deep se
dependable evolvable pervasive software engineering group

Copy Constructors and Destructors

Federica Filippini - Marco Lattuada - Danilo Ardagna

Politecnico di Milano

federica.filippini@polimi.it – marco.lattuada@polimi.it – danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- Copy and Assignment Constructors
- Destructor
- Default, Delete
- Copy control and resource management
- Implicit Class-Type conversions

Copy Control

- Each class defines a new type and defines the operations that objects of that type can perform
- Classes can control **what happens when objects of the class type are copied, assigned, or destroyed**
- Classes control these actions through special member functions
  - Copy constructor
  - Assignment operator
  - Destructor
- Collectively, we'll refer to these operations as **copy control**
- Move semantic (since C++11, in APSC course)

Copy Control

- If a class does not define all the copy-control members, the compiler **automatically defines** the missing operations
- As a result, many classes can ignore copy control
- For some classes, relying on the default definitions leads to disaster

Frequently, the hardest part of implementing copy-control operations is recognizing **when we need** to define them in the first place

Class example: Sales_data

```
class Sales_data {
public:
    /* Getters and Setters */
    Sales_data() :
        bookNo(""),
        units_sold(0),
       revenue(0.0)
    {}
private:
    std::string bookNo;
    unsigned units_sold;
    double revenue;
};
```

The Copy-Assignment Operator

- Just as a class controls how objects of that class are initialized, it also controls how objects of its class are assigned

```
Sales_data trans, accum;
trans = accum;  // uses the Sales_data copy-assignment operator
```

Sales_data::operator=

The Synthesized Copy-Assignment Operator

- Just as it does for the copy constructor, the compiler generates a **synthesized copy-assignment operator** for a class if the class does not define its own
- If all the members can be copy-assigned…
  - …each non-static member of the right-hand object is assigned to the corresponding member of the left-hand object using the copy-assignment operator for the type of that member
- If some members cannot be copy-assigned…
  - …the synthesized copy-assignment is **unavailable** (implicitly deleted)
- Array members are assigned by assigning each element of the array
- The synthesized copy-assignment operator **returns a reference to its left-hand object**

The Synthesized Copy-Assignment Operator

```
// equivalent to
Sales_data& Sales_data::operator=(const Sales_data &rhs)
{
   bookNo = rhs.bookNo;          // calls the string::operator=
   units_sold = rhs.units_sold;  // uses the built-in int assignment
   revenue = rhs.revenue;        // uses the built-in double assignment
   return *this;                 // return a reference to this object
}
```

Crucial to guarantee the expected behavior when we have chains of assignments:

```
Sales_data s1;
Sales_data s2;
Sales_data s3;
(s1 = s2) = s3;
```

We can define our own version!

Example: when we copy a Sales_data object, we may want to **copy the number of units sold and the revenue**, but to **keep the book number unchanged**

```
Sales_data& Sales_data::operator=(const Sales_data &rhs)
{
   bookNo = rhs.bookNo;
   units_sold = rhs.units_sold;
   revenue = rhs.revenue;
   return *this;
}
```

This should always be the same!

Copy Initialization

```
string dots(10, '.');                    // direct initialization
string s(dots);                          // direct initialization
string s2 = dots;                        // copy initialization
string null_book = "9-999-99999-9";      // copy initialization
string nines = string(100, '9');         // copy initialization
```

- When we use direct initialization, we are asking the compiler to use ordinary function matching to select the constructor that best matches the arguments we provide
- When we use **copy initialization**, we are asking the compiler to copy the right-hand operand into the object being created, converting that operand if necessary

Copy Initialization

- Copy initialization ordinarily uses the **copy constructor**
- Copy initialization happens not only when we **define** variables using an =, but also when we:
  - Pass an object as an argument to a parameter of non-reference type
  - Return an object from a function that has a non-reference return type
  - Brace initialize the elements in an array or the members of an aggregate class
- Some class types also use copy initialization for the objects they allocate
  - The library containers copy initialize their elements when we initialize the container, or when we call an insert or push member

The Copy Constructor

- A constructor is the copy constructor **if its first parameter** is a **reference to the class type** and any additional parameters have default values

```
class Foo {
public:
        Foo();                          // default constructor
        Foo(const Foo&);                // copy constructor
        // ...
};
```

- The first parameter must be a reference type: almost always a **reference to const**, although we can define the copy constructor to take a **reference to non-const**

The Synthesized Copy Constructor

- When we do not define a copy constructor for a class, the compiler tries to synthesize one for us
  - Unlike the synthesized default constructor, a copy constructor is synthesized even if we define other constructors
- If all the members can be copied…
  - …the synthesized copy constructor **member-wise copies** the members of its argument into the object being created
- If some members cannot be copied…
  - …the synthesized copy constructor is **unavailable** (implicitly deleted)

The Synthesized Copy Constructor

- The type of each member determines how that member is copied
- Members of class type are copied by the copy constructor for that class
- Members of built-in type are copied directly
- Array members are copied by copying elements one by one

Example

- Equivalent copy constructor signature:

```
Sales_data(const Sales_data&);
```

- Equivalent copy constructor implementation:

```
Sales_data::Sales_data(const Sales_data &orig):
   bookNo(orig.bookNo),          // uses the string copy constructor
   units_sold(orig.units_sold),  // copies orig.units_sold
   revenue(orig.revenue)         // copies orig.revenue
   { }                           // empty body
```

We can define our own version!

Example: when we create a new Sales_data object by copy, we may want to **copy the number of units sold and the revenue**, but to **generate a new book number**

```
Sales_data (const Sales_data& orig):
        bookNo("9-999-99999-9"),
        units_sold(orig.units_sold),
        revenue(orig.revenue)
        {}
```

Of course, a class copy constructor and assignment operator should be coherent!

REMARK: Container elements are copies

- When we use an object to initialize a container, or insert an object into a container, a **copy** of that object value is placed in the container, not the object itself
- Just as when we pass an object to a non-reference parameter, there is **no relationship** between the element in the container and the object from which that value originated

Subsequent changes to the element in the container have no effect on the original object, and vice versa

DEMO

Destructor

What a Destructor does

- The destructor does whatever operations the class designer wishes to have executed after the last use of an object
  - Typically, the destructor frees resources an object allocated during its lifetime
- The destructor operates inversely to the constructors
- Just as a constructor has an initialization part and a function body, a destructor has a function body and a destruction part
- In a destructor, there is nothing akin to the constructor initializer list to control how members are destroyed
  - the destruction part is implicit

Constructor vs. Destructor

- **Constructors** initialize the non-static data members of an object and may do other work
  - Members are initialized before the function body is executed
  - Members are initialized in the **same order** as they appear in the class
- **Destructors** do whatever work is needed to free the resources used by an object and destroy the non-static data members of the object
  - The function body is executed first and then the members are destroyed
  - Members are destroyed in **reverse** order from the order in which they were initialized

What a Destructor does

- What happens when a member is destroyed depends on the type of the member:
  - Members of class type are destroyed by running the member's own destructor
  - The built-in types do not have destructors, so nothing is done to destroy members of built-in type

The Destructor – Syntax

- The destructor is a member function with the name of the class prefixed by a tilde (~)
- It has no return value and takes no parameters

- **Because it takes no parameters, it cannot be overloaded**
  - **There is always only one destructor for a given class**

```
class Foo {
public:
    ~Foo(); // destructor
    // ...
};
```

When a Destructor is called

- Called automatically whenever an object of its type is destroyed:
  - Variables are destroyed when they go out of scope
  - Members of an object are destroyed when the object of which they are a part is destroyed
  - Elements in a container—whether a library container or an array—are destroyed when the container is destroyed
  - Dynamically allocated objects are destroyed when the delete operator is applied to a pointer to the object
  - Temporary objects are destroyed at the end of the full expression in which the temporary was created
- Because destructors are run automatically, our programs can allocate resources and (usually) not worry about when those resources are released
  - Provided that delete is invoked when necessary

Example

```
{ // new scope
    // p and p2 are smart pointers pointing to dynamically allocated objects
    shared_ptr<Sales_data> p = make_shared<Sales_data>();
    auto p2 = p;
    Sales_data item(*p);        // copy constructor copies *p into item
    vector<Sales_data> vec;     // local object
    vec.push_back(*p2);         // copies the object to which p2 points
} // exit local scope; destructor called on p, p2, item, and vec
```

Note: destroying vec destroys the elements in vec and then the vector itself…
…and the same happens to shared pointers!

The Synthesized Destructor

- The compiler defines a **synthesized destructor** for any class that does not define its own destructor
- As with the copy constructor and the copy-assignment operator, for some classes, the default destructor **cannot be synthesized**

The Synthesized Destructor

- The members are automatically destroyed **after** the (empty) destructor body is run
- Members are destroyed as part of the implicit destruction phase that follows the destructor body
  - A destructor body executes in addition to the member-wise destruction that takes place as part of destroying an object

```
class Sales_data {
public:
    // no work to do other than destroying the members,
    // which happens automatically
    ~Sales_data() { }
    // other members as be
};
```

Better to define it as virtual!
virtual ~Sales_data () {}

Copy Control Default, Delete

The Rule of Three

- There are three basic operations to control copies of class objects:
  - copy constructor
  - copy-assignment operator
  - destructor
- There is **no requirement** that we define all these operations:
  - We can define one or two of them without having to define all of them
- Ordinarily these operations should be thought of as a **unit**:
  - In general, it is unusual to need one without needing to define them all

Classes that need Copy need Assignment, and vice-versa

- Although many classes need to define all (or none of) the copy-control members, some classes have work that needs to be done to copy or assign objects but has no need for the destructor
- **Example:** consider a class that gives each object its own, unique, serial number
  - Such a class would need a copy constructor to generate a new, distinct serial number for the object being created
  - That constructor would copy all the other data members from the given object
  - This class would also need its own copy-assignment operator to avoid assigning to the serial number of the left-hand object
  - This class would have no need for a destructor

Classes that need Copy need Assignment, and vice-versa

- This example gives rise to a second rule of thumb:
  - If a **class needs a copy constructor**, it almost surely **needs a copy-assignment** operator
  - And **vice-versa**: if the class needs an assignment operator, it almost surely needs a copy constructor as well

- Nevertheless, needing either the copy constructor or the copy-assignment operator does not (necessarily) indicate the need for a destructor

DEMO

file Box.h

```
class Box {
public:
    // constructor
    Box (double l, double b, double h);

    // copy constructor
    Box (const Box& b);

    // member functions
    double volume (void) const;
    static unsigned getcount (void) {return count;}
    unsigned getid (void) const {return id;}

    // assignment operator
    Box& operator= (const Box &);

private:
    double length, breadth, height;
    unsigned id;              // Identification Number
    static unsigned count; // Number of boxes
};
```

file Box.cpp

```
// initialization of the static variable
unsigned Box::count = 0;

// constructor
Box::Box (double l, double b, double h): length(l), breadth(b),
height(h)
{
    std::cout << "Constructing a box" << std::endl;
    count++;
    id = count;
}

// copy constructor
Box::Box (const Box& b): length(b.length), breadth(b.breadth),
height(b.height)
{
    std::cout << "Using copy constructor" << std::endl;
    count ++;
    id = count;
}
```

Classes that need Destructors need Copy and Assignment

- One rule of thumb to use when you decide whether a class needs to define its own versions of the copy-control members is to **decide first whether** the class **needs** a **destructor**

- Often, the need for a destructor is more obvious than the need for the copy constructor or assignment operator
  - If the class **needs a destructor**, it almost surely **needs a copy constructor** and **copy-assignment** operator as well

vector

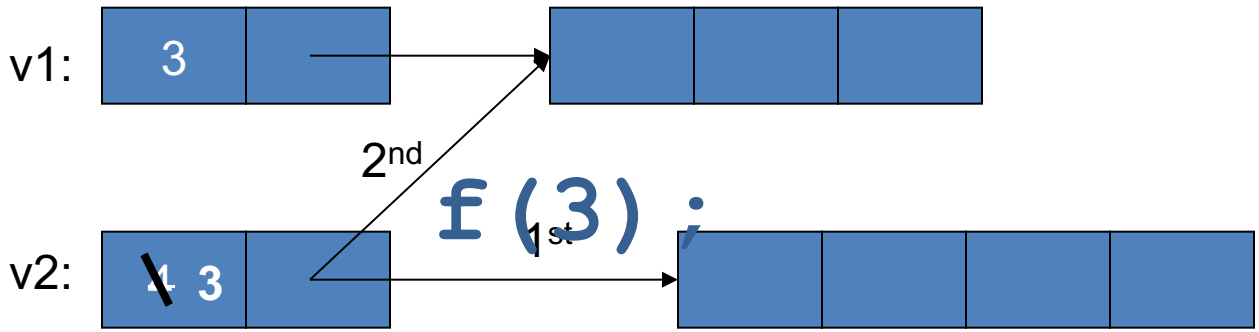
```
class vector {
    unsigned sz;          // the size
    double* elem;         // pointer to elements
public:
    vector(unsigned s): sz{s}, elem{new double[s]} { }

    ~vector() { delete[ ] elem; }

    /* Synthesized copy constructor
    vector(const & vector other) :
        sz{other.sz},
        elem{other.elem} { }
    */
    /* Other methods */
    …
};
```

Vector Synthesized Copy-Assignment Operator

```
void f(int n)
{
     vector v1(n);
     vector v2(4);
     v2 = v1;     // assignment:
                  // by default, a copy of a class copies its members
                  // so sz and elem are copied
}
```



Disaster when we leave f()!
v1's elements are deleted twice (by the destructor) and we have also a memory leakage

Overloading Copy-Assignment Operator

- The copy-assignment operator takes an argument of the same type as the class:

```
class Foo {
public:
        Foo& operator=(const Foo&); // assignment operator
        // ...
};
```

- **To be consistent** with assignment for the **built-in types**, assignment operators usually **return a reference** to their left-hand operand

Using = default

- Explicitly requires synthesis of default special method
  - Force synthesis of empty constructor
  - Prevent custom implementation
  - Improve readability

```
class Sales_data {
public:
    // copy control; use defaults
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // other members as before
};

Sales_data& Sales_data::operator=(const Sales_data&) = default;
```

Preventing Copies

- For some classes, there really is no sensible meaning for create a copy:
  - **copies or assignments** must be denied
- The **iostream** classes prevent copying to avoid letting multiple objects write to or read from the same IO buffer
- It might seem that we could prevent copies by not defining the copy-control members. However, this strategy doesn't work:
  - If our class doesn't define these operations, the compiler will synthesize them

Defining a Function as Deleted

- Under C++ 11, we can prevent copies by defining the copy constructor and copy-assignment operator as **deleted functions**
  - Syntax: its parameter list is followed by `= delete`
- Synthesized versions of deleted functions are not generated

```
struct NoCopy {
    NoCopy() = default;  // use the synthesized default constructor
    // disallow copy
    NoCopy(const NoCopy&) = delete;
    // disallow assignment
    NoCopy &operator=(const NoCopy&) = delete;
    ~NoCopy() = default; // use the synthesized destructor
    // other members
};
```

Copy Control and Resource Management

Copy Control and Resource Management

- Classes that manage resources that do not reside in the class must define the copy-control members
- In order to define these members, we first have to decide what copying an object of our type will mean. In general, we have two choices:
  - **like-a-value:** the class behaves like a value
  - **like-a-pointer**: the class behaves like a pointer

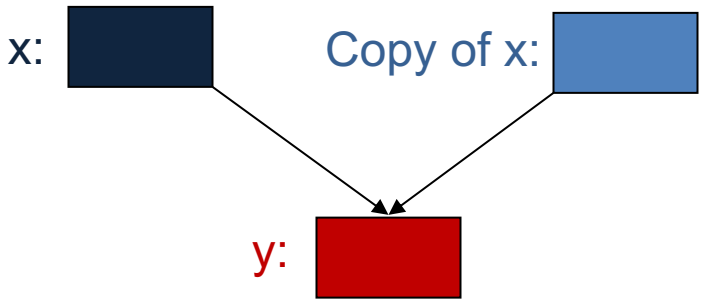
Copy Control and Resource Management

- **Like-a-value** classes have their own state
  - When we copy a like-a-value object, the copy and the original are independent of each other
  - Changes made to the copy have no effect on the original, and vice versa

- **Like-a-pointer** classes that act like pointers share part of the state
  - When we copy objects of such classes, the copy and the original use the same underlying data
  - Changes made to the copy also change the original, and vice versa

Copy + like-a-pointer / like-a-value semantic

- **Shallow copy → Like-a-pointer**
  - copy only a pointer so that the two pointers now refer to the same object
  - What pointers and references do
- **Deep copy → Like-a-value**
  - what the pointer points to so that the two pointers now each refer to a distinct object
  - What vector, string, etc. do
  - Requires copy constructors and copy assignments for container classes
  - Must copy “all the way down” if there are more levels in the object



Shallow copy / like-a-pointer

Deep copy / like-a-value

Copy Control and Resource Management

- Ordinarily, classes copy members of built-in type (other than pointers) directly; such members are values and hence ordinarily ought to behave like values

- What we do when we copy the pointer member determines whether the class has like-a-value or like-a-pointer behavior

Example: like-a-pointer

```
class StrLPVector {
public:
        typedef std::vector<std::string>::size_type size_type;

        StrLPVector();
        StrLPVector(std::initializer_list<std::string> il);

        size_type size() const { return data->size(); }
        bool empty() const { return data->empty(); }

        // add and remove elements
        void push_back(const std::string &t) {data->push_back(t);}
        void pop_back() {data->pop_back();};

        // element access
        std::string& front() {return data->front();}
        std::string& back() {return data->back();}

private:
        std::shared_ptr<std::vector<std::string>> data;
        // write msg if data[i] isn't valid
};
```

Example: like-a-value

```
class StrLPVector {
public:
        typedef std::vector<std::string>::size_type size_type;

        StrLPVector();
        StrLPVector(std::initializer_list<std::string> il);

        size_type size() const { return data.size(); }
        bool empty() const { return data.empty(); }

        // add and remove elements
        void push_back(const std::string &t) {data.push_back(t);}
        void pop_back() {data.pop_back();};

        // element access
        std::string& front() {return data.front();}
        std::string& back() {return data.back();}

private:
        std::vector<std::string> data;
        // write msg if data[i] isn't valid
};
```

Implicit Class-Type Conversions

Implicit Class-Type Conversions

- C++ defines several automatic conversions among the built-in types
- Classes can define implicit conversions as well
  - Every constructor that can be called with a **single argument** defines an implicit conversion to a class type
  - Such constructors are sometimes referred to as **converting constructors**
    - Define an implicit conversion from the constructor's parameter type to the class type

Example: MatlabVector

```
class MatlabVector {

    vector<double> elem;

public:
    MatlabVector(unsigned sz): elem(sz, 0.) {}
    MatlabVector() = default;

    double& operator[](unsigned n);
    size_t size() const; // return number of elements

    MatlabVector operator+(const MatlabVector& other) const;
    MatlabVector operator*(double scalar) const;
};
```

Implicit Class-Type Conversions

- The `MatlabVector` constructor that take an unsigned `sz` defines implicit conversions from that type to `MatlabVector`
  - We can use an unsigned where an object of type `MatlabVector` is expected

```
MatlabVector v1 = 7;      // v1 has 7 elements, each with the value 0

void do_something(MatlabVector v)
do_something(7);      // call do_something() with a vector of 7
                      // elements
```

- This is very error-prone
  - Unless, of course, that's what we wanted

DEMO

Suppressing Implicit Conversions Defined by Constructors

- A solution: we can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as **explicit**

```
class MatlabVector  {
    // ...
    public:
    explicit MatlabVector(unsigned sz): elem(sz, 0.) {}
    // ...
};

MatlabVector v1 = 7;      // error: no implicit conversion from unsigned

void do_something(MatlabVector v);

do_something(7);   // error: no implicit conversion from unsigned
```

`explicit` Constructors Can Be Used Only for Direct Initialization

- One context in which implicit conversions happen is when we use the copy form of initialization
- We cannot use an explicit constructor with this form of initialization; we must use direct initialization

```
MatlabVector v1(10);      // ok: direct initialization

MatlabVector v2 = 10;     // error: cannot use the copy form of
                          // initialization with an explicit
                          // constructor
```

Explicitly Using Constructors for Conversions

- Although the compiler will not use an explicit constructor for an implicit conversion, we can use such constructors explicitly to force a conversion

```
MatlabVector v1(10);
for (size_t j=0; j<v1.size(); ++j)
        v1[j] = j;

v1 += 10;       // error: no explicit conversion from unsigned to MatlabVector

v1 += MatlabVector(10);      // ok: the argument is an explicitly
                             // constructed MatlabVector object
```

References

- Lippman Chapter 13

Readings

Implicit Class-Type Conversions

Consider class `Sales_data`, with its constructor

```
Sales_data (const std::string&);
```

Even if it is not explicit, only one implicit conversion is allowed:

```
// error: requires two user-defined conversions:
//  (1) convert "9-999-99999-9" to string
//  (2) convert that (temporary) string to Sales_data
item += "9-999-99999-9";

// ok: explicit conversion to string, implicit conversion to Sales_data
item += string("9-999-99999-9");

// ok: implicit conversion to string, explicit conversion to Sales_data
item += Sales_data("9-999-99999-9");
```