#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXPAROLA 30
#define MAXRIGA 80
int main(int argc, char *argv[])
{
int freq[MAXPAROLA] ; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
char riga[MAXRIGA] ;
int i, inizio, lunghezza ;
FILE * f ;
for(i=0; i<MAXPAROLA; i++)
freq[i]=0 ;
if(argc != 2)
{
fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
exit(1);
}
f = fopen(argv[1], "rt") ;
if(f==NULL)
{
fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
exit(1);
}
while(fgets(riga, MAXRIGA, f) != NULL)

High Level Programming

Copy Control

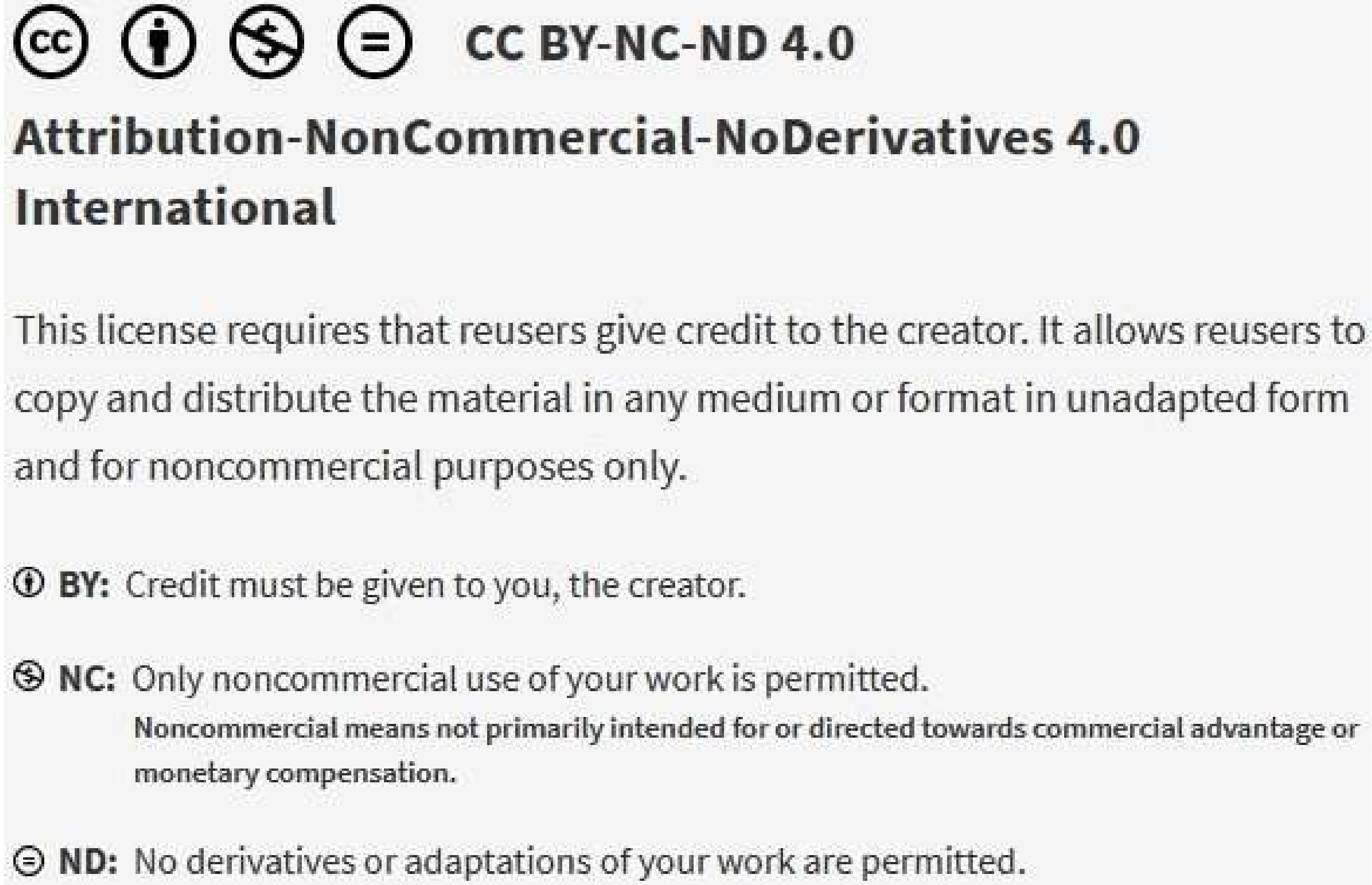
Stefano Quer

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 behaviour 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

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.

Constructor

#include <cstring>

Destructor

Destructors are denoted with ~

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

User-defined copy constructors

```
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

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.

#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.

We may want to duplicate the string

```
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.

User-defined copy constructor

Examples

Activation of the copy constructors

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

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;
```

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

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');

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

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;

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.

- 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;
}
```

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

Destructor

Introduced in Unit 03

- 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

Lvalue vs. Rvalue:
Lvalue: Represents something with a name or a memory location you can reference or modify directly. Examples include variables or objects you've created.
Rvalue: Represents a temporary value or an expression result. It's something you can't directly reference because it doesn't have a permanent memory location. Examples include literals (like numbers or strings) and temporary results of calculations.
Rvalue Reference (&&):
It's a special type of reference introduced in C++11 to handle temporary values or expressions.
You'll mainly see it when dealing with function arguments or return values that are temporary.
Move Semantic:
It's a way to efficiently transfer resources (like memory) from one object to another. Instead of copying large amounts of data, which can be slow and inefficient, move semantics allows you to "move" the data from one object to another, avoiding unnecessary duplication.

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

Activation of the move operator

```
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

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; }
```

Constructor

Move constructor

Destructor

```
  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;
  }
};
```

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