High Level Parallel Programming

Copy Control

Alessandro Savino and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- When a C++ class is defined the compiler **implicitly** (and automatically) defines how the class is
  - Copied, moved, assigned, destroyed
- A compiler controls these operations by defining 5 class members
  - Copy constructor, copy assignment, move constructor, move assignment, destructor

Destructors have been analyzed in the addendum section

- However, there are cases in which relying on the default definitions may lead to disaster
  - Thus, we need to learn how to define them

Copy Constructor

- A copy constructor is a special constructor that makes possible to define an object through a copy of an existing object of the same class
  - Given a class C, they are constructors that have a single argument of type C& or const C& (preferred)
  - There may be multiple copy constructors
- The copy constructors are often
  - Implicitly defined by the compiler
  - Implicitly called by the compiler whenever it is necessary to copy an object

Example

Wrong automatically created 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

#include <cstring>

Destructor

Example

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

Compiler-defined copy constructor

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?

User-defined copy constructor

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

Copy Constructor

- A copy constructor is a special constructor that makes possible defining an object as a copy of an existing object of the same class
- A copy constructor has only one formal parameter that is the type of the class (the parameter may be a reference to an object)

```
Rectangle(const Rectangle &to_copy);
```

Copy Constructor

- In the definition it is possible to refer to any private data of the object-to-copy directly
  - You must program what must be copied

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

Copy Constructor

- The invocation requires then to pass the object to be copied as parameter of the constructor

```
Rectangle r3(2,8)

Rectangle r4(r3);

Rectangle r5 = r3;

Rectangle r6;
...
r6 = r3;
```

This is still a copy constructor

This is an equivalent copy constructor

This is not a copy constructor (activated only when the object is created)

Move constructor

- Copy constructor and copy assignment follow a **copy semantics**
- Copy semantics often incur unnecessary or unwanted overhead
  - An object may
    - Be immediately destroyed after it is copied
    - Do not want to be used to share a resource that it is holding

Move constructor

- Move semantics provide a solution to such issues
  - Move constructors/assignment operators typically "steal" the resources of the argument
  - Leave the argument in a valid but indeterminate state
  - Greatly enhances performance in some cases

Move constructor

```
class_name ( class_name&& ) noexcept
```

- Typically called when an object is initialized from an rvalue reference of the same type
  - The class **class_name** must be the name of the current class
  - The **noexcept** keyword should be added to indicate that the constructor never throws an exception

Move constructor

- Explanation
  - Overload resolution decides if the copy or move constructor of an object should be called
  - The std::move function in the <utility> header may be used to convert a lvalue to a rvalue reference
  - We know that the argument does not need its resources anymore, so we can simply steal them

```
class_name ( class_name&& ) noexcept
```

Move constructor

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

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

Direct initialization

```
T a = std::move(b);
```

Copy initialization

```
f(std::move(a));        ←→        void f(T t);
```

Argument passing to a function

```
return a; inside T f();
```

Function return

Example

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

int main() {
  A a1;

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

Copy constructor definition

Move constructor definition

Calls copy constructor

Calls move constructor

Move assignment

```
class_name& operator=( class_name&& ) noexcept
```

- Typically called if an object appears on the left-hand side of an assignment with a rvalue reference on the right-hand side
  - The class **class_name** must be the name of the current class
  - The **noexcept** keyword should be added to indicate that the assignment operator never throws an exception

Move assignment

- Explanation
  - Overload resolution decides if the copy or move assignment operator of an object should be called
  - The **std::move** function in the **utility** header may be used to convert an lvalue to an rvalue reference
  - We know that the argument does not need its resources anymore, so we can simply steal them
  - The move assignment operator returns a reference to the object itself (i.e., ***this**) to allow for chaining

```
class_name& operator=( class_name&& ) noexcept
```

Example

```
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

Implementation guidelines

- A class that manages some kind of resource almost always requires custom move constructors and assignment operators
  - The programmer should either
    - Provide neither a move constructor nor a move assignment operator
    - Provide both
  - The move assignment operator should usually include a check to detect self-assignment

Implementation guidelines

- The move operations should typically not allocate new resources, but steal the resources from the argument
- The move operations should leave the argument in a valid state
- Any previously held resources must be cleaned up properly

Example

Objects

```
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; }
```

Example

```
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;
 }
};
```

Check for self-assignment

The "rule of three"

- If a class requires one of the following, it almost certainly requires all three
  1. A user-defined destructor
  2. A user-defined copy constructor
  3. A user-defined copy assignment operator
- Explanation
  - Having a user-defined copy constructor usually implies some custom setup logic which needs to be executed by copy assignment and vice-versa
  - Having a user-defined destructor usually implies some custom cleanup logic which needs to be executed by copy assignment and vice-versa
  - The implicitly-defined versions are usually incorrect if a class manages a resource of non-class type (e.g., a raw pointer, POSIX file descriptor, etc.)

The "rule of five"

- If a class follows the rule of three, move operations are defined as deleted
  - If move semantics are desired for a class, it must define all five special member functions
  - If only move semantics are desired for a class, it still must define all five special member functions, but define the copy operations as deleted
- Explanation
  - 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