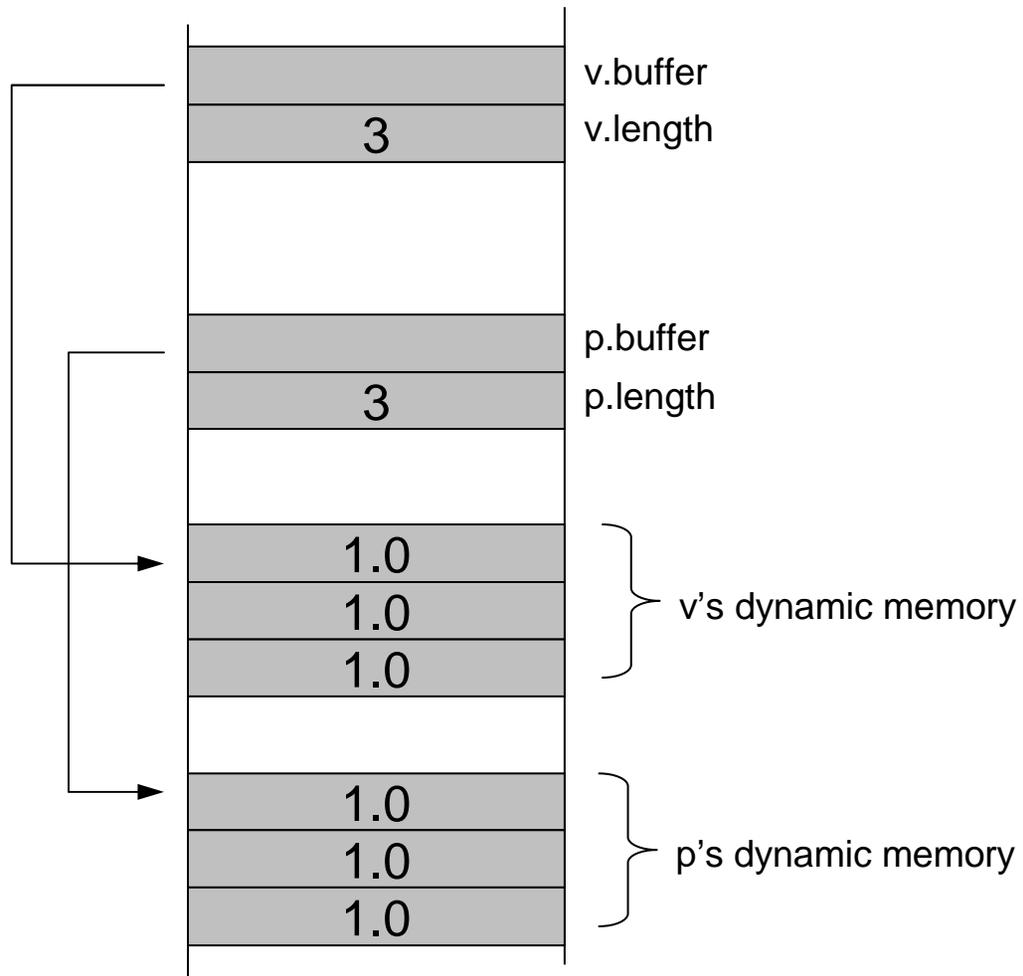
- To prevent a function from modifying the object it is possible to use the *const* modifier

```
void printVector(const Vector &p)
{ ... }
```

- **Solution 3, explicit *copy constructor***

We have this problem because the compiler does not know how to copy an object defined by the user; in lack of better information the compiler performs a bitwise copy. We can instruct the compiler so that it knows how to do this operation properly. We do this by writing a *copy constructor*, a constructor which takes as a parameter a reference of the class itself:

```
Vector::Vector(const Vector &v):
buffer=0,
length=0
{
    length=v.length;
    if (length!=0)
    {
        buffer=new double [length];
        for(int k=0;k<length;k++)
            buffer[k]=v.buffer[k];
    }
}
```

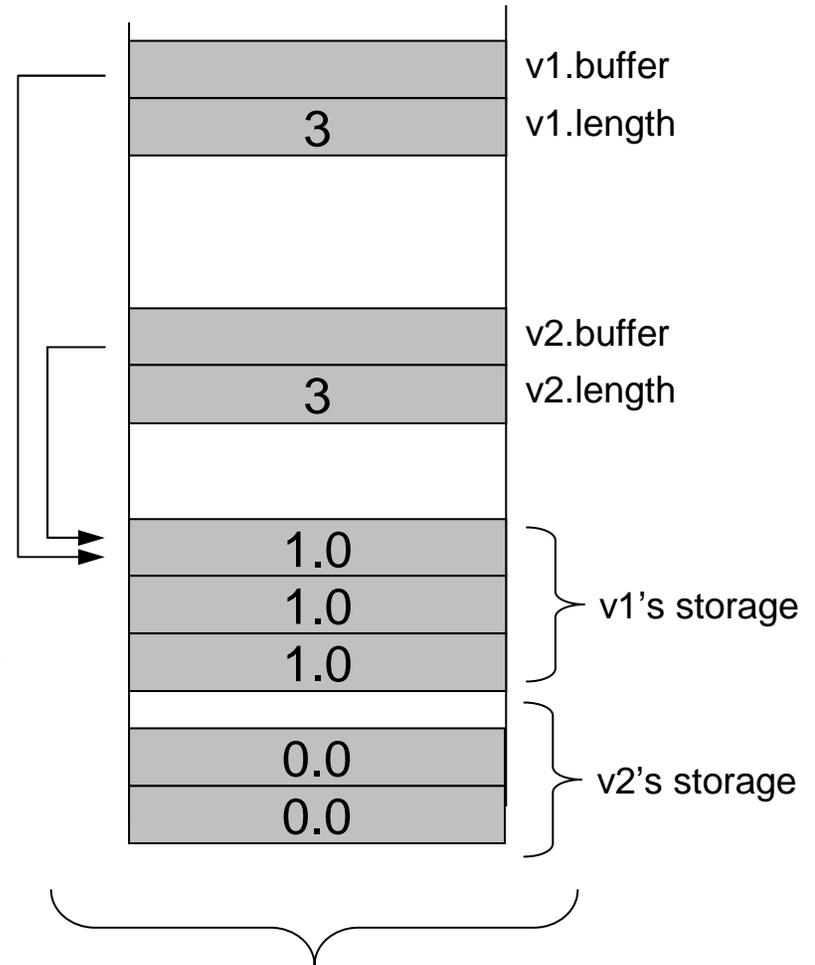


- A similar problem occurs when we copy objects, in assignment operations:

```
int main()
{
    Vector v1(3,1.1);
    Vector v2(2,0.0);

    v2=v1;
}
```

invoked v1 and v2's destructors
v1's storage is deallocated twice
v2's storage is lost



situation after copy v2=v1

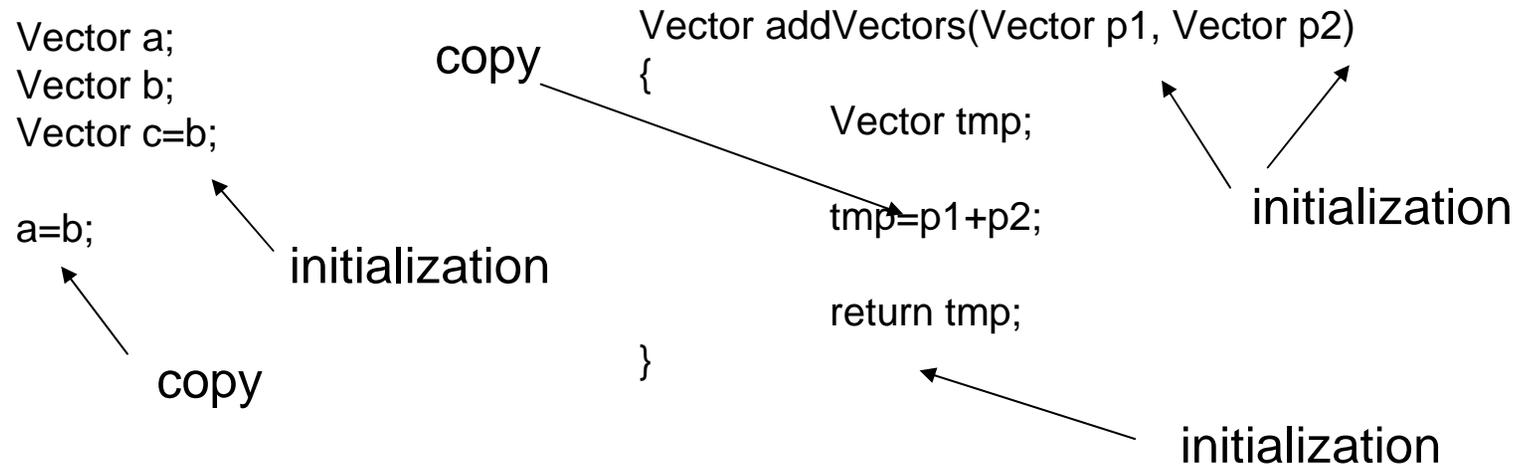
- Solution, overload *copy assignment operator*

```
Vector & Vector::operator=(const Vector &v)
{
    if (buffer!=0)
        delete [] buffer; ← deallocate first

    length=v.length;
    if (length!=0)
    {
        buffer=new double [length];
        for(int k=0;k<length;k++)
            buffer[k]=v.buffer[k];
    }
    return *this;
}
```

Final note on object duplication

- Troubles occur when we try to duplicate objects, this happens in two different cases:
 - object initialization
 - copy
- These cases look similar but are indeed quite different and have different solutions
- In the first case we are using an object to *create* a *new* one; in fact to solve this we need to declare a *copy constructor*
- In the second case, we are filling the content of an *existing object* with the content of another one, this is solved by writing a *assignment operator*



- This is often less than ideal, because it avoids a considerable overhead to the code; for this reason it is always preferable to pass arguments by *reference*
- The same overhead might occur in case of *large* objects

```
class Vector
{
    const int length=1000;
    double buffer[length];
public:
    Vector(double v=0.0);
    ~Vector();
    double get(int k);
    int getLength();
}

void printVector(Vector p)
{
    double a=v.get(0);
}

void printVector(const Vector &p)
{
    double a=v.get(0);
}
```

fine to copy this object, but costly

much better!

- Is pass-by-reference always less expensive?
- References are typically implemented as pointers
- So for built-in types, it might be more efficient to use pass-by-value
- What about small objects? They have inexpensive copy constructors, so we might think it is ok to pass them by value. However:
 - They are somehow treated differently by the compilers
 - An object might be small now, but could grow in the future
 - In different impl of STL, for example objects have different size

Better way to overload operators

Let's see how we should write our operators:

Members:

```
Complex &Complex::operator++ ()  
{...}
```

```
Complex &Complex::operator+= (double a)  
{...}
```

```
Complex Complex::operator++ (int x)  
{...}
```

General rule:
Pass as *const reference*, return
references to **this* and *copies* of
temporary objects

Nonmembers:

```
Complex operator+ (double a, const Complex &b)  
{...}
```

```
Complex operator+ (const Complex &a, const Complex &b)  
{...}
```

etc...

Returning references

- In some cases it is useful to return references to objects
- This allows writing functions that directly modify the content of an object

```
class Complex
{
private:
    double re;
    double im;

public:
    Complex(double r=0, double i=0) {
        re=r;
        im=i;
    }

    double &real()
    { return re; }

    double &imag()
    { return im; }
};
```

```
int main()
{
```

```
    Complex a;
    a.real()=10;
    a.imag()=10;
```

ok, copies r.re into a

```
    double b=a.real();
}
```

we get access to a.re and a.im, not to their copies

returning a reference

Returning references: pitfall

- Be careful not to return a reference to a temporary object!

```
double &Complex::real()
{
    double tmp=re;
    return tmp;
}
```

we are returning a reference to an object that is going to be destroyed after the invocation of `Complex::real()`

```
Complex a(1,1);
a.real()=10;

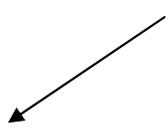
double t=a.real();
```

undefined behavior!

This is not what we want to do (we are not returning 're'), and dangerous (and wrong) !

- More realistic example:

wrong!

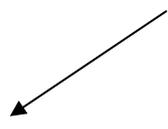


```
Complex &operator+ (const Complex &a, const Complex &b)
{
    Complex tmp=a;
    tmp +=b;
    return tmp;
}
```

```
Complex a(1,1);
Complex b(2,2);
```

```
Complex c=a+b; // c is now undefined...
```

not much we can do to avoid this, but at least it is correct...



```
Complex operator+ (const Complex &a, const Complex &b)
```

The Curse of const

- What happens when we have a const reference?
- The compiler does its best to check that this object is not modified
- How?

```
class Rational
{
public:
  int n;
  int d;

public:
  Rational(int nn=0,int nd=0);

  int &num()
  { return n; }

  int &den()
  { return d; }
};
```

public, for now

```
void myFunc(const Rational &b)
{
  int c=b.n;    // this is ok
  b.n=10;      // error, l-value specifies const object
  b.num()=10
  int d=b.num();
}
```

error cannot convert this pointer from const Rational & to Rational &

- Returning *const references* guarantees that nobody will be able to modify these references
- Member functions that do not modify an object, can (and must) be declared as *const*; this allows invoking them from const objects

```
class Rational
{
public:
    Rational(int nn=0,int nd=0);

    const int &num() const
    { return re; }

    const int &den() const
    { return im; }
};
```

```
void myFunc(const Rational &n)
{
    int num=n.num();
    int den=n.den();
    ....
}

ok, members are now const
```



```
int main()
{
    Rational a;
    myFunc(a);
    a.num()=10;
}
```

```
Rational a;
myFunc(a);
a.num()=10;
```

to be able to do this, we need to overload num() and den() with non const members (it is possible)



- All this might seem an overkill, but it is the price to pay to have the compiler check that the code is consistent
- Also, the compiler can generate more efficient/optimized code when dealing with const references
- If we are desperate we can force the compiler to do what we want:

```
Complex &a=const_cast<Complex &>(b);  
myFunc(b); //error! b is constant  
myFunc(a); //ok a is not const
```

```
void myFunc(Complex &b)  
{  
    ....  
}
```

“cast away const”

ok, it is possible to call non-const members of a

Mutable and *Logical Constness*

- In same (rare!) cases it is correct to change the state of a constant object
- This is sometimes referred to as *logical constness*, or when you can still think as the object being constant even if some of its internal variables have somehow changed
- In this cases it is possible to declare some of the member variables as “mutable”; mutable variables can be changed in a const object
- Here is an example:

```
class Complex
{
private:
    double re;
    double img;
    mutable int access_count;

public:
    Complex(double r=0, double i=0);
    ...
}
```

```
const double &Complex::getReal() const
{
    access_count++;
    return re;
}
```

this is ok



Example: a vector class

Inheritance

- Through inheritance an object acquires members of another object
- Subclasses “extend” their parent by defining members that make them unique

```
class derived_class: access base_class  
{  
    //body  
};
```

access can be:

public, private or protected

Public inheritance

- Variables and functions of the parent that are *public* or *protected* remain such
- the subclass can access *public* and *protected* members but not the ones that are *private*

Example:

```
class Building {
    int rooms;
    int floors;
    int area;
    ...
public:
    void setRooms(int n);
    int getRooms() const;
    void setFloors(int n);
    int getFloors() const;
    void setArea(int num);
    int getArea() const;
}
```

```
class House: public Building {
    int bedrooms;
    int baths;
public:
    void setBedrooms(int n);
    int getBedrooms() const;
}
```

```
class School: public Building {
    int classrooms;
    int offices;
public:
    void setClassrooms(int n);
    int getClassrooms() const;
}
```

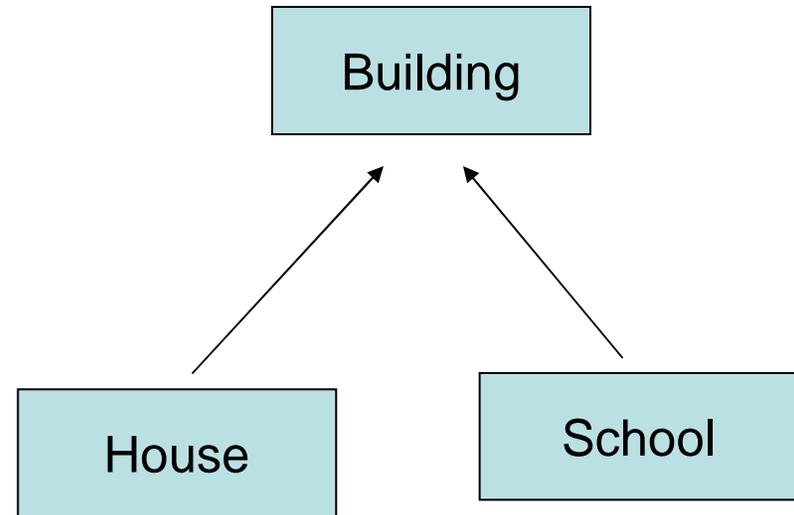
```
int main()
{
    House h;
    School s;

    h.setRooms(10);
    h.setFloors(3);
    h.setArea(5000);
    h.setBedrooms(5);
    h.setBaths(3);

    printf("House area: ", h.getArea());

    s.setRooms(200);
    s.setClassrooms(180);
    ...

    printf("School area:%d\n ",s.getArea());
}
```



- More interestingly, inheritance allows to group objects given their commonalities and treat them accordingly

```

int main() {
    House h;
    School s1;
    School s2;
    Building b1;
    ...
    Building *buildings[4];
    printf("House area: ", h.getArea());

    buildings[0]=&h;
    buildings[1]=&s1;
    buildings[2]=&s2;
    buildings[3]=&b1;

    for(int k=0;k<3;k++) {
        buildings[k]->setArea(10);
        buildings[k]->setFloors(5);
        //buildings[k]->setBaths(); //this would be an error
        printBase(buildings[k]);
    }
}

void printArea(const Building *b)
{
    printf("Area is: %d\n", b->getArea());
}

```

- House(s) and School(s) are subtypes of Building and can be used whenever a Building is acceptable (the opposite is not true)

Private members and inheritance

- Private members are hidden in derived classes

```
class Base
```

```
{
```

```
    private:
```

```
        int a;
```

```
    public:
```

```
        int b;
```

```
}
```

```
class Derived: public Base
```

```
{
```

```
    public:
```

```
        void foo(){ a=0;}
```

```
        void bar(){ b=0;}
```

```
}
```

error, a is hidden in Derived

error, a is private

```
int main()
```

```
{
```

```
    Derived d;
```

```
    d.a;
```

```
    d.b=10;
```

```
}
```

ok, b is public

Protected members and inheritance

- Protected members are hidden from outside, but can be accessed in derived classes

```
class Base {  
    protected:  
        int a;  
};
```

ok, a is protected in Base

```
class Derived1: public Base {  
    public:  
        void foo(){ a=0;}  
};  
  
class Derived2: public Derived1 {  
    public:  
        void bar(){ a=0;}  
};
```

```
int main()  
{  
    Derived1 d1;  
    Derived2 d2;  
  
    d1.a; ← error, a is protected  
    d2.a; ←  
}
```

Private inheritance

- All variables and functions of the parent are inherited as *private*

```
class Base {  
    public:  
        int a;  
};
```

```
class Derived1: private Base {  
    public:  
        void foo(){ a=0;}  
};
```

```
class Derived2: private Derived1 {  
    public:  
        void bar(){ a=0;}  
};
```

ok, a is private in Derived1

error!

```
int main()  
{
```

```
    Derived1 d1;  
    Derived2 d2;
```

```
    d1.a;  
    d2.a;
```

error, a is private

```
}
```

Protected inheritance

- All *public* (and *protected*) members of the parent become *protected*

```
class Base {  
    public:  
        int a;  
};
```

```
class Derived1: protected Base {  
    public:  
        void foo(){ a=0;}  
};
```

```
class Derived2: protected Derived1 {  
    public:  
        void bar(){ a=0;}  
};
```

ok, a is protected

```
int main()  
{  
    Derived1 d1;  
    Derived2 d2;
```

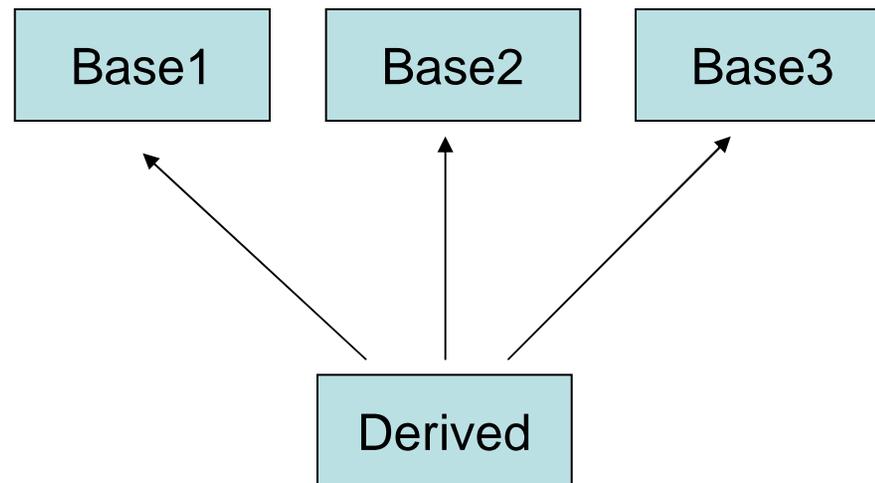
```
    d1.a;  
    d2.a;  
}
```

error, a is protected

Multiple Inheritance

- In C++ a class can inherit from more than one class

```
class Derived: public Base1, Base2, Base3 {  
    //class body  
}
```



Constructors, destructors and inheritance

- Some derived classes need constructors, if a base class has constructors then a constructor must be called
- Default constructors are invoked implicitly
- However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called:

```
class Building {
public:
    Building(int fl; int a);
}

class House: public Building {
    int bedrooms;
    int baths;

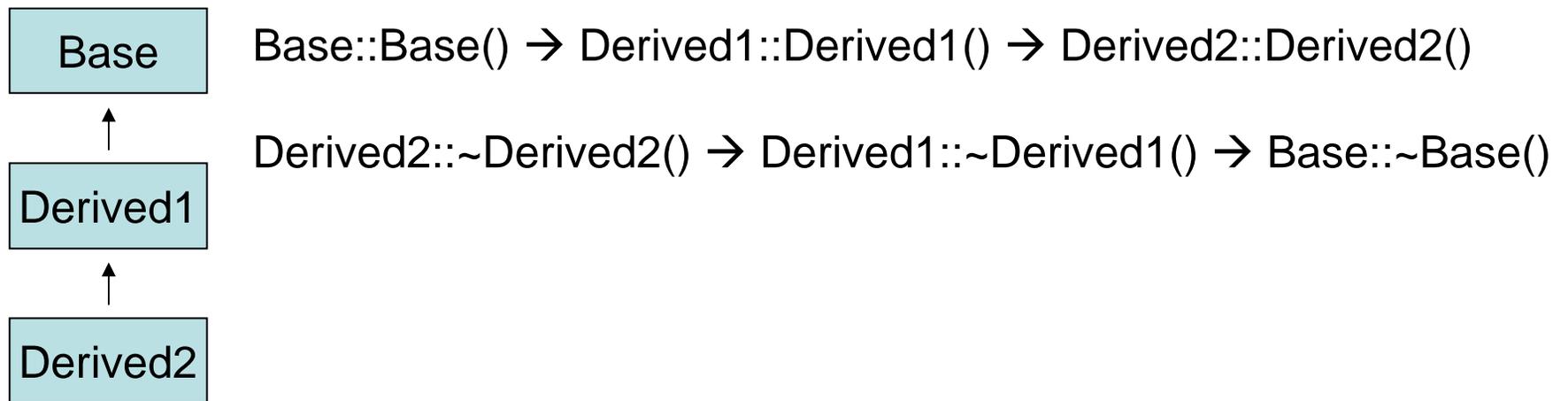
public:
    House(int fl, int a, brs, int bths );
    House(int fl, int a);
};

House::House(int fl, int a, int brs, int bths)
    :Building(fl, a)
{
    bedrooms=brs;
    baths=bths;
}

House::House(int fl, int a)
    :Building(fl, a),
    bedrooms(0),
    baths(0)
{
}
```

member initialization

- What is the order of constructors and destructors in derived classes?
- First is invoked the constructor of the base class and then the constructor of the derived class is called
- Destructors are invoked in reverse order



- In case of multiple inheritance, constructors follow the order of declaration (from left to right)

Note on inheritance and copy

- The default copy-constructor of a class also calls default copy-constructors of parents
- The same is true for the copy assignment operator

```
House h1;  
House h2=h1; //calls default copy-constructor
```

↑
copies members of House (bedrooms, baths) *and*
members of Building (rooms, floors, area...)

- But if you write your own copy constructor you have to call the copy constructor of the parent:

```
House(const House &c) {  
    bedrooms=c.bedrooms;  
    baths=c.baths;  
}
```

↑
wrong! Building::rooms, Building::floors,
Building::area are *not* copied

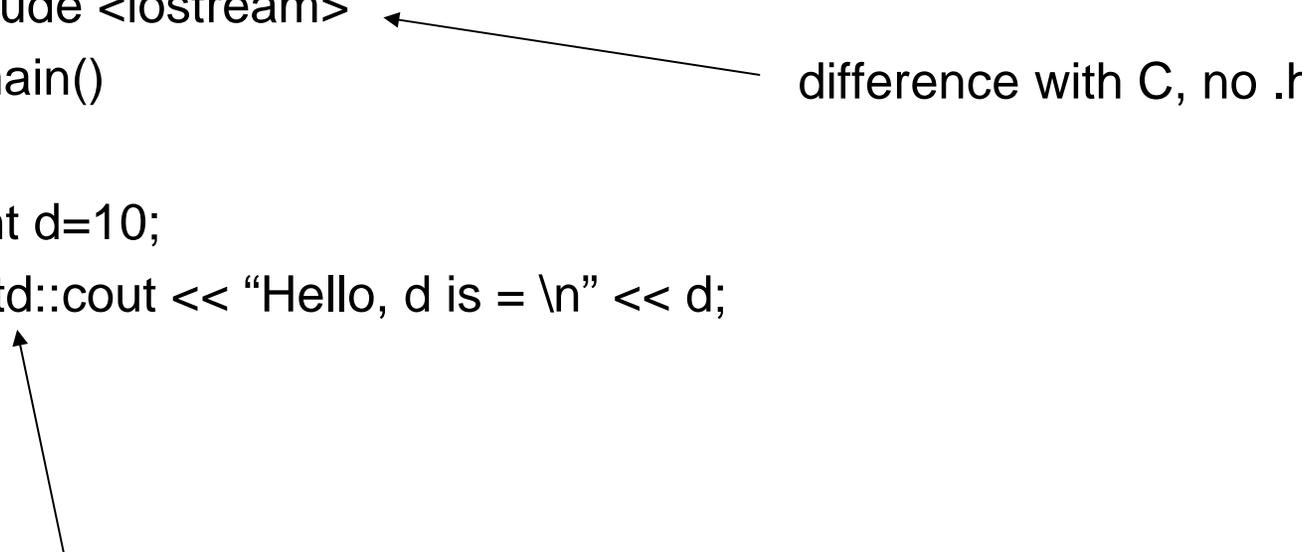
```
House(const House &c): Building(c) {  
    bedrooms=c.bedrooms;  
    baths=c.baths;  
}
```

↑
ok, calls Building copy constructor

Overview of useful C++ features

iostream class

```
#include <iostream>
int main()
{
    int d=10;
    std::cout << "Hello, d is = \n" << d;
}
```



difference with C, no .h

to use symbols defined in these header files you must specify the namespace

Warning: old C++ header files are similar to the C-style header files, but they do not have namespace. They are still around for compatibility reasons, but be careful not to mix old and new header files (e.g. never include both `<iostream>` and `<iostream.h>` at the same time)

- If we use the keyword *using namespace std* there is no need to add *std::*

ok in cpp/cc files, dangerous in header files!

```
#include <iostream>
using namespace std;
int main()
{
    int d=10;
    cout << "Hello, d is = \n" << d;

    //use of formats:
    cout << "a number in decimal" << dec << 15 << endl;
    cout << "in octal" << oct << 15 << endl;
    cout << "in hex" << hex << 15 <<endl;
}
```

- Reading input:

```
#include <iostream>
using namespace std;
int main()
{
    int d;
    cout << "Please enter a decimal number\n";
    cin >> d;
}
```

Strings

- C style strings (char [] or char *) is tedious
- Size is predefined (unless we use dynamic memory), difficult to compare, concatenate...
- C++ provides a string class to simplify these tasks

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
    string s1, s2; //empty strings
    string s3="Hello"; //initialized string
    s2 = "World"; //assignment
    s1=s3+s2;
    s1+="!!"
    cout << s1;    //print "Hello World!!"
    printf("%s",s1.c_str()); //print "Hello World!!"
}
```

if you like printf, you can get a c-style string out of a string

Reading/Writing files

- `<fstream>` defines `ifstream/ofstream` which behave like `cin/cout` but on files

```
#include <string>
#include <fstream>
using namespace std;
int main()
{
    ifstream in ("input.txt");
    ofstream out ("output.txt");

    string s;
    while(getline(in, s))
        ofstream << s << "\n";
}
```

read a line from input file and put it in a string
(discard new lines "\n"), return false when reaches
EOF

writes s into output file

Virtual functions and Polymorphism

A problem:

```
class Building {
    int rooms;
    int floors;
    int area;
    ...
public:
    ...
    void print() {
        printf("Calling Building::print\n");
    }
};
```

```
class House: public Building {
    int bedrooms;
    int baths;
public:
    ..
    void print() {
        printf("Calling House::print\n");
    }
};
```

```
House h;
Building *p=&h;
```

```
p->print();
```

upcast: loses type information about the object

calls Building::print() and *not* House::print()

```

class Employee {
public:
    enum Empl_type {M,E};
    Empl_type type;

    Employee():type(E) {}

    string name;
    Date hiring_date;
    int department;

    void print() const {
        cout << name << department << endl;
    }
};

```

also called "type field"

```

class Manager: public Employee {
public:
    Manager()
    {type = M;}

    Employee *group; //people managed
    int level;

    void print() const {
        cout << level;
        // print all people managed
    }
};

```

```

void print_employee(const Employee *e)
{
    switch (e->type) {
        case Employee::E:
            e->print();
            break;
        case Employee::M:
            e->print();
            Manager *p=static_cast<Manager *>(e);
            p->print();
            ...
            break;
    }
}

```

print_employee must know of all possible kind of employee(s), error prone, maintenance problem, changed/additions to the type of Employees are propagated to the code

Virtual functions

- Virtual functions overcome the “type-field” problem
- They allow the programmer to declare functions in a base class that can be redefine in each derived class
- The compiler will guarantee the correct correspondence between objects and functions applied to them

```

class Employee {
public:
    string name;
    Date hiring_date;
    int department;

    Employee();

    virtual void print() const {
        cout << name << department;
    }
};

```



```

class Manager: public Employee {
public:
    Manager()
    Employee *group; //people managed
    int level;

    void print() const {
        Employee::print();
        cout << level;
    }
};

```

print() act as an interface to print functions defined in this class *and* in derived classes, the compiler ensures that the right print() for the given object is invoked in each case

Manager *overrides* the base class version of print()

Note: the use of the scope resolution operator :: as in Employee::print() prevents the use of the virtual mechanism (avoid infinite recursion)

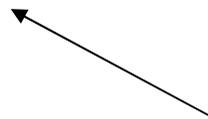
```
const int N=4;  
Employee *empls[N];
```

```
Manager m1;  
Manager m2;  
Employee e1;
```

```
empls[0]=&e1;  
empls[1]=&m1;  
empls[2]=&m2;
```

```
...
```

```
for(int k=0;k<N;k++)  
{  
    empls[k]->print();  
}
```



calls the right print() for each given object

Abstract classes

- In some cases base classes represent abstract concepts, for which objects cannot exist

```
class Shape {  
public:  
    virtual void rotate(int) { printf("Error cannot rotate a Shape\n") }  
    virtual void draw() {printf("Error cannot draw a Shape\n")}  
}
```

legal but useless

```
Shape s;  
s.rotate(10);  
s.draw();
```

```

class Shape {
public:
    virtual void rotate(int) = 0;
    virtual void draw() = 0;
    virtual bool isClosed() = 0;
}

```

pure virtual functions

```

Shape s;

```

error! cannot instantiate *abstract class*

An abstract class can be used as an *interface* and as a base for other classes

```

class Point {...};

class Circle: public Shape {
    Point center;
    int radius;
public:
    void rotate(int c) {}
    void draw() {..};
    bool isClosed() {return true;}
};

```

Circle is said to *implement* Shape
 is Circle does not implement any of
 the methods of Shape, it remains
 an abstract class (and cannot be
 instantiated)

Destructors

- Shape provides access to methods of Circle
- Problems might occur if we try to destroy an object through a pointer to one of its interface

```
{  
  
Circle circle;  
Shape &sh=circle;  
  
sh.draw();  
sh.rotate();  
}
```

fine, sh is just a reference, no destructor is called
circle destructor is called when required

```
Shape *sh=new Circle;
```

```
sh->draw();  
sh->rotate();
```

```
delete sh;
```

↑
troubles! not really
calling Circle::~~Circle()

Virtual Destructors

- The solution to this problem is simple: declare the destructor as *virtual*

```
class Shape {  
public:  
    virtual void rotate(int) = 0;  
    virtual void draw() = 0;  
    virtual bool isClosed() = 0;  
  
    virtual ~Shape(){}  
};
```

must provide implementation

```
Shape *sh=new Circle;
```

```
sh->draw();  
sh->rotate();
```

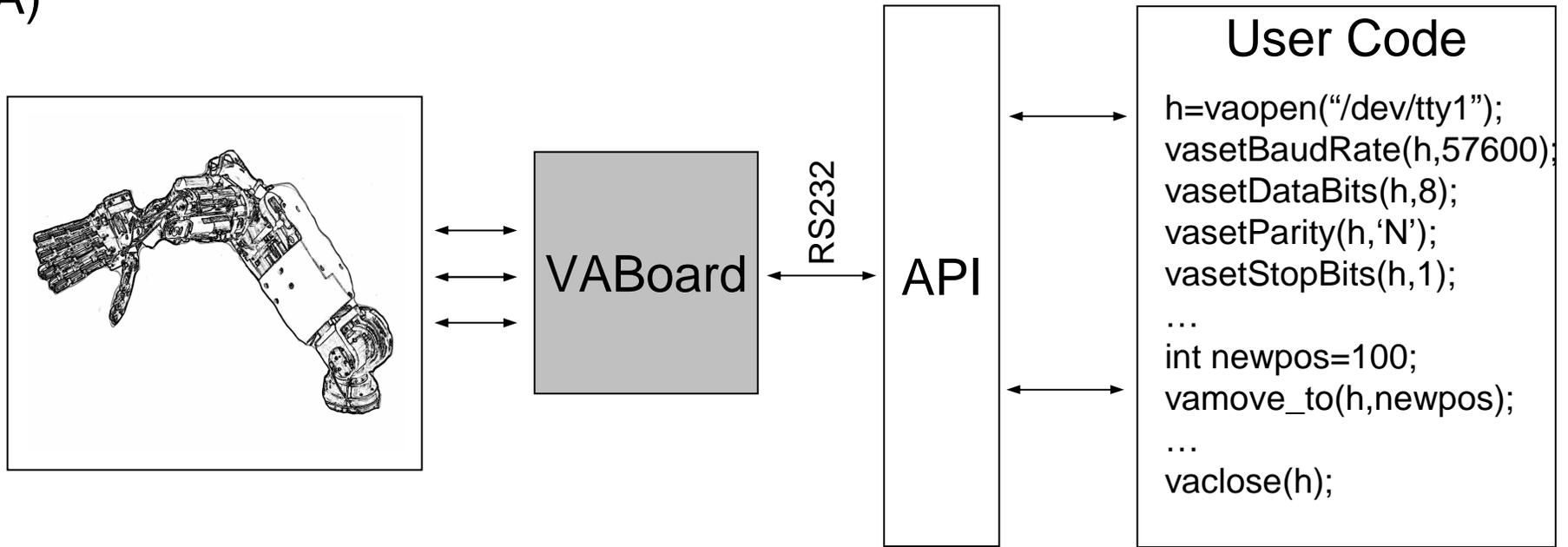
```
delete sh;
```

now calling Circle::~~Circle, though ~Shape

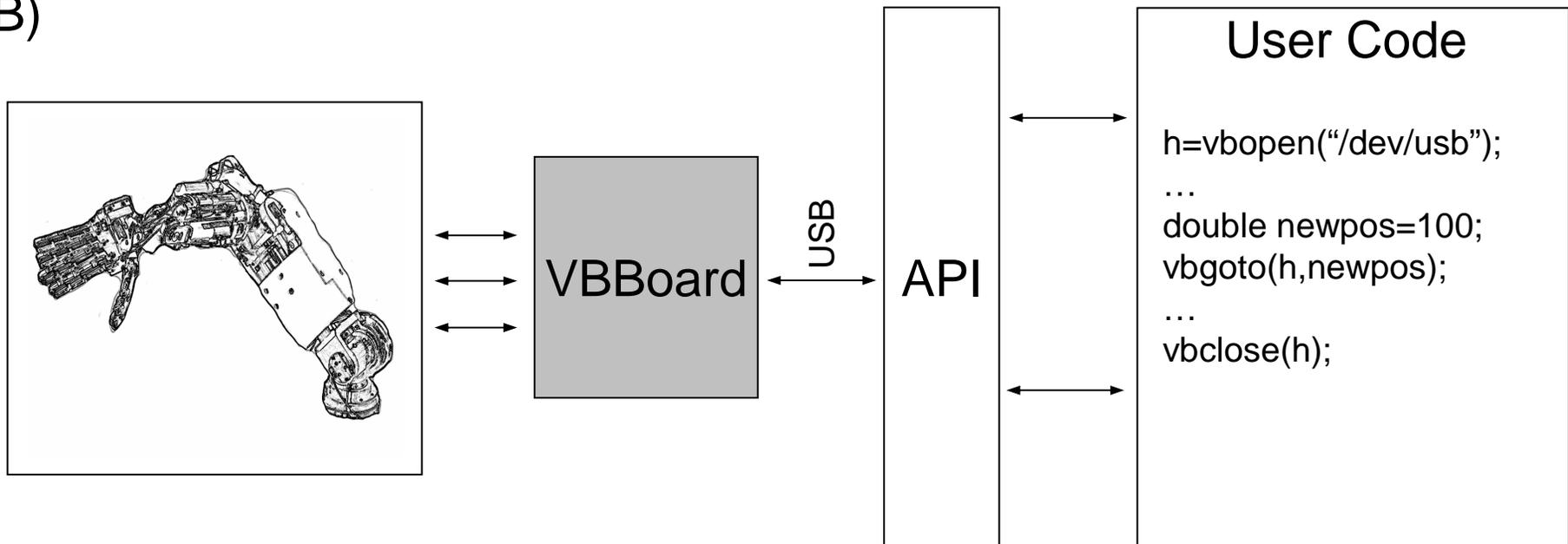
Why interfaces?

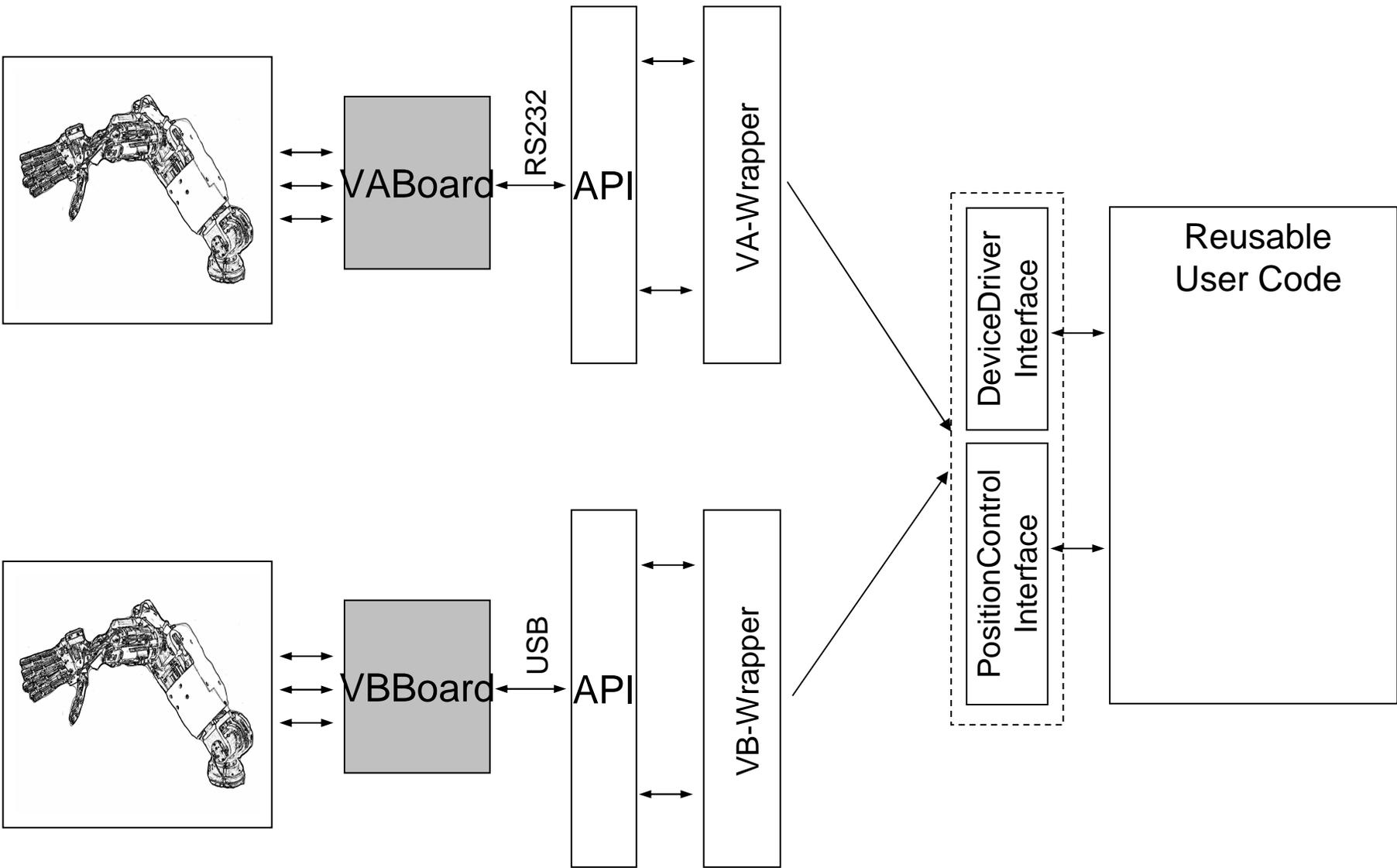
- Interfaces allow us to reduce dependencies between user code (i.e. the code that uses a given class) and the implementation of a specific class
- Example: yarp device drivers

A)



B)





IBoard

```
virtual bool open(const char *filename) = 0;  
virtual bool close() = 0;
```

IBoard

```
virtual bool open(const char *filename) = 0;  
virtual bool close() = 0;
```

IPosition

```
virtual void positionMove(int j, double np)=0;  
virtual void positionMove(double *np)=0;  
virtual int getAxis()=0;  
...
```

IBoard

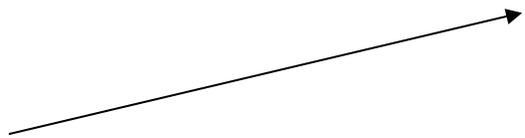
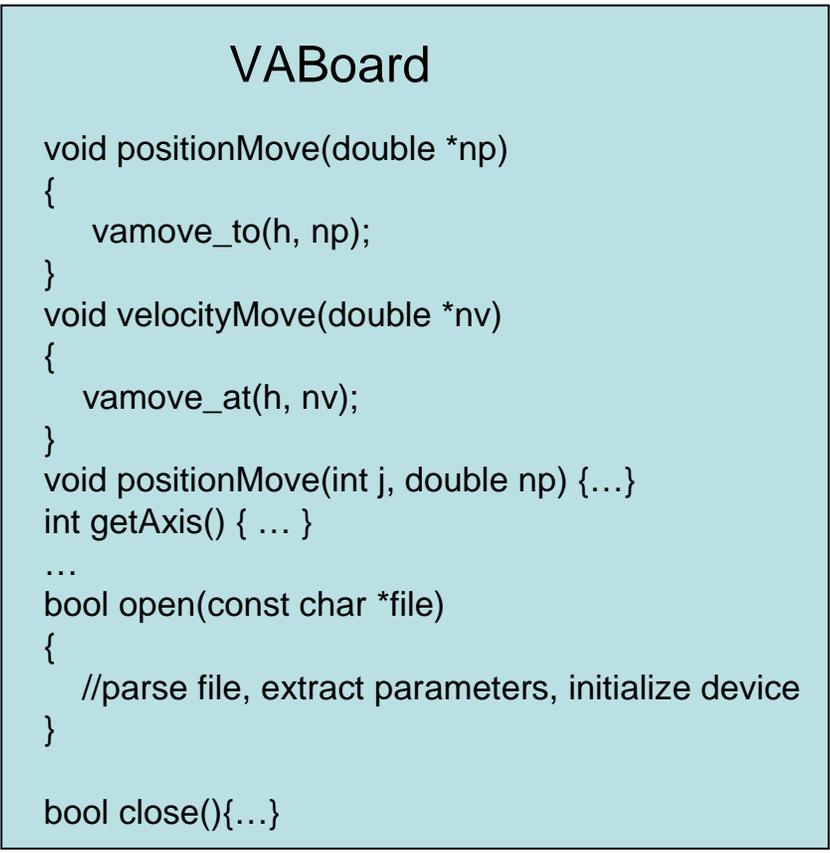
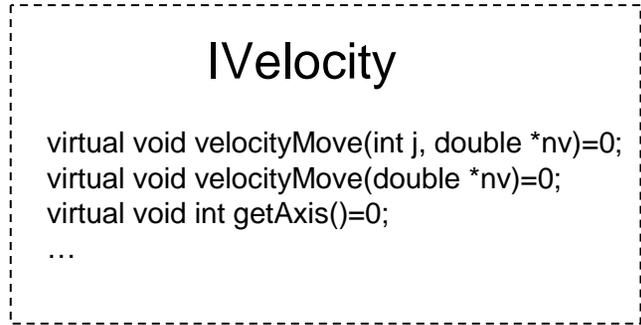
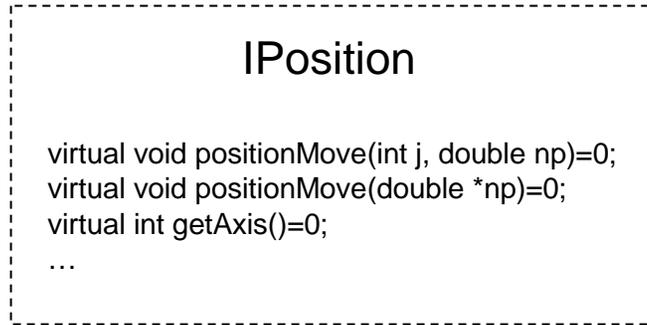
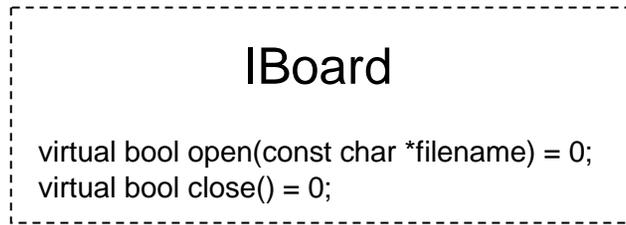
```
virtual bool open(const char *filename) = 0;  
virtual bool close() = 0;
```

IPosition

```
virtual void positionMove(int j, double np)=0;  
virtual void positionMove(double *np)=0;  
virtual int getAxis()=0;  
...
```

IVelocity

```
virtual void velocityMove(int j, double *nv)=0;  
virtual void velocityMove(double *nv)=0;  
virtual void int getAxis()=0;  
...
```



IBoard

```
virtual bool open(const char *filename) = 0;  
virtual bool close() = 0;
```

IPosition

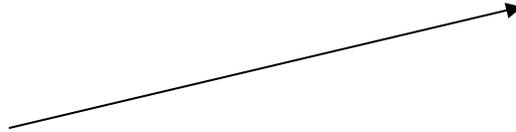
```
virtual void positionMove(int j, double np)=0;  
virtual void positionMove(double *np)=0;  
virtual int getAxis()=0;  
...
```

IVelocity

```
virtual void velocityMove(int j, double *nv)=0;  
virtual void velocityMove(double *nv)=0;  
virtual void int getAxis()=0;  
...
```

VBoard

```
void positionMove(double *np)  
{  
    vbgoto(h, np);  
}  
  
void positionMove(int j, double np) {...}  
int getAxis() { ... }  
...  
bool open(const char *file)  
{  
    //parse file, extract parameters, initialize device  
}  
  
bool close(){...}
```



```
int main()
{
    VABoard robot;
    robot.open("vaboard.conf");

    //begin generic code
    moveRobot(&robot);

    robot.close();
}
```

```
void moveARobot(IPosition *ipos)
{
    int nj=ipos->getAxis();
    for(int k=0;k<nj)
        ipos->positionMove(0,10);
}
```

```
int main()
{
    VBBoard robot;
    robot.open("vbboard.conf");

    //begin generic code
    moveRobot(&robot);

    robot.close();
}
```

```
int main()
{
    VABoard robot;

    robot.open("vaboard.conf");

    //begin generic code
    moveRobot(&robot);

    robot.close();
}
```

```
void moveARobot(IPosition *ipos)
{
    int nj=ipos->getAxis();
    for(int k=0;k<nj)
        ipos->positionMove(0,10);
}
```

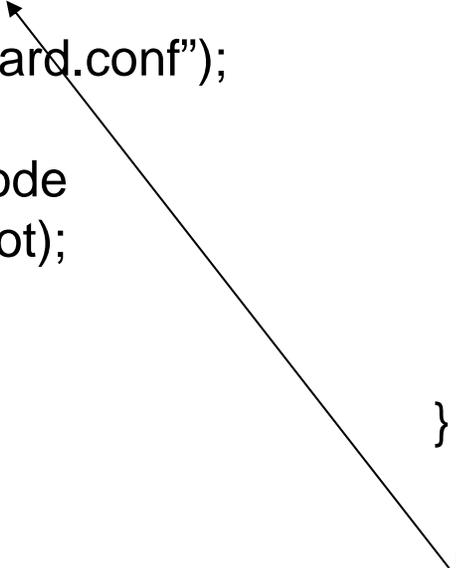
```
int main()
{
    VBBoard robot;

    robot.open("vbboard.conf");

    //begin generic code
    moveRobot(&robot);

    robot.close();
}
```

still implementation details...what if we need to create objects in different parts of our code?



Factory

- There is not much we can do, but at least we can minimize the number of places in which object creation takes place
- Delegate object creation to a class (usually called a *factory*)

```
class BoardMaker
{
    static IBoard *create(const string type)
    {
        if (type=="VABoard") return new VABoard;
        if (type=="VBBoard") return new VBBoard;
    }
};
```

```
int main()
{
    IBoard *va=BoardMaker::create("VABoard");
    IBoard *vb=BoardMaker::create("VBBoard");

    va->open("va.conf");
    vb->open("vb.conf");

    IPosition *iposa=(IPosition *)(va);
    IPosition *iposb=(IPosition *)(vb);

    moveRobot(iposa);
    moveRobot(iposb);

    ....
}
```

we will see soon that the use of `dynamic_cast` is more appropriate, here



- How do we check that we perform the correct cast?
- Or, in other words how can we determine from a pointer to a base class the type of the original object?
- Example:

```
IPosition *iposa=(IPosition *)(va);
IPosition *iposb=(IPosition *)(vb);
```

fine, va points to a VABoard
which derives from (implements)
IVelocity

```
IVelocity *ivela=(IVelocity *)(va);
IVelocity *ivelb=(IVelocity *)(vb);
```

troubles, vb points to a VBBoard
which does not derive from
IVelocity

```
ivela->velocityMove(..); //ok
ivelb->velocityMove(..); ????
```

```
VABoard *boardA=(VABoard *) iposa; // is this safe?
VBBoard *boardB=(VBBoard *) iposb; // is this safe?
VBBoard *boardC=(VBBoard *) iposa; // ????
```

Downcasting

- Casting a pointer or a reference to a base class (or up an inheritance hierarchy) is called *upcasting*.
Upcasting is safe, because the classes converge to a more general class
- Downcasting is more difficult, there are in general multiple derived classes from a base class
- C++ provides a special *explicit cast* called *dynamic_cast* which is a type-safe downcast operation
- The return value of `dynamic_cast` is a valid pointer only if the cast is successful, otherwise the pointer is zero

```
IBoard *va=BoardMaker::create("VABoard");
IBoard *vb=BoardMaker::create("VBBoard");

IPosition *iposa=dynamic_cast<IPosition *>(va);
IPosition *iposb=dynamic_cast<IPosition *>(vb);

IVelocity *ivela=dynamic_cast<IVelocity *>(va);
IVelocity *ivelb=dynamic_cast<IVelocity *>(vb);

if (ivela)
    ivela->velocityMove(..);
if (ivelb)
    ivelb->velocityMove(..);

VABoard *boardA=dynamic_cast<VABoard *>(iposa);
VBBoard *boardB=dynamic_cast<VBBoard *>(iposb);
VBBoard *boardC=dynamic_cast<VBBoard *>(iposa);
```

- Mechanisms like `dynamic_cast` uses what is called run-time type information (RTTI)
- RTTI allows to discover type information that has been lost by upcasting
- Another mechanism to detect the type of an object is *typeid*
- Consider this example:

```
class Shape {
public:
    virtual ~Shape(){}
};
```

```
class Circle: public Shape{};
class Square: public Shape{};
```

```
#include <typeinfo>
using namespace std;
...
Shape *c=new Circle;
Shape *sq=new Square;

if((typeid(*c)==typeid(Circle))
    printf("c points to a Circle");

if(typeid(*sq)==typeid(Square))
    printf("sq points to a Square");

cout << "sq points to";
cout << typeid(*sq).name() << endl;
```

Pointers to functions

- in C it is possible to declare pointers to functions
- pointers to functions can be stored, assigned as parameters, returned or organized in vectors

- **Syntax:**

```
int myFunction(char c);
```

f is a pointer to a function which received a char as a parameter and returns an integer

```
int (*f)(char);
```

```
f=myFunction;
```

ok, assign *myFunction* to *f* (its address), this is fine because prototypes of *myFunction* and *f* match

```
(*f)('c');
```

call *myFunction* through *f*

- This is powerful because the function that is actually called is determined at runtime

```
int foo (char c){...}  
int bar(char d){...}
```

```
int (*f)(char);
```

```
if (rand()%2==0)  
    f=foo;  
else  
    f=bar;
```

```
(*f)('c');
```

which function is called?

Example, a simple FSM

Another example, “callback”

```
float cost(float x) {  
    ...  
    return x*x-0.5;  
}
```

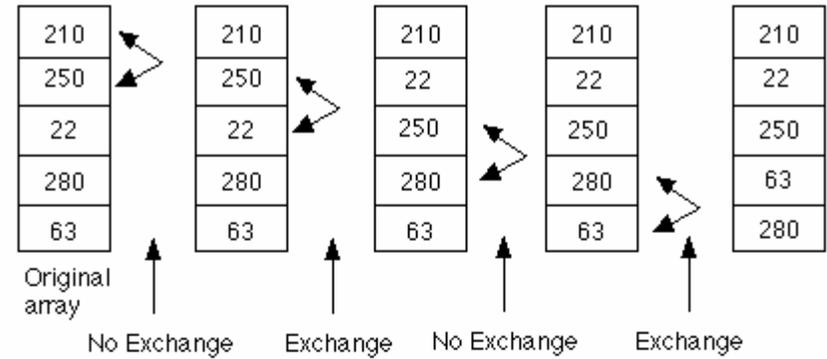
```
int main() {  
    ..  
    float min=minimize(cost);  
    ..  
}
```

```
float minimize( float (*g)(float))  
{  
    float x; // pick initial guess for x  
    while(!done)  
    {  
        // update x  
        // call g(x) to get cost  
        // check stop criteria  
    }  
  
    return x;  
}
```

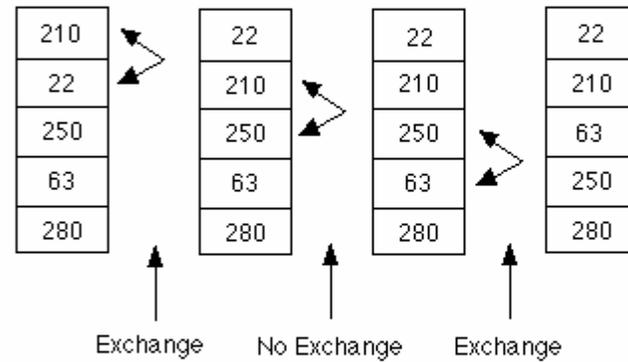
Templates

Example: Bubble sort

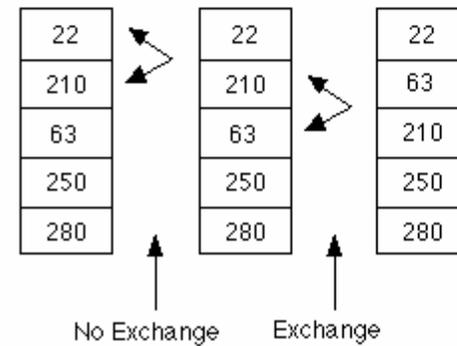
1st pass



2nd pass



3rd pass



**The sort is
completed after
three passes.**

Bubble sort: code

```
int main()
{
    int elems[6]={10,5,4,3,1,8};
    bubble(elems, 6);

    for(int k=0;k<6;k++)
    {
        cout<<elems[k]<< " ";
    }
}
```

print 1 3 5 8 10



```
void bubble(int *items, int count)
{
    int a, b;
    int t;
    for (a=1;a<count;a++)
        for(b=count-1;b>=a;b--)
            if (items[b-1] > items[b]) {
                t=items[b-1];
                items[b-1]=items[b];
                items[b]=t;
            }
}
```

- The bubble sort algorithm is generic, it does not need knowledge about the objects it is manipulating, provided it can compare and copy them
- What if we need to sort floating point numbers? or strings?
- Templates offer a good (and efficient) solution to this, as they allow to reuse *source code*
- The *template* keyword tells the compiler that the function will manipulate one or more unspecified types. At the time the code is generated those types must be specified so that the compiler can substitute them

```
template<class X> void bubble(X *items, int count)
```

```
{
```

```
    int a, b;
```

```
    X t;
```

```
    for (a=1;a<count;a++)
```

```
        for(b=count-1;b>=a;b--)
```

```
            if (items[b-1] > items[b]) {
```

```
                t=items[b-1];
```

```
                items[b-1]=items[b];
```

```
                items[b]=t;
```

```
            }
```

```
}
```

X is the substitution parameter

assumptions:
can copy and compare T

```
int main()
```

```
{
```

```
    int iv[6]={10,5,4,3,1,8};
```

```
    bubble(iv, 6);
```

```
    double db[4]={9.3,1,0.1,4};
```

```
    bubble(db, 4);
```

```
}
```

can deduce the value of T from the
parameters, in these cases *int ** and *double **

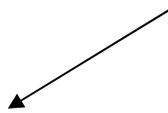
```
double elems[]={...};  
bubble(elems, 6);
```

```
bubble<double>(elems, 6);
```

```
template<class T> T* create()  
{  
    return new T;  
}
```

```
int *p=create<int>();
```

cannot deduce 'int' from parameters, must
provide template argument



- Let's go back to our stack... suppose we want to create a stack of int(s)

```
class IntStack {
    enum{ssize=100};
    int stack[ssize];
    int top;
public:
    IntStack(): top(0) {}
    void push(int e) {
        if (top>=ssize)
            cout<<"Error stack full" << endl;
        else
            stack[top++]=e;
    }
    int pop(){
        if (top<=0) {
            cout << "Error stack empty" << endl;
            return 0;
        }
        else
            return stack[--top];
    }
};
```

- Similar question: we need a stack of double(s) and we want to reuse as much as possible code written (and debugged) for IntStack

```

template<class T>
class Stack {
    enum{ssize=100};
    T stack[ssize];
    int top;
public:
    Stack(): top(0) {}
    void push(const T &e) {
        // check if stack is full
        stack[top++]=e;
    }
    T pop(){
        // check if stack is empty
        return stack[--top];
    }
};

```

T is the substitution parameter

```

int main()
{
    Stack<int> is;
    Stack<double> ds;
    Stack<string> ss;

    is.push(10);
    ds.push(3.14);
    ss.push("hello");

    cout<<ss.pop();
}

```

assumption:
can copy T

the compiler expands the Stack template for the type in <> and creates new classes; these are classes like the ones you would get by performing substitution by hand (expect it is automatic and correct...)

- Non-inline function definitions

```
template<class T>
class Stack {
    enum{ssize=100};
    T stack[ssize];
    int top;
public:
    Stack(): top(0) {}
    void push(const T &e);
    T pop();
};
```

```
template<class T>
void Stack<T>::push(const T&e) {
    // push code...
}
```

```
template<class T>
T Stack<T>::pop() {
    // pop code...
}
```

although separated, usually go in header files

Source code organization

```
#include <iostream>
```

```
template<class T> void out(const T&t) {std::cout<<t;}
```



make it a single file out.h and #include it when needed

```
user1.cpp:  
#include "out.h"  
// use out
```

```
user2.cpp:  
#include "out.h"  
// use out
```



```
out.h:  
template<class T> void out(const T&t);
```

```
out.cpp:  
#include <iostream>  
#include "out.h"
```

```
export template<class T>  
void out(const T&t) {std::cout<<t;}
```

```
user1.cpp:  
#include "out.h"  
// use out
```

```
user2.cpp:  
#include "out.h"  
// use out
```

Constants in templates

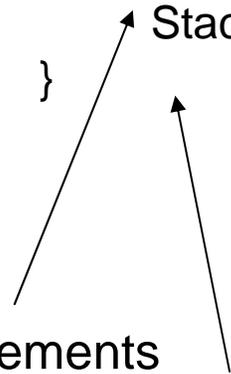
- Template arguments can be built-in types
- The values of these arguments become compile-time constant for a given template
- You can even have default values

```
template<class T, int ssize=100>
class Stack {
    T stack[ssize];
    int top;
public:
    Stack(): top(0) {}
    void push(const T &e);
    T pop();
};
```

```
int main()
{
    Stack<int> is;
    Stack<double,10> ds;
}
```

stack of 100 elements

stack of 10 elements



Template specialization

- It is possible to perform the overload of a template for a particular value of the parameter

```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp=a;
    a=b;
    b=temp;
    cout<<"Generic\n";
}
```

```
template<> void swapargs<int> (int &a, int &b)
{
    X temp;
    temp=a;
    a=b;
    b=temp;
    cout<<"Specialized for int\n";
}
```

```
int main() {
    int a=10;
    int b=20;
```

```
    double c=3.14;
    double d=0.0;
    swapargs(a,b);
    swapargs(c,d);
```

uses specialized version

uses generic version

```
}
```

- Specialization can also be applied to template objects

```
template<>
class Stack<int> {
    int stack[ssize];
    int top;
public:
    Stack(): top(0) {}
    void push(const int &e);
    int pop();
};
```

specifies that this specialization is to be used for Stack<int>



It is ok to declare several function template with the same name:

```
template<class T> T sqrt(T);  
template<class T> complex<T> sqrt(complex<T>);  
double sqrt(double);
```

Use of Template Arguments to Specify Policy

- The sorting algorithm we implemented was using a sorting criteria for comparing elements in the array
- Who specifies this criteria?
- One possibility is to hardwire it in the type (by overloading operator $>$ or $<$), but in some cases there are different ways of sorting elements... for example you might want to use a case sensitive or case insensitive criteria
- Another possibility is to express the sorting criteria in general terms that can be defined not only for a specific type, but for a specific *use* of a specific type

Example: compare two strings

- Suppose we want to compare two strings and we want to specify the way characters are compared

```
int compare(const string &str1, const string &str2)
{
    for(int i=0; i<str1.length() && i<str2.length(); i++)
        if (!str1[i]==str2[i])
            return 0;
    return 1;
}
```

here we rely on comparison between char

return 0 if different, 1 otherwise,
check only elements common to
both strings (simplified version)

```
class Cmp {
public:
    static int eq(char a, char b)
    { return a==b; }
};
```

```
template<class C>
int compare(const string &str1, const string &str2)
{
    for(int i=0; i<str1.length() && i<str2.length(); i++)
        if (!C::eq(str1[i],str2[i]))
            return 0;
    return 1;
};
```

now we use C::eq() static method inside the class C

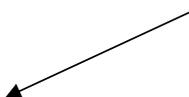
```
use:
string string1;
string string2;
compare<Cmp>(string1, string2);
```

the difference is that in this case we are passing the class Cmp which specifies how we want to compare elements T

- Now we can define another class:

```
CICmp {  
public:  
    static int eq(char a, char b)  
        { return tolower(a)==tolower(b); }  
};
```

case insensitive eq



- Here is how we use different policies:

```
string string1;  
string string2;
```

```
compare<Cmp>(string1, string2);    //use standard compare
```

```
compare<CICmp>(string1, string2); //use case ins. compare
```

- If we want, we can make one of the choices a default:

```
template<class C=Cmp>
int compare(const string &str1, const string &str2)
{
    ....
};
```

```
compare<Cmp>(string1, string2);
compare(string1,string2); //same thing, uses default
```

Standard Template Library Overview

- STL (Standard Template Library) was created and designed during the standardization of C++
- It provides template classes of general use to implement algorithms and data structure like vectors, lists, queues and stacks
- Being template based it is general and efficient

- It is made of three elements: containers, iterators and algorithms
- Containers are objects which contain other objects; vector, list, queue, stack
- Iterators are similar to pointers; they are useful to get access to a container like a pointer to an array; like pointers iterators can be dereferenced with *, incremented or decremented
- Algorithms contain code to manipulate containers, like initialization, sorting, search...

Vector

- Implements a dynamic array

```
#include <vector>
using namespace std;
struct Entry{
    string name;
    int number;
};
```

```
vector<Entry> phone_book(1000);
```

```
cout << phone_book[0].name << " " << phone_book[0].number;
```

```
phone_book.resize(phone_book.size()+10); ← easy to change size
```

```
//vectors have correctly behaving copy constructors and copy assignment
```

```
vector pb2=phone_book; ← copies all elements, can be expensive
```

Warning: vector does not provide range checking:

```
Entry e=phone_book[10000]; ← out of range?
```

- Other member functions:
 - empty() //return true/false if the vector is empty
 - clear() //remove all elements in the vector
 - size() //return the size of the vector
 - push_back(const T &) //insert an element at the end of the vector
- Use the iterator:

```
vector<double> v(10);  
vector<double>::iterator p; //p is an iterator of vector<double>
```

```
p=v.begin();  
while(p!=v.end())  
{  
    *p=0.0;  
    p++;  
}
```



```
int k=0;  
for(int k=0;k<v.size();k++)  
    v[k]=0.0;
```

- Insert/erase elements

```
vector<double> v;  
vector<double>::iterator p; //p is an iterator of vector<double>
```

```
for(int k=0;k<10;k++)  
    v.push_back(0.0);
```

```
p=v.begin();  
p+=2;
```

```
v.insert(p, 5, 1.0); //insert 5 times 1.0 at position pointed to by p
```

```
p=v.begin();  
p+=2;
```

```
v.erase(p,p+5);
```

```
vector<double> v2;  
p=v.begin();  
v2.insert(p, v1.begin(), v1.end()); //insert v1 in v2
```

List

- It is a bidirectional list
- Different than vector, only sequential access
- Easier to insert/remove elements

```
#include <list>
```

```
using namespace std;
```

```
list<Entry> phone_book;
```

```
Entry e;
```

```
phone_book.push_front(e); //add at beginning
```

```
phone_book.push_back(e); //add at end
```

```
phone_book.insert(p, e); //add before the element referred to by p
```

```
phone_book.erase(p); //remove element referred to by p
```

```
Entry e=phone_book[0]; //error [] not available for list..., must use iterators
```

- Using iterators to access elements of a list

```
// example search s in phone_book
```

```
Entry s;
```

```
s.name="Foo";
```

```
s.number=11111;
```

```
list<Entry>::iterator i;
```

or, since we are not going to modify elements, list<Entry>::const_iterator i

```
for(i=phone_book.begin(); i!=phone_book.end(); ++i)
```

```
{
```

```
    if (s.name==(i).name)
```

```
        cout << (i).name << " " << (i).number << "\n";
```

```
}
```

returns the element *after* the last one

- Notice that the use of iterators makes containers interchangeable. In the above example list<Entry> can be substituted with vector<Entry>

- Easy to sort a list:

```
list<int> myList;
```

```
for(int k=0;k<10; k++)  
    myList.push_back(rand());
```

```
myList.sort();
```

- Merging lists:

```
list<int> list1;
```

```
list<int> list2;
```

```
for(int k=0;k<10; k+=2)  
    list1.push_back(k);
```

```
//list1 is 0,2,4,6,8
```

```
for(int k=1;k<11;k+=2)  
    list2.push_back(k);
```

```
//list2 is 1,3,5,7,9
```

```
list1.merge(list2);
```

```
//list 2 is empty, list1 is 0,1,2,3,4,5,6,7,8,9
```

Map

- Looking up a name in a list of (name,number) pairs is tedious
- In addition a linear search can be inefficient for long lists
- A *map* is a more suitable container for such tasks

```
map<string, int> phone_book;
```

- We can search in the map a value of its first type (the *key*); the map returns the element whose key matches the one we specified

```
string s="Foo";  
map<string, int> iterator p;  
p=phone_book.find(s);  
if (p!=phone_book.end())  
    cout << (*p).first << " " << (*p).second << endl;
```

in our case this is the "string" part

in our case this is the "int" part

- How to insert elements in a map

```
pair<string,int> tmp;  
tmp.first="Foo";  
tmp.second=424242;
```

```
phone_book.insert(tmp);
```

or:

```
phone_book.insert(pair<string, int>("Bar", 1234567));
```

....

- A map can be indexed by the value of the *key* (the first element), to get the corresponding *value* (the second element)

```
string s="Foo";  
int i=phone_book[s];  
if (i!=0)  
    cout << s << " " << i << endl;
```

return the second element if found (0 otherwise)



More Advanced Topics

Algorithms

- Containers are not really useful on their own, most of the time we want to manipulate the data they contain
- Example: sort, print, remove elements, extract subsets, search...
- STL contains algorithms (usually template functions) to simplify these operations
- To use STL algorithms you `#include <algorithm>`

Examples:

- Count elements in a container with *count* and *count_if*

```
vector<bool> v;  
// suppose we fill v with random true/false
```

```
i=count(v.begin(), v.end(), true); //count all elements in v s.c. v[]==true
```

- Another example:

```
bool dvidesBy3(int i)  
{  
    if ((i%3)==0)  
        return true;  
    return false;  
}
```

```
vector<int> v;  
// fill v...  
i=count_if(v.begin(), v.end(), dvidesBy3)
```

Function Object

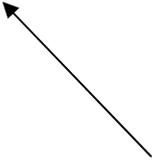
```
class ParF
{
    double a;
    double b;
    double c;
public:
    ParF(double aa, double bb, double cc)
    {
        a=aa;
        b=bb;
        c=cc;
    }
    double operator()(double x)
    {
        return (a*x*x+b*x+c);
    }
};
```

```
int main()
{
    ParF f(1,0,1); //x^2+1
    for(int k=-10;k<10;k++)
    {
        double x=k;
        cout << f(x);
    }
    return 0;
}
```

call f.operator()



overload operator()



More stuff...

- How polymorphism is implemented in C++
- exceptions
- return const copy
- return type optimization
- self assignment in copy operator
- use of private copy constructor to protect accidental copy
- const cast implementation

References

Linguaggio C, B.W. Kernighan, D.M. Ritchie, Jackson Libri

C++ Seconda Edizione, H. Schildt, McGraw Hill

The C++ Programming Language, Bjarne Stroustrup, Addison Wesley

Thinking in C++ Volume I, Bruce Eckel, Prentice Hall
(online: www.bruceeckel.com)

Thinking in C++ Volume II, Bruce Eckel, Prentice Hall
(online: www.bruceeckel.com)