CS 131 Compilers: Discussion 13: Garbage Collection

杨易为 季杨彪 尤存翰
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 5 月 26 日

1 C++ garbage collection

1.1 Resource Acquisition Is Initialization

RAII, which is the compiled time garbage collector first introduced in this thread, guarantees that the resource is available to any function that may access the object (resource availability is a class invariant, eliminating redundant runtime tests). It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition. Likewise, if resource acquisition fails (the constructor exits with an exception), all resources acquired by every fully-constructed member and base subobject are released in reverse order of initialization. This leverages the core language features (object lifetime, scope exit, order of initialization and stack unwinding) to eliminate resource leaks and guarantee exception safety. Another name for this technique is Scope-Bound Resource Management (SBRM), after the basic use case where the lifetime of an RAII object ends due to scope exit.

RAII can be summarized as follows:

1. encapsulate each resource into a class, where
   (a) the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
   (b) the destructor releases the resource and never throws exceptions;
2. always use the resource via an instance of a RAII-class that either
   (a) has automatic storage duration or temporary lifetime itself, or
   (b) has lifetime that is bounded by the lifetime of an automatic or temporary object

   Move semantics make it possible to safely transfer resource ownership between objects, across scopes, and in and out of threads, while maintaining resource safety.

   Classes with open()/close(), lock()/unlock(), or init()/copyFrom()/destroy() member functions are typical examples of non-RAII classes:

```
#include <bits\stdc++.h>

class ResourceGuard{
  private:
    const std::string resource;
  public:
    ResourceGuard(const std::string& res):resource(res){
      std::cout << "Acquire the " << resource << "." <<  std::endl;
    }
    ~ResourceGuard(){
      std::cout << "Release the "<< resource << "." << std::endl;
    }
};

int main() {
  ResourceGuard resGuard1{"memoryBlock1"};
```

```
        std::cout << "\nBefore local scope" << std::endl; {
          ResourceGuard resGuard2{"memoryBlock2"};
        }
        std::cout << "After local scope" << std::endl;
        std::cout << std::endl;
        std::cout << "\nBefore try-catch block" << std::endl;
        try {
            ResourceGuard resGuard3{"memoryBlock3"};
            throw std::bad_alloc();
        }
        catch (std::bad_alloc& e){
            std::cout << e.what();
        }
        std::cout << "\nAfter try-catch block" << std::endl;
        std::cout << std::endl;
      }
```

1.2 Modern C++ Resource Lifetime

1.2.1 string_view

The string ”move” util to make the char like variable, whatever it come from, the fastest to deal with the memory reallocation.

```
#include <string>
std::size_t length(const std::string &s){
  return s.size();
}
int main(){
  return length("hello world!");
}
```

1.2.2 PMR

1.3 Modern C++ Resource Lifetime