Unique Pointer Modern C++

By

SimplyProgram.Hub

Ashish Shelke

Raw Pointer – Potential Pitfalls

- Raw pointer is very powerful but has potential pitfalls because of developers' lack of awareness for memory management.

1. Raw pointer's declaration doesn't indicate whether it points to a single object or to an array.
2. Raw pointer's declaration doesn't reveal how to destroy objects it points to.
3. There is no way to provide how to destroy object pointed by raw pointer
4. Dynamic memory mismanagement causes, memory leaks due to inappropriate object deletion or sometimes segmentation fault due to accessing memory which is already deleted/freed.

Smart Pointer

- Smart pointer is wrapper around the raw pointer that acts like raw pointer but avoids memory leaks and segmentation faults.
- Smart pointers used to make sure that an object (dynamically allocated) is deleted. Such objects are destroyed in appropriate manner and at appropriate time.

- There are four smart pointers (#include <memory>)
  1. std::auto_ptr
  2. std::unique_ptr
  3. std::shared_ptr
  4. std::weak_ptr

Unique Pointer

- Unique pointer is used for exclusive ownership resource management.
- Non-null unique pointer always owns what it points to.
- Moving std::unique_ptr transfers ownership from source pointer to destination pointer, source pointer is set to null.

- Unique pointer is (It is move-only smart pointer)
  - MoveConstructible – Instance of the type can be constructed from r-value argument
  - MoveAssignable - instance of the type can be assigned from r-value argument

- Unique Pointer is not
  - CopyConstructible – Instance of the type can be copy constructed from l-value expression
  - CopyAssignable - instance of the type can be copy assigned from l-value expression

Deleter

- By default, object is deleted by ‘delete’ operator inside unique_ptr destructor

```
std::unique_ptr<T>ptr(nullptr);
```

- Also, unique_ptr deletes objects using user supplied function objects (functions/lambda expressions)

```
std::unique_ptr<T,decltype(deleterFunction)>ptr(nullptr, deleterFunction);
```

Versions

1. Manages single object (e.g., allocated with ‘new’)

```
std::unique_ptr<T> ptr(new T);
```

This version don’t provide indexing operator[].

2. Manages dynamically allocated array of objects (e.g., allocated with ‘new[]’)

```
std::unique_ptr<T[]> ptr(new T[3]);
```

This version don’t provide dereferencing operator* and operator->

std::make_unique<T>()

- It constructs an object of type T and wraps it in a std::unique_ptr

```
template< class T, class... Args >
unique_ptr<T> make_unique( Args&&... args );
```

- Return type is std::unique_ptr<T>

- Examples:
  1. std::unique_ptr<T> ptr = std::make_unique<T>();
  2. std::unique_ptr<T[]> ptr = std::make_unique<T[]>(size_t);

Relation with std::shared_ptr

- std::unique_ptr is easily and efficiently convertible to std::shared_ptr

- This feature so well suited for factory function return type.
- As factory function can’t know whether caller will want to use exclusive ownership for the object or shared ownership.

- Caller can convert this factory function returned unique pointer to shared pointer.
- Once ownership is made shared ownership, its not reversible. Though ownership count is one, it can’t claim ownership in order to mange it by unique pointer