High Level Programming

Copy Control

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

Introduction

- When a C++ class is defined, we implicitly or explicitly specify what happens when the class is
  - Copied, moved, assigned, and destroyed
- A class controls these operations with five special class member functions
  - They are referred to as “copy control” functions
  - We can write them explicitly
  - If we do not write them, the compiler creates them **automatically**
    - There are cases in which relying on the default definitions may lead to **disaster**
    - Thus, we need to learn how to define them

Introduction

- Copy control is performed by
  - Copy and Move constructors
    - Define the behavior when an object is **initialized** from another object
  - Copy and Move Assignment Operators
    - Define the behavior when we **assign** an object to another object
  - Destructor
    - Defines the behavior when an object **ceases** to exist

Beyond the standard constructor

Already analyzed in Unit 03

Copy Constructor

- A copy constructor is a special constructor that allows the **definition** of an object **through** a **copy** of an existing object of the same class
  - There may be multiple copy constructors
  - Given a class C, copy constructors have
    - The **same name** of the class
    - An **argument** of type **C&** or **const C&** (preferred)
    - Possibly, additional parameters with default values

```
class Foo {
  public:
    Foo ();             // Default constructor
    Foo (const Foo&);   // Copy constructor
}
```

Copy Constructor

- The copy constructor
  - Is called by the compiler whenever an object is defined through a copy
  - By default copies all members of its argument into the object being created
  - Can refer directly to any private data of the object that must be copied into the current one

```
Rectangle::Rectangle(const Rectangle &to_copy) {
  this->m_width = to_copy.m_width;
  this->m_length = to_copy.m_length;
}
```

Pointer to the current instance

Parameter

Private data

Example

```
class Class {
  public:
    Class (const char *str);
    ~Class();
  private:
    char *str;
}
Class::Class (const char *s) {
  str = new char[strlen(s)+1];
  strcpy(str,s);
}
Class::~Class(){
  delete[] str;
}
```

Constructor & Destructor

Constructor

Destructor

#include <cstring>

Synthesized copy constructors

The **synthetized** copy constructor copies each non static member from the given object to the created object. **Do we need to copy the pointer or duplicate the string?**

Compiler-defined copy constructor

```
Class::Class (const Class &another) {
  str = another.str;
}
```

Example

```
class Class {
  public:
    Class (const char *str);
    ~Class();
  private:
    char *str;
}
Class::Class (const char *s) {
  str = new char[strlen(s)+1];
  strcpy(str,s);
}
Class::~Class(){
  delete[] str;
}
```

Constructor & Destructor

Constructor

#include <cstring>

Destructor

User-defined copy constructors

We may want to duplicate the string

```
Class::Class (const Class &another) {
  str = new char[strlen(another.str)+1];
  strcpy(str,another.str);
}
```

User-defined copy constructor

Examples

- It is now possible to better understand the difference between
  - Direct initialization and copy initialization

Activation of the copy constructors

```
// Direct initialization
string s1(10,'.');
string s2(s1);

// Copy initialization
string s3 = s1;
string s4 = "1234567890";
string s5 = string (100, '9');
string s6;
s6 = s1;
```

Standard constructor:
The compiler calls the function that best matches the arguments

Copy constructor:
The compiler copies the right-hand operand into the object being created

This is not a constructor (activated only when the object is created) but an **assignment**

Copy assignment operator

- If the
  - Copy control is called when object are **copied at initialization**
  - Copy assignment operator is called when objects are **assigned**

```
my_class c1, c2;
...
c2 = c1;


class sales myc1, myc2;.
...
myc1 = myc2;
```

Use the my_class copy assignment operator
Either the implicitly or the user-defined one

Copy assignment operator

- The copy assignment operator controls how objects are assigned
  - Given a class C, assignment operators have
    - The name **operator=**
    - An argument of type **C&** or **const C&** (preferred)
    - A return type (usually a C&)
  - The compiler generates a synthesized copy assignment constructor if the class does not define one

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

Examples

```
class sales {
  public:
    sales (const sales&);
    sales& operator= (const saless=&);
  private:
    std::string number;
    int sold = 0;
    double revenue = 0.0;
}
```

Copy Constructor & Assignment

Synthesized copy assignment

Equivalent to the synthesized copy constructor

```
sales::sales (const sales &orig):
  number(orig.number),
  sold(orig.sold),
  revenue(orig.revenue)
  {  }
sales& sales::operator= (const sales &orig) {
  number = orig.number;
  sold = orig.sold;
  revenue = orig.revenue;
  return *this;
}
```

Empty body

Equivalent to the synthesized copy assignment

Destructor

Introduced in Unit 03

- The destructor reverse the operations done by the constructors
  - Variables are destroyed when they go out of scope
  - Member of an object are destroyed when the object to which they belong to is destroyed
  - Elements is a container are destroyed when the container is detroyed
  - Dynamically allocated objects are destroyed when **delete** is called
  - Temporary objects are destroyed at the end of the expression in which they were temporary created

Introduced in Unit 03

Destructor

- The destructor do whatever is need to reverse done by the constructors
  - Given a class C, the destructor has
    - The name **~C**
    - No argument (does it **cannot** be overloaded)
  - It is called automatically whenever an object is destroyed

```
class Foo {
  public:
    ~Foo ();
}
```

Examples

Activation of the destructor

```
// New scope
{
my_class *p1 = new my_class;         // p1 is a standard ptr
auto p2 = make_shared<my_class>();// p2 is a shared ptr
my_class item(*p1);                  // Constructor copy
                                     //   p1 into item
vector<my_class> v;                  // Local object
v.push_back(*p2);                    // Copy the object to which
                                     //   p2 points
delete p1;                           // Destrutor called on
                                     //   the object pointed by p1
}
// Scope ends
// Destructor called on item, p2, and v
// Destroying p2 decrements its counter; if it goes to zero,
//   the object is free
// Destroying v destroys the element in v
```

The "rule of three"

- If a class requires
  - A user-defined copy constructor
  - A user-defined copy assignment operator
  - A user-defined destructor

  it almost certainly requires all three
- Explanation
  - A user-defined copy constructor (destructor) usually implies some custom setup (cleanup) logic which needs to be executed by copy assignment and vice-versa

Move semantic

- Copy constructor and copy assignment follow a copy semantics
  - There are cases in which the object is immediately **destroyed** after it is copied
    - In those cases we incur in unnecessary and unwanted overhead
  - In those cases **moving** instead of copying may enhance performance
    - C++11 introduced the "move semantic"
    - Move operators typically "steal" resources
      - They do not usually allocate resources
      - They do not ordinarily throw exceptions

Move semantic

- To support move C++11 introduced a new kind of reference, i.e., a **rvalue** reference
- Generally speaking
  - **lvalue** expressions
    - Can stand on the left-hand side of an expression
    - Refer to an object's identity
    - Have persistent state
  - **rvalue** expressions refer to an object's value
    - Are either literal or temporary objects create in the course of evaluating expressions
    - An rvalue reference is obtained by using && rather than &

In C lvalue stands on the left-hand side of assignments; rvalue could not

Examples

```
int i = 5;           // rvalue = i, lvalue = 42
                     // The rvalue is just another
                     // name for the object

int &&r1 = 42;       // bind an rvalue to a constant
                     // OK, because the constant is
                     // an rvalue

int &&r2 = i * 10;   // OK as before
                     // i*10 is an rvalue

int &&r3 = i;        // Error: We cannot bind an
                     // rvalue to a variable i
                     // which is an lvalue
```

Move constructor

- A move constructor is typically called when an object is **initialized** from an **rvalue reference** of the same type
  - Given a class C, the move constructor has
    - The name **C**
    - An argument of type **C&&**
    - The **noexcept** keyword added to indicate that the constructor never throws an exception

```
class Foo {
  public:
     Foo (Foo&&) noexcept;
}
Foo::Foo (Foo&&) noexcept : { ... }
```

Examples

- We cannot bind an rvalue to an lvalue directly

```
int &&r = i;    // Error
```

- However, we can cast an lvalue to its corresponding rvalue
  - The **utility** header includes the function **move**
  - The function **move** can be used to convert an **lvalue** to an **rvalue** reference

```
int &&r = std::move(i);        // OK
```

Examples

```
struct X {
  int i;
  std::string s;
}
struct Y {
  X mem;
}

X x1;
Y y1;
...
X x2 = std::move(x1);
Y y2 = std::move(y1);
```

String has its own move constructor

Activation of the move operator

Y has a synthesized move constructor

x1 and y1 are variable, i.e. lvalue

Calls the synthesized move constructor

Examples

- For a class type C and objects a, b, the move constructor is invoked on

Activation of the move operator

```
C a(std::move(b));
```

Direct initialization

```
f(std::move(a));
```

Argument passing to a function

```
C f(C p) {
  ...
  return a;
}
```

Function return

Examples

```
class A {
  A(const A& other);
  A(A&& other);
};

int main() {
  A a1;

  A a2(a1);
  A a3(std::move(a1));
}
```

Copy constructor

Move constructor

Calls copy constructor

Calls move constructor

Move assignment

- A move assignment is typically called if an object appears on the **left-hand** side of an assignment with a **rvalue reference** on the right-hand side
  - Given a class C, the destructor has
    - The name **operator=** of type **C&**
    - An argument of type **C&&**
    - The **noexcept** keyword added to indicate that the constructor never throws an exception

```
class Foo {
  public:
    Foo& operator=(Foo&&) noexcept;
}
Foo& &Foo::operator=(Foo&& in) noexcept { ... }
```

Examples

```
class A {
    A();
    A(const A&);
    A(A&&) noexcept;
    A& operator=(const A&);
    A& operator=(A&&) noexcept;
};

int main() {
    A a1;

    A a2 = a1;
    Class a3 = std::move(a1);
    a3 = a2;
    a2 = std::move(a3);
}
```

Calls copy constructor

Calls move constructor

Calls copy assignment

Calls move assignment operator

Examples

```
class A {
  unsigned capacity;
  int* memory;

  A(unsigned capacity): capacity(capacity),
  memory(new int[capacity]) { }

  A(A&& other) noexcept :capacity(other.capacity),
  memory(other.memory) {
    other.capacity = 0;
    other.memory = nullptr;
  }

  ~A() { delete[] memory; }

  A& operator=(A&& other) noexcept {
    if (this == &other)
      return *this;

    delete[] memory;
    capacity = other.capacity;
    memory = other.memory;
    other.capacity = 0;
    other.memory = nullptr;
    return *this;
  }
};
```

Constructor

Move constructor

Destructor

Move assignment operator

The "rule of five"

- The presence of a user-defined copy constructor or copy assignment operator or destructor prevents the implicit definition of the move constructor and move assignment operator
- As a consequence, if a class follows the rule of three, it must define all five special member functions
  - Not adhering to the rule of five usually does not lead to incorrect code
  - However, many optimization opportunities may be inaccessible to the compiler if no move operations are defined

Summary

- The constructor is called when objects are created
- The copy constructor is called when objects are created (assigned) from existing objects
- The copy assignment operator is called when objects are assigned (it appearrs as lvalue)
- The destructor is called to desrtroy the objects created by the constructors
- A move constructor is called when objects are initialized from an rvalue reference
- A move assignment operator is called when objects (lvalue) are assigned from an rvalue reference

Exercise

- Which copy control functions are called in the following code snippet?

```
class C {
...
};

int main() {
  C e1, e2;
  e2 = e1;
  C *e3 = new C;
  e2 = *e3;
  return 0;
}
```

Exercise

- Which copy control functions are called in the following code snippet?

```
class C {
...
};

int main() {
  C e1, e2;         // Line 1: Constructor: 2 times
  e2 = e1;          // Line 2: Copy Assignment Operator
  C *e3 = new C;    // Line 3: Constructor (from new)
  e2 = *e3;         // Line 4: Copy Assignment Operator
  return 0;         // Line 5: Destructor: 2 times
}
```

e3 is not destroyed: Dynamically allocated objects are destroyed when delete is called

Exercise

- Which copy control functions are called in the following code snippet?

```
class C {
...
};

int main() {
  C e1, *e2;
  C e3 = *new C;
  C *e4 = new C[10];
  e1 = e3;
  e2 = e4;
  e1 = (std::move(e3));
  e2 = (std::move(e4));
  return 0;
}
```

Exercise

- Which copy control functions are called in the following code snippet?

```
class C {
...
};

int main() {
  C e1, *e2;                  // Line 1: Constructor e1 (e2=pointer)
  C e3 = *new C;              // Line 2: Constructor + Copy Con.
  C *e4 = new C[10];          // Line 3: Constructur: 10 times
  e1 = e3;                    // Line 4: Copy Assignement Operator
  e2 = e4;                    // Line 5: Nothing (e4=pointer)
  e1 = (std::move(e3));       // Line 6: Move Assignement Operator
  e2 = (std::move(e4));       // Line 7: Nothing (e4=pointer)
  return 0;                   // Line 8: Desctructor: 2 times
}
```

Contructor for new
Copy constructor for e3

e1 and e3
e2 and e4 are pointers