Computing at Scale

Lecture 7

Jacob Merson

February 3, 2024

Today's Agenda

- Virtual Functions
- RAII
- Smart Pointers
- Overload Resolution
- Start Templates

Virtual Functions

Noisy Class

Implement the destructor and copy/move assignment and copy/move constructors of the ”NoisyClass”. In the initial implementation, the constructor should take an integer that is stored as a member variable. For each of the ”rule of 5 methods” you should print a message saying which method are you in, and the number associated with that object. We will be using this substantially next class. Let me know if you have any questions.

Play with Noisy Class (when do copy/move happen)?

How does the compiler know which function to call? Overload resolution.

RAII

C++ requires explicit management of resources such as memory. This is why we have to match the calls to `new` with calls to `delete`. This is a source of many bugs. One of the most important idoms in C++ is **Resource Acquisition Is Initialization** (RAII).

RAII (cont.)

Key idea is to wrap resource construction and destruction in a class. Where the resource is acquired in the constructor and released in the destructor. This ensures that the resource is always released when the object goes out of scope. When RAII types are used, you can (almost) forget about managing memory.

RAII (example)

In Class Exercise

Write a RAII class that wraps allocation of a dynamic array of doubles. Write the necessary constructors/destructors/assignment operators. Replace the use of `new` and `delete` in our vector class with your RAII class.

Question: does your vector class still need:

- Copy constructor/assignment operator
- Move constructor/assignment operator
- Destructor?

RAII makes your code safer, easier to reason about and robust to exceptions. When using RAII types, you will only need to worry about the rule of 0! No need to write those pesky copy and move constructor, copy and move assignment operators and destructors.

Smart Pointers

C++ provides some basic RAII types to manage memory:

- `std::unique_ptr`
- `std::shared_ptr`

std::unique_ptr

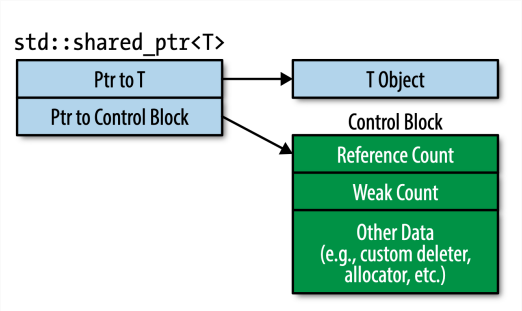
- “uniquely” owns dynamic object
- Create with `std::make_unique` or `std::unique_ptr<T> p(new T)`
- Object is deleted when it goes out of scope.
- Move only type

std::shared_ptr

- “shared” ownership of dynamic object
- Create with `std::make_shared` or `std::shared_ptr<T> p(new T)`
- Object is deleted when the last `shared_ptr` goes out of scope.
- Copyable

std::shared_ptr (cont.)

- Shared pointer uses reference counting
- `make_shared` uses single allocation for control block (reference count, deleter, etc.) and object.



Moving Forward

- Use `std::unique_ptr` by default
- Use `std::shared_ptr` when you need shared ownership. Use with caution as this can lead to tightly coupled code.
- No raw `new` or `delete`!
- Wrap those resources! You need to check this on code review.

Passing Smart Pointers

- Pass by value if you want to transfer ownership
- Pass raw pointer if function does not need to take ownership.
- Document behavior of function with respect to ownership. And, be consistent.

Overload Resolution

Overload resolution is the mechanism by which C++ picks the function to use when many functions have the same name. It is one of the most complicated parts of C++. We will not cover every detail here.

Overload Basics

- Name is looked up from an initial overload set
- Set is modified based on template argument deduction, etc. (SFINAE)
- Candidate that don't match call with implicit conversions and default arguments are removed from overload set.
- If there is one "best" candidate it is selected. If there is more than one, the call is ambiguous.
- Candidate is checked. If it is not valid, a diagnostic is issued. This can happen if it is deleted or inaccessible (private member function).

Note: Overload resolution is done after template deduction (more soon).

Rule of Thumb

Overload resolution will take the “most specific” function.

Example

```
void foo(int x) { std::cout << "int\n"; }
void foo(double x) { std::cout << "double\n"; }
void foo(float x) { std::cout << "float\n"; }
int main() {
  foo(1.0);
}
```

Example

```
void foo(int x) { std::cout << "int\n"; }
void foo(double x) { std::cout << "double\n"; }
void foo(float x) { std::cout << "float\n"; }
int main() {
  foo(1.0);
}
```

Ambiguous Example

```
void add(int, double);
void add(long, int);

int main() {
  add(1, 2); // ambiguous
}
```

Overload Resolution Match Order

1. Exact match. The parameter has the type of the expression, or a type that is a reference to the type of the expression.
2. Match with “minor adjustments”. For example, decay of an array to a pointer or addition of const (e.g. `int**`->`int const * const *`).
3. Match with promotions. For example implicit conversion of bool, char, short to int, unsigned int, long, or float to double.
4. Match with standard conversions. Conversion of derived class to base class, other built in conversions. No conversion operators or converting constructors.
5. Match with user defined conversions.
6. Match with ellipsis. Anything except class types with nontrivial copy constructor (implementation defined).