Scuola Superiore Sant'Anna

Real-Time System Laboratory

Advanced course on C++

Giuseppe Lipari

Scuola Superiore S. Anna

Classes

"Those types are not abstract:

they are as real as int and float"

-Doug McIlroy

Abstraction

- An essential instrument for OO programming is the *support for data abstraction*
- C++ permits to define new types and their operations
- Creating a new data type means defining:
  - Which elements it is composed of (*internal structure*);
  - How it is built/destroyed (*constructor/destructor*)
  - How we can operate on this type (*methods/operations*)

Abstraction

- Abstraction is a very powerful instrument
  - By composing predefined data types (int, double, char, etc.) we create higher level entities;
- OO programming is based on data abstraction
- Given a complex problem:
  - decomposition in *objects* (data types)
  - define relationship and interactions between objects
  - decompose each object in smaller objects
  - the decomposition ends when every low level object can be represented with a predefined data-type

Data abstraction in C

- We can do data abstraction in C (and in almost any language)
  - however, the syntax is awkward

```
typedef struct __complex {
   double real_;
   double imaginary_;
} Complex;

void add_to(Complex *a, Complex *b);
void sub_from(Complex *a, Complex *b);
double get_module(Complex *a);
```

we have to pass the main data to every function

name clashing: if another abstract type defines a function add_to(), the names will clash!

No protection: any user can access the internal data using them improperly

Classical example

- Let's define the same data type in C++

```
class Complex {
   double real_;
   double imaginary_;
public:
    Complex();                              // default constructor
    Complex(double a, double b);            // constructor
    ~Complex();                             // destructor

    double real() const;                    // member function to get the real part
    double imaginary() const;               // member function to get the imag. part
    double module() const;                  // member function to get the module
    Complex &operator =(const Complex &a);       // assignment operator
    Complex &operator+=(const Complex &a);       // sum operator
    Complex &operator-=(const Complex &a));      // sub operator
};
```

How to use Complex

- Example:

```
Complex c1;        // default constructor
Complex c2(1,2);  // constructor
Complex c3(3,4);  // constructor

cout << "c1=(" << c1.real() << "," << c1.imaginary() << ")"<< endl;

c1 = c2;          // assignment

c3 += c1;         // operator +=

c1 = c2 +  c3;    // ERROR: operator + not yet defined
```

Using new data types

- The new data type is used just like a predefined data type
  - it is possible to define new functions for that type:
    - real(), imaginary() and module()
  - It is possible to define new operators
    - =, += and -=
  - The compiler knows automatically which function/operator must be invoked
- C++ is a *strongly typed* language
  - the compiler knows which function to invoke by looking at the type!

Class

- Class is the main construct for building new types in C++
  - A class is almost equivalent to a struct with functions inside
- In the C-style programming, the programmer defines structs, and global functions to act on the structs
- In C++-style programming, the programmer defines classes with functions *inside* them

Class

- A class contains *members*
- A member can be
  – any kind of variable (*member variables*)
  – any kind of function (*member functions or methods*)

```
class MyClass {
    int a;
    double b;
public:
     int c;

     void f();
     int getA();
     int modify(double b);
};
```

member variables

member variable

member functions

Accessing members

- A member can be accessed with the *dot* notation (or with the *arrow* in case of a pointer).

```
MyClass obj;

obj.c = 10;             // assigning to a member variable

obj.modify(8);          // invoking a member function

MyClass *p = &obj;   // pointer to obj

p->f();                 // invoking member f() through p;

cout << p->getA() << endl;   // invokes member getA()
```

Implementing member functions

- You can implement a member function in a separate .cpp file

```
class Complex {
  double real_;
  double imaginary_;
public:
   …
   double module() const;
   …
};
```

complex.h

```
double Complex::module() const
{
    double temp;
    temp = real_ * real_ + imaginary_ * imaginary_;
    return temp;
}
```

complex.cpp

Accessing internal members

```
double Complex::module() const
{
    double temp;
    temp = real_ * real_ + imaginary_ * imaginary_;
    return temp;
}
```

scope resolution

local variable

access to internal variables

- The :: operator is called *scope resolution* operator
- like any other function, we can create local variables
- member variables and functions can be accessed without *dot* or *arrow*

Access control

- A member can be:
  - **private**: only member functions of the same class can access it; other classes or global functions can't
  - **protected**: only member functions of the same class or of *derived* classes can access it: other classes or global functions can't
  - **public**: every function can access it

```
class MyClass {
private:
    int a;
public:
    int c;
};
```

```
MyClass data;

cout << data.a;          // ERROR: a is private!
cout << data.c;          // OK:  c is public;
```

Access control

- Default is private
- An access control keyword defines access until the next access control keyword

private (default)

public

private again

```
class MyClass {
   int a;
   double b;
public:
   int c;

   void f();
   int getA();
private:
   int modify(double b);
};
```

Access control and scope

- Member functions can access any other member

```
int xx;

class A {
   int xx;
public:
   void f();
};
```

global variable

member variable

```
void A::f()
{
   xx = 5;
   ::xx =3;

   xx = ::xx + 2;
}
```

access local xx

access global xx

Why access control?

- The technique of declaring private members is also called *encapsulation*
  - In this way we can precisely define what is interface and what is implementation
  - The *public* part is the interface to the external world
  - The *private* part is the implementation of that interface
  - When working in a team, each group take care of a module
  - To ensure that the integration is done correctly and without problems, the programmers agree on interfaces

Private

- Some people think that *private* is synonym of *secret*
  - they complain that the private part is visible in the header file
- private means *<u>not accessible from other classes</u>* and does not mean *<u>secret</u>*
- The compiler *needs* to know the size of the object, in order to allocate memory to it
  - In an hypothetical C++, if we hide the private part, the compiler cannot know the size of the object

Friends

- Access control rules can be broken with the *friend* keyword
  - a *friend* class can access the private members of the current class

```
class A {
  friend class B;
  int y;
  void f();
public:
  int g();
};
```

```
class B {
  int x;
public:
  void f(A &a);
};
```

```
void B::f(A &a)
{
  x = a.y;
  a.f();
}
```

B is friend of A

B can access private members of A

Friend functions and operator

- Even a global function or a single member function can be friend of a class

```
class A {
   friend B::f();
   friend h();
   int y;
   void f();
public:
   int g();
};
```

friend member function

friend global function

It is better to use the *friend* keyword only when it is really necessary

Nested classes

- It is possible to declare a class inside another class
- Access control keywords apply

```
class A {
  class B {
    int a;
  public:
    int b;
  }
  B obj;
public:
  void f();
};
```

Class B is private to class A: it is not part of the interface of A, but only of its implementation.

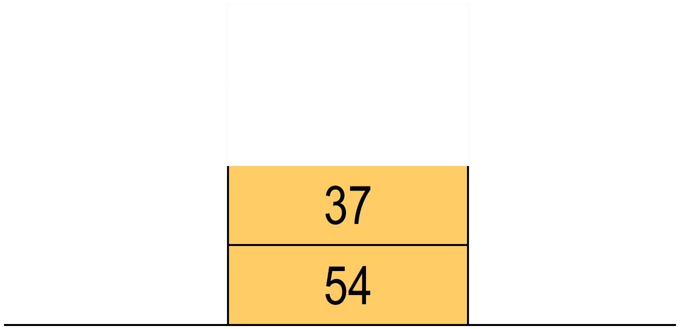
However, A is not allowed to access the private part of B!! (A::f() cannot access B::a).

To accomplish this, we have to declare A as friend of B

Time to do an example

- Let us implement a Stack of integers class
- At this point, forget about the std library
  - This is a didactic example!

```
Stack stack;
...
stack.push(12);
stack.push(7);
...
cout << stack.pop();
cout << stack.pop();
```



First, define the interface

```
class Stack {
   …
public:
    Stack();
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

constructor
&
destructor

Now the implementation

- Now we need to decide:
  - how many objects can our stack contain?
  - we can set a maximum limit (like 1000 elements)
  - we can dynamically adapt
    - computer memory is the limit
- Let's first choose the first solution
  - notice that this decision is actually part of the interface contract!

Implementation

```
class Stack {
public:
    Stack(int size);
    ~Stack();

    int push(int a);
    void pop();
    int size();
private:
    int *array_;
    int top_;
    int size_;
};
```

Constructor

- The constructor is the place where the object is created and initialized
  - Every time an object is defined, the constructor is called automatically
  - There is no way to define an object without calling the constructor
  - Sometime the constructor is called even when you don’t suspect (for example for temporary objects…)
- It’s a nice feature
  - it forces to think about initialization

Constructor for Stack

- The constructor is a function with the same name of the class and no return value
- It can have parameters:
  - in our case, the max_size of the stack

```
class Stack {
public:
    Stack(int size);
    ...
};
```

```
Stack::Stack(int size)
{
    array_ = new int[size];
    top = 0;
}
```

The new operator

- In C, if you needed memory from the heap, you would use malloc()
- In C++, there is a special operator, called new

```
Stack::Stack(int size)
{
   array_ = new int[size];
   size_ = size;
   top_ = 0;
}
```

creates an array of *size* integers

Destructor

- When the object goes out of scope, it is *destructed*
  - among the other things, its memory is de-allocated
- A special function, called *destructor*, is defined for every class
  - its name is a ~ followed by the class name
  - takes no parameters

```
class Stack {
    ...
    ~Stack();
    ...
};
```

```
Stack::~Stack()
{
    delete []array_;
}
```

The *delete* operator

- The opposite of *new* is *delete*
  - it frees the memory allocated by new

```
Stack::~Stack()
{
   delete [ ]array_;
}
```

deallocates *size* integers

this operation is needed because otherwise the memory pointed by *array_* would remain allocated

this problem is called *memory leak*

When are they called?

```
class Stack {
public:
    Stack();
    ~Stack();
    …
};
```

```
Stack::Stack(int size)
{
   size_ = size;
    array_ = new int[size_];
    top_ = 0;
    cout << “Constructor has been called\n!”;
}

Stack::~Stack()
{
    delete []array_;
    cout << “Destructor has been called\n”;
}
```

```
int main()
{
    cout << “Before block\n”;
    {
        Stack mystack(20);
        cout <<  “after constructor\n”;
        …
        cout << “before block end\n”;
    }
    cout << “After block\n”;
}
```

Default constructor

- A constructor without parameters is called *default constructor*
  - if you do not define a constructor, C++ will provide a default constructor that does nothing
  - if you **do provide** a constructor with parameters, the compiler does not provide a default constructor

```
Stack s1;

Stack s2(20);
```

Error!! No default constructor for Stack!

alling the user-defined constructor

Default constructor

- We did not define a default constructor on purpose
  - we cannot construct a Stack without knowing its size
  - see how C++ forces a clean programming style?
- However it is possible to define different constructors using *overloading*
  - usually, we need to provide several constructors for a class
- The compiler always provide a destructor, unless the programmer provides it

Implementing the Stack interface

- see the code in directory stack1/

Improving Stack

- let’s get rid of the size (see stack2/)

Initializing internal members

- Another feature of C++ is the *initialize-list* in the constructor
  - each member variable can be initialized using a special syntax

```
Stack::Stack()
{
  head_ = 0;
  size_ = 0;
}
```

⟺

```
Stack::Stack() : head_(0), size_(0)
{
}
```

It is like using a constructor for each internal member

Function overloading

- In C++, the argument list *is part of the name* of the function
  - this mysterious sentence means that two functions with the same name but with different argument list are considered *two different functions* and not a mistake
- If you look at the internal name used by the compiler for a function, you will see three parts:
  - the class name
  - the function name
  - the argument list

Function overloading

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

__A_f_int

__A_f_int_int

__A_f_double

__B_f_int

- To the compiler, they are all different functions!
  - beware of the type...

So, which one is called?

- The compiler looks at the type of the actual arguments to know which one is going to be called!

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

Beware of hidden casting!!

a.f(5);	__A_f_int
b.f(2);	__B_f_int
a.f(3.0);	__A_f_double
a.f(2,3);	__A_f_int_int
a.f(2.5, 3);	__A_f_int_int

Return values

- Notice that return values are not part of the name
  - the compiler is not able to distinguish two functions that differs only on return values!

```
class A {
   int floor(double a);
   double floor(double a);
};
```

It is not possible to overload the return value

Default arguments in functions

- Sometime, functions have long argument lists
- Sometime, some of these arguments do not change often
  - we would like to set default values for some argument
  - this is a little different from overloading, since it is the same function we are calling!

```
int f(int a, int b = 0);

f(12);    // it is equivalent to f(12,0);
```

Example

- see overload/
- You have also seen some debugging trick
  - when you cannot use more sophisticated debugging…
- Simple exercise:
  - write another constructor that takes a char* instead of a string

Constants

- In C++, when something is *const* it means that it cannot change. Period.
- Now, the particular meanings of const are a lot:
  - Don’t to get lost! Keep in mind: *const = cannot change*
- Another thing to remember:
  - constants must have an initial (and final) value!

Constants - I

- As a first use, *const* can substitute the use of #define in C
  - whenever you need a constant global value, use *const* instead of a define, because it is clean and it is type-safe

```
#define PI 3.14            // C style

const double pi = 3.14; // C++ style
```

In this case, the compiler **does not** allocate storage for pi

In any case, the const object has an *internal linkage*

Constants - II

- You can use *const* for variables that never change after initialization. However, their initial value is decided at run-time

```
const int i = 100;
const int j = i + 10;

int main()
{
    cout << “Type a character\n”;
    const char c = cin.get();
    const char c2 = c + ‘a’;
    cout << c2;

    c2++;            // ERROR! c2 is const!
}
```

compile-time constants

run-time constants

Constant pointers

- There are two possibilities
  - the pointer itself is constant
  - the pointed object is constant

```
int a
int * const u = &a;

const int *v;
```

the pointer is constant

the pointed object is constant (the pointer can change and point to another const int!)

Remember: a const object needs an initial value!

const function arguments

- An argument can be declared constant. It means the function can't change it
  - particularly useful with references

```
class A {
public:
    int i;
};

void f(const A &a) {
  a.i++;                    // error! cannot modify a;
}
```

You can do the same thing with a pointer to a constant, but the syntax is messy.

Passing by const reference

- Remember:
  - we can pass argument by value, by pointer or by reference
  - in the last two cases we can declare the pointer or the reference to refer to a constant object: it means the function cannot change it
  - Passing by constant reference is equivalent, from the user point of view, to passing by value
  - From an implementation point of view, passing by const reference is much faster!!

Constant member functions

- If we want to say: “this member function does not modify the object” we can use const at the end of the function prototype

```
class A {
    int i;
public:
    int f() const;
    void g();
};
void A::f() const
{
    i++;        // ERROR! this function cannot
                // modify the object
    return i;  // Ok
}
```

The compiler can call only const member functions on a const object!

```
const A a = …;

a.f();      // Ok
a.g();      // ERROR!!
```

Constant return value

- This is tricky! We want to say: “the object we are returning from this function cannot be modified”
  - This is meaningless when returning predefined types

```
const int f1(int a) {return ++a;}

int f2(int a) {return ++a;}

int i = f1(5);    // legal
i = f2(5);

const int j = f1(5); // also legal
const int k = f2(5); //also legal
```

these two functions are equivalent!

Return mechanism

- When returning a value, the compiler copies it into an appropriate location, where the caller can use it

```
int f2(int a) {return ++a;}

int i = f2(5);
```

1) *a* is allocated on the stack
2) the compiler copies *5* into *a*
3) *a* is incremented
4) the modified value of a is then copied directly into *i*
5) *a* is de-allocated (de-structed)

why const does not matter?

since the compiler *copies* the value into the new location, who cares if the original return value is constant? It is deallocated right after the copy!

Returning a reference to an object

- Things get more complicated if we are returning an object, by value or by *address*, that is by *pointer* or by *reference*
- But before looking into the problem, we have to address two important mechanisms of C++
  - copy constructors
  - assignment and temporary objects

Copy constructor

- When defining a class, there is another *hidden* default member we have to take into account

```
class X {
   int ii;
public:
   X(int i) {ii = i;}
   X(const &X x);

   int get() {return ii;}
   int inc() {return ++ii;}
};
```

copy constructor

If we don't define it, the compiler will define a standard version
The standard version will perform a copy member by member (*bitwise copy*)

Copy constructor

- An example

```
X x1(1);

X x2(x1);

cout << x1.get() << "   " << x2.get() << endl;

x1.inc();
cout << x1.get() << "   " << x2.get() << endl;
```

Copy contructor

We should be careful when defining a copy constructor for a class

we will address more specific issues on copy constructors later

Copy constructor

- The copy constructor is implicitly used when passing an object by value

```
void f(X x)
{
   ...
}

X x1;

f(x1);
```

when calling f(), the actual argument x1 is copied into formal parameter x by using the copy constructor

This is another reason to prefer passage by const reference!

The *complex* example

```
class Complex {
  double real_;
  double imaginary_;
public:
   Complex();                             // default constructor
   Complex(const Complex& c);             // copy constructor
   Complex(double a, double b);           // constructor
   ~Complex();                            // destructor

   double real() const;                   // member function to get the real part
   double imaginary() const;              // member function to get the imag. part
   double module() const;                 // member function to get the module
   Complex& operator =(const Complex& a);     // assignment operator
   Complex& operator+=(const Complex& a);     // sum operator
   Complex& operator-=(const Complex& a));    // sub operator
};
```

How to implement the copy constructor

```
Complex::Complex(const Complex& c)
{
   real_ = c.real_;
   imaginary_ = c.imaginary_;
}
```

this is equivalent to the default one!

- Now we can invoke it for initializing c3:

```
Complex c1(2,3);
Complex c2(2);
Complex c3(c1);

cout << c1 << "   " << c2 << "   " << c3 << "\n";
```

Copy constructor and assignment operator

- Remember that we also defined an assignment operator for Complex:

```
Complex c1(2,3);
Complex c2(2);
Complex c3 = c1;

c2 = c3;
c1 += c2;

cout << c1 << “   “ << c2 << “   “ << c3 << “\n”;
```

copy constructor

assignment

The difference is that c3 is being defined and initialized, so a constructor is necessary;
c2 is already initialized

The add function

- Now suppose we want to define a function *add* that returns the sum of two complex numbers
  - the return type is Complex
- a first try could be

```
Complex add(Complex a, Complex b)
{
    Complex z(a.real() + b.real(), a.imaginary() + b.imaginary());
    return z;
}
```

This is not very good programming style for many reasons!

can you list them?

Using the add

- Let's see what happens when we use our *add*

```
Complex c1(1,2),c2(2,3),c3;

c3 = add(c1,c2);
```

the temporary object is destroyed

the temporary object is assigned to c3 calling the assignment operator

```
Complex add(Complex a, Complex b)
{
    Complex z(a.real() + b.real(), a.imaginary() + b.imaginary());
    return z;
}
```

temp is copied into a temporary object

z is constructed

c1 and c2 are copied (through the copy constructor), into a and b

7 function calls are involved!

(not considering real() and imaginary()) ...

First improvement

- Let's pass by const reference:

```
Complex c1(1,2),c2(2,3),c3;

c3 = add(c1,c2);
```

only the addresses are copied: much faster!

```
Complex add(const Complex& a, const Complex& b)
{
    Complex temp(a.real() + b.real(), a.imaginary() + b.imaginary());
    return temp;
}
```

We already saved 2 function calls!

notice that c1 and c2 cannot be modified anyway...

Temporaries

- Why the compiler builds a temporary?
  - because he doesn't know what we are going to do with that object
  - consider the following expression:

```
Complex c1(1,2), c2(2,3), c3(0,0);

c3 += add(c1,c2);
```

first, the add is called

second, operator+= is called

operator+=(const Complex &c);

So, the compiler is forced to build a temporary object of type Complex and pass it to operator+= by reference, which will be destroyed soon after operator+= completes

Temporary objects

- A temporary should always be constant!
  - otherwise we could write things like:

```
add(c1,c2) += c1;
```

It is pure non-sense!

To avoid this let us, return a const

```
const Complex add(const Complex& a, const Complex& b)
{
    Complex temp(a.real() + b.real(), a.imaginary() + b.imaginary());
    return temp;
}
```

Returning a const

- Thus, now it should be clear why sometime we need to return a const object
  - the previous example was trivial, but when things get complicated, anything can happen
  - by using a const object, we avoid stupid errors like modifying a temporary object
  - the compiler will complain if we try to modify a const object!

More on *add*

- Ok, that was clean, but did not save much
- However, there is a way to save on another copy constructor

```
const Complex add(const Complex& a, const Complex& b)
{
    return Complex(a.real() + b.real(), a.imaginary() + b.imaginary());
}
```

It means: create a temporary object and return it

Now we have 4 function calls:

add, temporary constructor, assignment, temporary destructor

More on copy constructors

- let's make an exercise.

```
class A {
   int ii;
public:
   A(int i) : ii(i) { cout << “A(int)\n”;}
   A(const A& a) { ii = a.ii; cout << “A(A&)\n”;}
};
```

```
int main()
{
    A a1(3);
    B b1(5);
    B b2(b1);
}
```

```
class B {
   int ii;
   A a;
public:
   B(int i) : ii(i), a(i+1) {cout << “B(int)\n”; }
};
```

```
A(int)
A(int)
B(int)
A(A&)
```

Changing the copy constructor

- We can change the behavior of B...

```
class A {
   int ii;
public:
   A(int i) : ii(i) { cout << “A(int)\n”;}
   A(const A& a) { ii = a.ii; cout << “A(A&)\n”;}
};
```

```
int main()
{
    A a1(3);
    B b1(5);
    B b2(b1);
}
```

```
class B {
   int ii;
   A a;
public:
   B(int i) : ii(i), a(i+1) {cout << “B(int)\n”; }
   B(B& b) : ii(b.ii), a(b.ii+1) {cout << “B(B&)\n”;}
};
```

```
A(int)
A(int)
B(int)
A(int)
B(B&)
```

Static

- *static* is another keyword that is overloaded with many meanings
- Here, we will discuss only one of them: how to build static class members
  - sometime, we would like to have a member that is common to all objects of a class
  - for doing this, we can use the *static* keyword

static members

- We would like to implement a counter that keeps track of the number of objects that are around
  - we could use a global variable, but it is not C++ style
  - we can use a static variable

```
class ManyObj {
   static int count;
   int index;
public:
   ManyObj();
   ~ManyObj();

   int getIndex();
   static int howMany();
};
```

static variable

```
static int ManyObj::count = 0;

ManyObj::ManyObj() { index = count++;}
ManyObj::~ManyObj() {count--;}
int ManyObj::getIndex() {return index;}
int ManyObj::howMany() {return count;}
```

static member function

static members

```
int main()
{
    ManyObj a, b, c, d;
    ManyObj *p = new ManyObj;
    ManyObj *p2 = 0;
    cout << “Index of p: “ << p->getIndex() << “\n”;
    {
        ManyObj a, b, c, d;
        p2 = new ManyObj;
        cout << “Number of objs: “ << ManyObj::howMany() << “\n”;
    }
    cout << “Number of objs: “ << ManyObj::howMany() << “\n”;
    delete p2; delete p;
    cout << “Number of objs: “ << ManyObj::howMany() << “\n”;
}
```

calling a static member function

```
Index of p: 4
Number of objs: 10
Number of objs: 6
Number of objs: 4
```

static members

- There is only *one copy* of the static variable for all the objects
- All the objects refer to this variable
- How to initialize a static member?
  - cannot be initialized in the class declaration
  - the compiler does not allocate space for the static member until it is initiliazed
  - So, the programmer of the class must define and initialize the static variable

Initialization

- It is usually done in the *.cpp* file where the class is implemented

```
static int ManyObj::count = 0;

ManyObj::ManyObj() { index = count++;}
ManyObj::~ManyObj() {count--;}
int ManyObj::getIndex() {return index;}
int ManyObj::howMany() {return count;}
```

initialization of the static member

There is a famous problem with static members, known as the *static initialization order failure*

we will not study it here. See the book on page 432

Copy constructors and static members

- What happens if the copy constructor is called?

```
void func(ManyObj a)
{
    …
}

void main()
{
    ManyObj a;
    func(a);
    cout << "How many: " << ManyObj::howMany() << "\n";
}
```

What is the output?

Solution in manyobj/

Again on copy constructors

- If we want to prevent passing by value we can *hide* the copy constructor
- You hide copy constructor by making it private
  - in this way the user of the class cannot call it

```
class ManyObj {
    static int count;
    int index;
    ManyObj(ManyObj &);
public:
    ManyObj();
    ~ManyObj();

   static int howMany();
};
```

```
void func(ManyObj a)
{
    …
}

void main()
{
    ManyObj a;
    func(a);        //ERROR! No copy constructor
}
```

Singleton object

- A singleton is an object that can exist in only one copy
  - we want to avoid that a user creates more than one of these objects
- We can make a singleton by combining static members and constructor hiding

```
class Singleton {
    static Singleton s;
    Singleton();
    Singleton(Singleton &);
public:
    static Singleton & instance();
};
```

Singleton object

- First, we hide both constructors, so no user can create a singleton object
  - we also hide assignment operator
- We define one singleton object as static
- To obtain the singleton object, users must invoke member instance();

```
Singleton &s = Singleton::instance(); // ok
Singleton s2 = Singleton::instance(); // ERROR! No copy constructor!
```

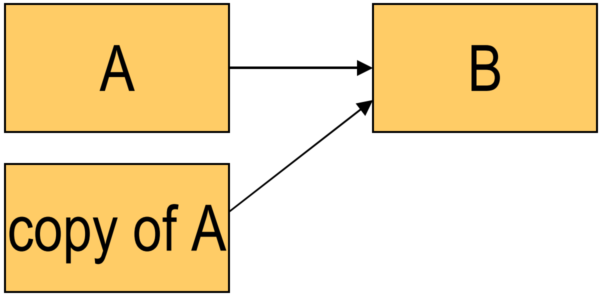
see oneobj/

Last thing on copy constructors

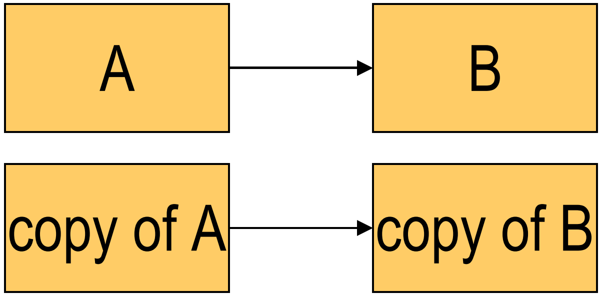
- If you are designing a class for a library, always think about what does it mean to copy an object of your class
  - a user could try to pass the object to a function by value, and obtain an inconsistent behavior
- For example, consider that your class contains pointers to objects of other classes
  - when you *clone* your object, do you need to copy the pointers or the pointed classes? Depends on the class!

Last on copy constructors

- First option (default):



Second option:



Quite different, don't you think?

How to copy an unknown object??

- to be discussed when we talk about inheritance...

Ownership

- Last argument introduce the problem of *ownership*
- A class can contain pointers to other objects;
  - suppose they were created dynamically (with *new*), so they are allocated on the heap
- At some point, your object is deallocated (destructed)
  - should your object destroy the other objects?
  - It depends on *ownership*: if your object is *owner* of the other objects, then it is responsible for destruction

Example

- Here the caller of the function is responsible for deleting the string:

```
string *getNewName()
{
    string *t = new string(…);
   …
   return t;
}

int main()
{
    string *s = getNewName();
    …
    delete s;
}
```

Inside the function call, the function is the *owner*

After the return, the main function becomes the *owner*

Ownership is very important in order to avoid memory leaks

See owner/

- Notice that compiler does not know anything about ownership!
  - It’s the logic of the program that says who is the owner of the object each time
- The general rule that I apply is
  - If an object creates another object, he is responsible for destruction
  - of course there are zillion of exceptions to this rule…
  - pay attention to ownership!!

Inlines

- Performance is important
  - if C++ programs were not fast, probably nobody would use it (too complex!)
  - Instead, by knowing C++ mechanisms in depth, it is possible to optimize a lot
  - One possible optimizing feature is *inline* function

Complex inlines

```
class Complex {
  double real_;
  double imaginary_;
public:
   Complex();                                  // default constructor
   Complex(const Complex& c);                  // copy constructor
   Complex(double a, double b = 0);            // constructor
   ~Complex();                                 // destructor

   inline double real() const {return real_;}
   inline double imaginary() const {return imaginary;}
   inline double module() const {return real_*real_ + imaginary_*imaginary_;}
   Complex& operator =(const Complex& a);      // assignment operator
   Complex& operator+=(const Complex& a);      // sum operator
   Complex& operator-=(const Complex& a));     // sub operator
};
```

What is inlining

- when the compiler sees inline, tries to substitute the function call with the actual code
  - in the complex class, the compiler substitute a function call like real() with the member variable real_

```
Complex c1(2,3), c2(3,4), c3;

cout << c1.real() << “\n”
```

substituted by
c1.real_

we save a function call!

in C this was done through *macros*

macros are quite bad. Better to use the inlining!

again, the compiler is much better than the pre-compiler

Inline

- Of course, inline function must be defined in the header file
  - otherwise the compiler cannot see them and cannot make the substitution
  - sometime the compiler refuses to make inline functions

Excessive use of inlines

- People tend to use inlines a lot
  - first, by using inline you expose implementation details
  - second, you clog the interface that becomes less readable
  - Finally, listen to what D.Knuth said:

> *“Premature optimization*
> *is the source of all evil”*

So, first design and program, then test, then optimize and test again

Operator overloading

- After all, an operator is like a function
  - binary operator: takes two arguments
  - unary operator: takes one argument
- The syntax is the following:
  - Complex &operator+=(const Complex &c);
- Of course, if we apply operators to predefined types, the compiler does not insert a function call

```
int a = 0;

a += 4;
```

```
Complex a = 0;

a += 5;
```

To be member or not to be...

- In general, operators that modify the object (like ++, +=, --, etc…) should be member
- Operators that do not modify the object (like +, -, etc,) should not be member, but friend functions
- Let’s write operator+ for complex (see complex/)
- Not all operators can be overloaded
  – we cannot “invent” new operators, we can only overload existing ones
  – we cannot change number of arguments
  – we cannot change precedence
  – *. (dot)* cannot be overloaded

Strange operators

- You can overload
  - *new* and *delete*
    - used to build custom memory allocate strategies
  - operator[]
    - for example, in vector<>…
  - operator,
    - not very useful, it's there for consistency. You can write funny programs!
  - operator->
    - used to make *smart pointers*!!

How to overload operator []

- the prototype is the following:

```
class A {
   …
public:
   A& operator[](int index);
};
```

exercise: add operator [] to you List class

the operator must never go out of range

How to overload new and delete

- This is the way to do it:

```
class A {
   …
public:
    void* operator new(size_t size);
    void operator delete(void *);
};
```

You can also overload the global version of new and delete

How to overload * and ->

- This is the prototype

```
class Iter {
…
public:
   Obj operator*() const;
  Obj *operator->() const;
};
```

Why should I overload operator*() ?

to implement iterators!

Why should I overload operator->() ?

to implement smart pointers

Example

- A simple iterator for stack
  - It is a *forward iterator*

Exercise

- Build a Iterator class for your list of strings
  - You can define an object of type Iterator, that can point to objects inside the List container
  - Write operator++() the Iterator
  - Write operator* for Iterator that de-reference the pointed object;
  - Compare your implementation with the list<> container of the std library
  - Try to call foreach() on your container. What happens?

A more complex exercise

- Define a SmartPointer for objects of class A
  - This pointer must always be initialized
  - When no object points to the object, the object is automatically destroyed

```
class A { … };
class SSP { … };

SSP p1 = A::getNew(); // p1 points to a new obj
SSP p2 = p1;          // p1 and p2 point to obj

p1 = 0;               // only p2 points to obj
p2 = 0;               // destroy the object
```

Hint:

you should create a static repository…

This will become a template soon!