High Level Parallel Programming

Dynamic Memory

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

Introduction

- C++ provides several mechanisms for dynamic memory management
  - New and delete expressions — Discouraged
  - C functions malloc and free — Discouraged
  - Smart pointers and ownership semantics — Preferred
- These mechanisms give control over the storage duration and object lifetime
  - Level of control varies by method
  - In all cases the manual intervention is required

Example

```
int foo(unsigned length) {
  int* buffer = new int[length];

  ... do something ...

  if (condition)
    return 42;

  ... do something else ...

  delete[] buffer;

  return 123;
}
```

C++ allocation

Memory leak

C++ deallocation

Lifetime and storage duration

- The lifetime of an object is equal to or nested within the lifetime of its storage
  - Equal for regular **new** and **delete**
  - Possibly nested for placement **new**

```
class A { };

int main() {
  A a1;

  A* a2 = new A();
  delete a2;
}
```

Lifetime of a1 begins
Storage begins

Lifetime of a2 begins

Lifetime of a2 ends

Lifetime of a1 ends.
Storage ends

Function std::memcpy

```
void* memcpy (
  void* dest, const void* src, std::size_t count
);
```

- Function std::memcpy copies bytes between non-overlapping memory regions
  - Defined in <cstring> standard header (thus, it exists in C too)
  - Copies **count** bytes from the object pointed to by **src** to the object pointed to by **dest**
  - Can be used to work around strict aliasing rules without causing undefined behavior

Function std::memcpy

- Restrictions (undefined behavior if violated)
  - Objects must not overlap
    - Not even partially
  - Pointers **src** and **dest** must not be **nullptr**
  - Objects must be trivially copyable (more details soon)
  - **dest** must be aligned suitably

```
void* memcpy (
  void* dest, const void* src, std::size_t count
);
```

Examples

Straightforward copy

```
#include <cstring>
#include <vector>

int main() {
  std::vector<int> buffer = {1, 2, 3, 4};
  buffer.resize(8);
  std::memcpy(&buffer[4],
    &buffer[0],
    4 * sizeof(int));
}
```

Copy first set of 4 elements into second set of 4 elements

Work around strict aliasing

```
#include <cstring>
#include <vector>

int main() {
  int64_t i = 42;
  double j;
  std::memcpy(&j, &i, sizeof(double));
}
```

Copy 64 bits
The value will probably not be 42
(double internal representation)

Function std::memmove

```
void* memmove(
  void* dest, const void* src, std::size_t count
);
```

- Function std::memmove copies bytes between possibly overlapping memory regions
  - Defined in <cstring> standard header (thus, it exists in C too)
  - Copies **count** bytes from the object pointed to by **src** to the object pointed to by **dest**
    - Acts as if the bytes were copied from the source buffer to a temporary buffer and then back to the destination

Function std::memmove

- Restrictions (undefined behavior if violated)
  - src and dest must not be nullptr
  - Objects must be trivially copyable (more details soon)
  - dest must be suitably aligned

```
void* memcpy (
  void* dest, const void* src, std::size_t count
);
```

Example

Straightforward copy

```
#include <cstring>
#include <vector>

int main() {
  std::vector<int> buffer = {1, 2, 3, 4};
  buffer.resize(6);

  std::memmove(&buffer[2],
    &buffer[0],
    4 * sizeof(int));
}
```

C++ standard library
We will analized it soon

Buffer now includes:
1 2 1 2 3 4

RAII

- In C/C++, it is possible to allocate different type of resources
  - Heap memory, sockets, files, mutexes, disk space, database connections, etc.
- To be safe we would like to **bind** the lifetime of the resource to the lifetime of the object representing the resource
  - A resource must be available during the entire lifetime of the object
  - All resources must be released when the lifetime of the object ends
  - Object should have automatic storage duration

RAII

- RAII stands for **Resource Acquisition is Initialization**
  - It is one of the most important and powerful idioms in C++
- Consequences
  - Never use **new** and **delete** outside a RAII class
  - C++ defines **smart pointers** that are RAII wrappers for **new** and **delete**
    - We (almost) never need to use **new** and **delete** in our code

RAII Implementation

- To adopt RAII, we must
  - Encapsulate each resource into a class whose sole responsibility is managing the resource
  - The constructor must acquire all resources and establish all class invariants
  - The destructor must release all resources
  - Typically, copy operations should be deleted

  and custom move operations need to be implemented
    - Thus we should not use copy
    - but use move instead

```
MyClass (const MyClass&) = delete;
MyClass& operator= (const MyClass&) = delete;
```

RAII Usage

- RAII classes should only be used with automatic or temporary storage duration
  - Ensures that the compiler manages the lifetime of the RAII object and thus indirectly manages the lifetime of the resource

Example

Buffer manipulation

```
class CustomIntBuffer {
  int* memory;

  public:
    CustomIntBuffer(unsigned size) :
      memory(new int[size]){}

    CustomIntBuffer(const CustomIntBuffer&) = delete;

    CustomIntBuffer(CustomIntBuffer&& other)
      noexcept : memory(other.memory) {
             other.memory = nullptr;
    }

  ~CustomIntBuffer() { delete[] memory; }
```

Constructor

Copy constructor
Defined and then deleted

Move constructor
Takes the object memory

Destructor

Example

```
  CustomIntBuffer& operator=
    (const CustomIntBuffer&) = delete;

  CustomIntBuffer& operator=
    (CustomIntBuffer&& other) noexcept {
      if (this != &other) {
        delete[] memory;
        memory = other.memory;
        other.memory = nullptr;
      }
    return *this;
  }

  int* getMemory() { return memory; }

  const int* getMemory() const { return memory; }
};
```

Copy assignment
Defined and then deleted

Move assignment
Takes the object memory

R/W

R only

Example

```
#include <utility>

bool foo(CustomIntBuffer buffer) {
  /* do something */
  if (condition)
    return false;   // no worries about
                    // forgetting to free memory
  /* do something more */
  return true;      // no worries about
                    // forgetting to free memory
}

int main() {
     CustomIntBuffer buffer(5);
     return foo(std::move(buffer));
}
```

Ownership

- One of the main challenges in manual memory management is tracking ownership
  - Traditionally, owners can be, e.g., functions or classes
  - Only the owner of some dynamically allocated memory may safely free it
  - Multiple objects may have a pointer to the same dynamically allocated memory
- The RAII idiom and move semantics together enable ownership semantics
  - A resource should be "owned", i.e. encapsulated, by exactly one C++ object always
  - Ownership can only be transferred explicitly by moving the respective object
    - E.g., the CustomIntBuffer class implements ownership semantics for a dynamically allocated int-array

C++ Smart Pointers

- Using C++, a set of special classes are dedicated to implement pointers
  - They resemble a pointer variable but being an object, they add several features that common pointers do not have
  - They all work exploiting the Template mechanism to adapt to all data

Unique Pointer

- std::unique_ptr is a so-called smart pointer (defined in <memory>)
  - Essentially implements RAII/ownership semantics for arbitrary pointers
  - Assumes unique ownership of another C++ object through a pointer
  - Automatically disposes of that object when the std::unique_ptr goes out of scope
- A std::unique_ptr may own no object, in which case it is empty
- Can be used (almost) exactly like a raw pointer
  - **std::unique_ptr** can only be moved, not copied

Unique Pointer

- It is a template class and can be used for arbitrary types
  - Syntax
    - std::unique_ptr< type >
    - Where one would otherwise use type*
- Creation
  - std::make_unique<type>(arg0, ..., argN)
    - Where arg0, ... are passed to the constructor of type
- Dereferencing, subscript, and member access
  - The dereference, subscript, and member access operators *, [] and -> can be used in the same way as for raw pointers

Unique Pointer

- Conversion to bool
  - **std::unique_ptr** is contextually convertible to bool, i.e. it can be used in if statements in the same way as raw pointers
- Accessing the raw pointer
  - The **get** member function returns the raw pointer
  - The **release** member function returns the raw pointer and releases ownership

Example

```
#include <memory>

class A {
  int a;
  int b;
public:
  A(int a, int b) : a(a), b(b) { }
  int getA() { return a; }
  int getB() { return b; }
};
```

Example

Assumes ownership

```
void foo(std::unique_ptr<A> aptr) {
  /* do something */
}

void bar(const A& a) {
  /* do something */
}

int main() {
    std::unique_ptr<A> aptr = std::make_unique<A>(42, 123);
    int a = aptr->getA();
    bar(*aptr);
    foo(std::move(aptr));
}
```

Does not assume ownership

Retains ownership

Transfers ownership

```
#include <memory>
class A {
  int a;
  int b;
public:
  A(int a, int b) : a(a), b(b) { }
  getA() { return a; }
  getB() { return b; }
};
```

Shared Pointer

- Rarely, true shared ownership is desired
  - A resource may be simultaneously having several owners
  - The resource should only be released once the last owner releases it
- **std::shared_ptr** defined in the **<memory>** standard header can be used for this
  - Multiple **std::shared_ptr** objects may own the same raw pointer (implemented through reference counting)
- It may be copied and moved

Shared Pointer

- Usage
  - Use **std::make_shared** for creation
  - Remaining operations analogous to **std::unique_ptr**
- **std**::**shared_ptr** is rather expensive and should be avoided when possible

Shared Pointer

```
#include <memory>
#include <vector>
using namespace std;

class Node {
   vector<shared_ptr<Node>> children;
public:
   void addChild(shared_ptr<Node> child);
   void removeChild(unsigned index);
   vector<shared_ptr<Node>> getChildren(){…};
};

int main() {
    Node root;
    root.addChild(make_shared<Node>());
    root.addChild(make_shared<Node>());
   root.getChildren()[0]->addChild(root.getChildren()[1]);


    root.removeChild(1);
    root.removeChild(0);
}
```

Does not free memory yet

Free memory of both children

Smart Pointer Usage Guidelines

- You still need to allocate memory, because those are still only pointers to memory location with some smart features
- Example

```
void my_func() {
   std::unique_ptr<int> valuePtr(new int(15));
   ...
   if (need to exit)
    return;
   ...
}
```

No memory leak anymore

Smart Pointer Usage Guidelines

- It is always possible to re-create the memory, if needed
- Example

```
std::unique_ptr<int> valuePtr;
...
valuePtr.reset(new int(47));
```

Smart Pointer Usage Guidelines

- **std::unique_ptr** represents ownership
  - Used for dynamically allocated objects
    - Frequently required for polymorphic objects
    - Useful to obtain a movable handle to an immovable object
  - used as a function parameter or return type indicates a transfer of ownership
  - it should almost always be passed by value

Smart Pointer Usage Guidelines

- Raw pointers represent resources
  - Should almost always be encapsulated in RAII classes (mostly **std::unique_ptr**)
  - Very occasionally, raw pointers are desired as function parameters or return types
    - If ownership is not transferred, but there might be no object (i.e., **nullptr**)
    - If ownership is not transferred, but pointer arithmetic is required

More To Explore

- std::weak_ptr
- std::auto_ptr
  - Replaced by unique_ptr since C++11
  - Read about its weaknesses to better understand the concepts