High Level Programming

Copy Control

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

This work is licensed under the license

CC BY-NC-ND 4.0

Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

BY: Credit must be given to you, the creator.

NC: Only noncommercial use of your work is permitted.
Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

ND: No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:
https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1

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

Simple Example to understand what copy constructor is:
For example, imagine you have a class called Car, and you create an object car1 of type Car. Now, if you want to make another Car object that's exactly like car1, you use the copy constructor.

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

When declaring a copy constructor, you typically want to pass the object to be copied by reference rather than by value. Passing by reference (Car&) is more efficient than passing by value (Car) because it avoids making a copy of the object being passed, especially for large objects.
Passing by reference allows the copy constructor to access the original object directly, without the overhead of creating a new copy of the object.

In the copy constructor declaration, you might see const Car& instead of just Car&. Adding const indicates that the reference is to a constant object, meaning the copy constructor promises not to modify the original object

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

The arrow (->) is a C++ operator used to access members of a class or structure through a pointer. It is essentially a shorthand notation for dereferencing a pointer and accessing a member of the object being pointed to.

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

This line assigns the value of the m_width member variable of the to_copy object to the m_width member variable of the current object (referred to by this).

Similarly, this line assigns the value of the m_length member variable of the to_copy object to the m_length member variable of the current object.

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

dynamic mem allocation

Destructor

dynamically alloc mem is deleted

Destructors are denoted with ~

Synthesized copy constructors

When implementing a copy constructor for a class that manages dynamically allocated resources, such as char* in this case, it's crucial to perform a deep copy. This involves allocating new memory and copying the contents of the source object's dynamically allocated memory into the newly allocated memory.

#include <cstring>

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

User-defined copy constructors

Constructor

#include <cstring>

Memory Allocation:
new char[strlen(str) + 1] dynamically allocates memory on the heap for storing a string. strlen(str) calculates the length of the input string str, and +1 is added to account for the null terminator ('\0') required at the end of the string.
The result is a pointer to the first character of the allocated memory block, which is assigned to the pointer variable m_string.
String Copying:
strcpy(m_string, str) copies the content of the input string str to the dynamically allocated memory block pointed to by m_string.
This function iterates through each character of the source string str and copies it to the destination string m_string, including the null terminator.

Destructor

The destructor deallocates the dynamically allocated memory to prevent memory leaks. It is responsible for releasing any resources acquired during the object's lifetime.
The correct destructor implementation provided in your code snippet Class::~Class() is appropriate for deallocating the dynamically allocated memory pointed to by str.

We may want to duplicate the string

User-defined copy constructor

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

When implementing a copy constructor for a class that manages dynamically allocated resources, such as char* in this case, it's crucial to perform a deep copy. This involves allocating new memory and copying the contents of the source object's dynamically allocated memory into the newly allocated memory.

Examples

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

Activation of the copy constructors

Direct initialization involves calling a constructor explicitly with a set of arguments enclosed in parentheses.
Example: string s1(10, '.');
In direct initialization, the compiler calls the constructor that best matches the provided arguments.
The copy constructor is not typically invoked during direct initialization, as it's called only when an object is being created as a copy of another object.

```
// 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;
```

Copy initialization involves initializing an object using the = operator with another object or a value of compatible type.
Example: string s3 = s1;
In copy initialization, the compiler invokes the copy constructor to create a new object by copying the contents of the right-hand operand into the object being created.
Copy initialization can also occur when creating a temporary object to initialize another object, such as string s5 = string(100, '9');

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**

The copy assignment operator is called when an already initialized object is assigned a new value from

Copy assignment operator

Copy Assignment Operator:
The copy assignment operator (operator=) is a special member function in a class that defines how one object can be assigned the value of another object of the same type.
It is used when objects are assigned to each other using the assignment operator (=).
Example: c2 = c1; or myc1 = myc2;

- If the

The copy assignment operator is invoked when an object is already initialized, and its value is being replaced with the value of another object.
It is used to perform a shallow copy or deep copy, depending on the requirements of the class and the semantics desired by the programmer.

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

The copy assignment operator is called when an already initialized object is assigned a new value from another existing object.

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

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

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

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;
}
```

The copy constructor creates a new sales object by copying the data members from another sales object passed as a parameter.
It initializes the new object with the same values as the original object.
The member initialization list (number(orig.number), sold(orig.sold), revenue(orig.revenue)) initializes the data members number, sold, and revenue of the new object with the corresponding values from the original object orig.

Empty body

Equivalent to the synthesized copy assignment

The copy assignment operator (operator=) defines how one sales object can be assigned the value of another sales object.
It copies the data members from the right-hand operand (orig) to the left-hand operand (*this).
Each data member (number, sold, and revenue) of the current object (*this) is assigned the corresponding value from the orig object.
The assignment operator returns a reference to the current object (*this) to allow chaining of assignment operations (sales1 = sales2 = sales3;).

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
```

This line is creating a new instance of my_class on the heap and storing the address of the newly created object in the pointer p1.

A shared pointer is a smart pointer that retains shared ownership of an object through a pointer. Multiple shared pointers can own the same object, and the object is destroyed when the last shared pointer is destroyed.

This line is creating a new instance of my_class on the heap using a shared pointer.

This line is creating a new instance of my_class on the stack by copying the object pointed to by p1. This is done using the copy constructor of my_class.

This line is creating a vector v that can store objects of type my_class.

This line is adding a copy of the object pointed to by p2 to the end of the vector v. This is done using the copy constructor of my_class.

delete p1; - This line is deleting the object pointed to by p1 from the heap. This calls the destructor of my_class for the object pointed to by p1.

The destruction of p2 decrements its reference count, and if the reference count goes to zero (which it does in this case, as there are no other shared pointers to the object), the object it points to is destroyed. The destruction of v calls the destructor for each of its elements.

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 semantics, introduced in C++11, are designed to efficiently transfer resources (like dynamically allocated memory) from one object to another, typically in cases where creating a copy is unnecessary or expensive.

Before move semantics, C++ primarily relied on copy semantics. When you pass objects to functions or return them from functions, copies are often made.

```
std::string func() {
    std::string str = "Hello";
    return str; // A copy of str is made
}

std::string s = func();

// Here str is copied when returned from the function
func(), which can be costly, especially for large strings.
```

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

Lvalues:

An lvalue refers to an object that has a name and can appear on the left-hand side of an assignment expression.
Lvalues represent objects that have a persistent identity and can be referenced or modified.
Examples of lvalues include variables, objects, and elements of arrays.
int x = 5; // 'x' is an lvalue
int arr[5]; // 'arr' is an lvalue

An rvalue refers to an object that does not have a name and typically appears on the right-hand side of an assignment expression.
Rvalues represent temporary values or temporary objects that are not persistently stored in memory.
Examples of rvalues include literal values, temporary objects created during expressions, and the results of function calls.Like the str in the previous slide example which is returned.
Or these
int result = 3 + 4; // '3 + 4' is an rvalue
int* ptr = new int(10); // 'new int(10)' returns an rvalue

The distinction between lvalues and rvalues is based on the expression context.
Lvalues are objects that have a memory location and can be referenced or modified, while rvalues are temporary values or objects that don't have a persistent identity.
Lvalues are typically used to represent variables, objects, or memory locations, while rvalues are used to represent temporary values or results of expressions.

Examples

Rvalue references, denoted by &&, are a special type of reference introduced in C++11.
They can bind to temporary objects (rvalues) and are often used in move semantics to efficiently transfer resources from temporary objects.
Rvalue references are essential for enabling move semantics and improving performance in modern C++ code.

int&& rvref = 10; // '10' is an rvalue, 'rvref' is an rvalue reference

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

int i = 5;: Here, i is an lvalue because it has a name and represents a memory location where the value 5 is stored. The value 5 itself is an rvalue because it's a temporary value without a persistent identity.

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

int &&r1 = 42;: Here, r1 is an rvalue reference (denoted by &&) bound to the rvalue 42. Since 42 is an rvalue, it can be bound to an rvalue reference.

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

int &&r2 = i * 10;: The expression i * 10 evaluates to a temporary value (an rvalue), which is then bound to the rvalue reference r2. The result of i * 10 is an rvalue because it's a temporary value resulting from the expression.

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

int &&r3 = i;: This line causes a compilation error because r3 is an rvalue reference, but i is an lvalue. We cannot bind an rvalue reference to an lvalue directly. Rvalue references are typically used to bind to temporary values (rvalues), not to objects with persistent identities (lvalues).

Move constructor

Move constructors are special member functions in C++ that enable the efficient transfer of resources from one object to another.

- 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

noexcept is a C++ keyword used to specify that a function (including constructors, destructors, and assignment operators) does not throw exceptions.

```
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

Although you cannot directly bind an lvalue to an rvalue reference, you can use the std::move function to cast the lvalue to an rvalue reference.

The std::move function is provided by the <utility> header and is used to convert an lvalue to an rvalue reference.

  - 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

Y has a synthesized move constructor

x1 and y1 are variable, i.e. lvalue

Calls the synthesized move constructor

Activation of the move operator

Move Constructor Activation:

When you use std::move on an object, you're signaling to the compiler that you're willing to move its resources elsewhere.
Regular Variables (lvalues):

Before std::move, x1 and y1 are regular variables, also known as lvalues.
std::move Transforms Variables to rvalues:

After std::move, although x1 and y1 were initially lvalues, std::move transforms them into rvalues. This transformation essentially says, "I'm okay with this object being moved from."

Once std::move is applied, it triggers the move constructors for X and Y. If not explicitly defined, the compiler generates synthesized move constructors.
Move Constructors Move Resources:

The move constructors for X and Y move the resources from the source object (the one being moved from) to the destination object (the one being constructed or assigned to).

The move constructor for X is called with x1 as its argument. This means that the resources owned by x1 are transferred to x2, leaving x1 in a valid but unspecified state.

In summary, x2 and y2 are new objects that inherit the resources from x1 and y1, respectively, through the use of move constructors. After this operation, x1 and y1 may be left in a valid but unspecified state, ready for destruction or further assignment.

Examples

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

Activation of the move operator

Direct initialization

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

Argument passing to a function

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

```
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 { ... }
```

this is the move constructor for transferring ownership of data

Examples

```
class A {
    A();                          constructor
    A(const A&);                  copy constructor
    A(A&&) noexcept;              move constructor
    A& operator=(const A&);       copy assignment
    A& operator=(A&&) noexcept;   copy assignment with noexcept to indicate it never throws an exception
};

int main() {
    A a1;                         default constructor

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

Copy operations (copy constructor and copy assignment operator) create a new object as a copy of an existing object or assign the value of an existing object to another existing object. This means that the same data exists in two places, which can be inefficient, especially for large objects or when the object owns resources like dynamic memory or file handles.

Move operations (move constructor and move assignment operator), on the other hand, transfer resources from one object (the source) to another (the target). After the move, the source object is in a valid but unspecified state, and the target object has the same state as the source object had before the operation. This can be more efficient than copying, especially for large objects or when the object owns resources.

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; }
```

Constructor

Move constructor

Destructor

Move assignment operator

```
  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;
  }
};
```

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;        default constructors 2 times
  e2 = e1;         copy assignment operator
  C *e3 = new C;   default constructor
  e2 = *e3;        copy assignment operator
  return 0;
}                  destructor e2
```

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
}
```

Destructor Calls
Why two destructor calls?
e1 and e2 are automatic (stack-allocated) variables.
When the program exits main(), the destructors for e1 and e2 are called automatically.
e3 is a dynamic (heap-allocated) variable.
The destructor for e3 is not called automatically because e3 is allocated with new.
To properly manage the dynamic memory, you would need to delete e3 using delete e3; before returning from main().
Since delete e3; is not called, the destructor for the object pointed to by e3 is never called, leading to a memory leak.
Therefore, the destructors for the objects e1 and e2 are called exactly two times (once for each), but not for the object pointed to by e3 because it is dynamically allocated and not deleted.

Summary
The destructor is called two times because there are two automatic variables (e1 and e2) of class C that go out of scope at the end of main(). The dynamically allocated object (e3) is not deleted, hence its destructor is not called, which would result in a memory leak.

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;
}
```

default constructor

default constructor called once for e1, for *e2 nothing is called because it is just a pointer Instead, it simply declares a pointer to an object of type C without actually creating an instance of C.For the constructor to be called, you would need to create an instance of C and assign its address to the pointer p. For example, C *e2 = new C;

default constructor and copy constructor

Remember how C *e = new C means an instance of C is created and its address is assigned to the pointer.
*new C means instead of the address the value of the class is assigned. So default constructor is called once more.

10 default constructors are called

copy assignment operator

nothing

calls move assignment operator

since e2 is pointer and so is e4 nothing is called

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

C e1, *e2; - This line declares an object e1 of class C and a pointer e2 to an object of class C. The constructor for e1 is called here.

C e3 = *new C; - This line dynamically allocates memory for a new object of class C and then immediately dereferences it to initialize e3. This calls the constructor for the new object and then the copy constructor for e3.

e1 = std::move(e3); - This line assigns the value of e3 to e1 using the move assignment operator. After this line, e3 is in a valid but unspecified state.

e2 = std::move(e4); - This line assigns the value of e4 (which is a pointer to C) to e2 using move semantics. However, since e4 is a pointer, this doesn't actually call any special functions like the move assignment operator. It just copies the pointer value.

return 0; - At the end of the main function, all local objects are destroyed. This calls the destructor for e1 and e3. The dynamically allocated memory for the objects pointed to by e2 and e4 is not automatically deallocated, which would result in a memory leak. To avoid this, you should manually delete dynamically allocated memory when you're done with it.

e1 and e3
e2 and e4 are pointers