Memory Management & Smart Pointers

Proto Team 技术培训系列

上海合见工业软件集团有限公司
Shanghai UniVista Industrial Software Group Co.,Ltd.

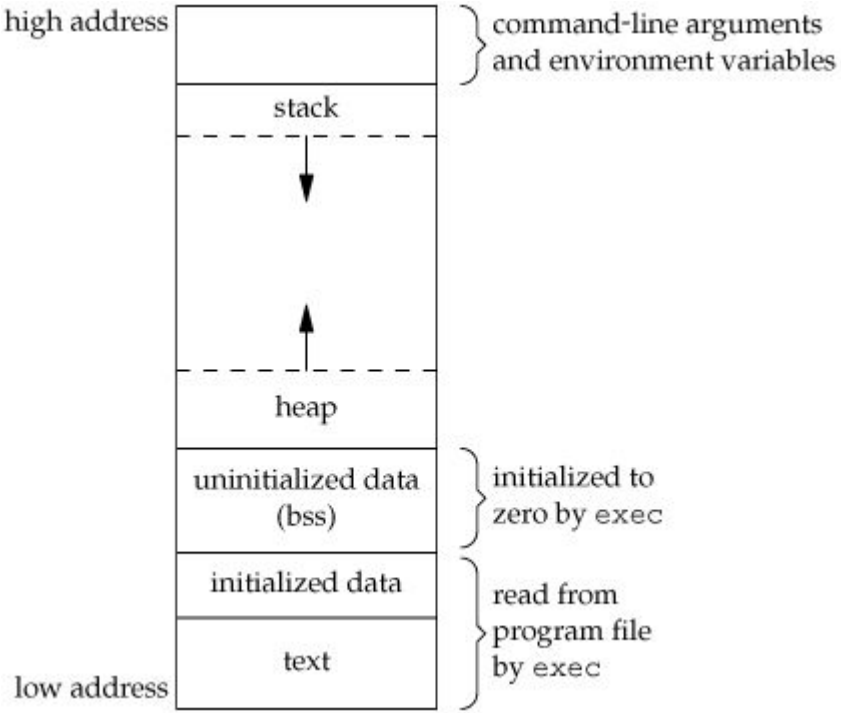
References

Back to Basics: C++ Smart Pointers - David Olsen - CppCon 2022

CppCon 2019: Arthur O'Dwyer “Back to Basics: Smart Pointers”

Memory Management

Process Memory




```cpp
#include <iostream>

// Global variables (initialized and uninitialized)
int global_var;
int global_initialized_var = 5;

// Function to demonstrate stack memory
void function() {
    int stack_var;
    std::cout << "The function's stack_var is at address: " << &stack_var << std::endl;
}

int main() {
    int stack_var; // Local variable on the stack
    static int static_initialized_var = 5; // Static variable in data segment
    static int static_var; // Uninitialized static variable in bss segment

    // Allocate memory on the heap
    int* heap_var_ptr = new int;
    std::cout << "heap_var is at address: " << heap_var_ptr << std::endl;

    // Print addresses of global and static variables
    std::cout << "global_initialized_var is at address: " << &global_initialized_var << std::endl;
    std::cout << "static_initialized_var is at address: " << &static_initialized_var << std::endl;
    std::cout << "static_var is at address: " << &static_var << std::endl;
    std::cout << "global_var is at address: " << &global_var << std::endl;

    // Print address of local variable on the stack
    std::cout << "stack_var is at address: " << &stack_var << std::endl;

    // Call the function to demonstrate stack memory
    function();

    // Clean up heap memory
    delete heap_var_ptr;

    return 0;
}
```

https://godbolt.org/z/P55PGav95

Stack

- 连续内存块
- 固定大小
  - 栈的大小通常在编译时固定
  - 栈的大小由操作系统或编译器设置确定
- 永远不会碎片化
- 分配速度快
- 每个线程独立

Stack Overflow

1. 定义大的局部对象，比如超大数组
2. 调用层数过深，比如无限递归

https://godbolt.org/z/dGj4s3rEd

```cpp
#include <iostream>

void func(std::byte* stack_bottom_addr)
{
    std::byte data[1024];
    std::cout << stack_bottom_addr - data << '\n';
    func(stack_bottom_addr);
}

int main()
{
    std::byte b;
    func(&b);

    return 0;       You, 2 seconds ago • Uncommitted ch
}
```

对象的内存占用

空类	1字节
成员变量	求和，加上数据对齐处理
静态成员变量	不占对象内存区
成员函数	不占对象内存区
静态函数	不占对象内存区
虚函数	增加8字节（64位系统）

内存对齐和填充

```
4   struct Foo {
5     int i;
6     double d;
7     char c;
8   };
```

sizeof(Foo) == ?

```
4   struct Foo {
5     double d;
6     int i;
7     char c;
8   };
```

扩展知识

- std::alignas – 可以指定对齐的大小
- std::alignof – 可以获得对齐的大小
- 可以

如果追求数据访问的效率

- 把大的数据放在前面
- 把常用的数据放在前面

https://godbolt.org/z/GEqh71acT

Heap Memory (or Free Store Memory)

属性	栈	堆
分配方式	由编译器自动分配	由程序员手动分配
分配范围	固定大小，由编译器指定	可变大小，由程序员指定
分配地址	从高地址向低地址增长	从低地址向高地址增长
分配效率	快	慢
内存碎片	不易产生	容易产生
使用场景	存放局部变量、函数参数	存放动态分配的对象、数组

- 栈的大小是有限的，局部变量过多或者调用层次过多，可能溢出。
- 堆的分配需要程序员手动释放，释放不当则可能内存泄漏。

内存碎片化

- 分配内存的效率降低

  当需要分配一个较大的内存块时，内存管理器需要将多个小的内存块合并在一起。这会导致内存管理器需要花费更多的时间和资源来分配内存。

- 回收内存的效率降低

  当释放一个内存块时，内存管理器需要将该内存块与其他小的内存块合并在一起。这会导致内存管理器需要花费更多的时间和资源来回收内存。

- 内存使用率降低

  由于内存被分割成许多小块，因此只有小部分内存可以被有效使用。

new & delete

- 对比 malloc 和 free
  - new 会调用类的构造函数，而 delete 会调用类的析构函数
- new 一个数组时，必须使用 delete[] 来释放
- 在使用智能指针后，要尽可能不使用new和delete
- 可以重载来自定义对内存分配和释放
  - 局部重载
  - 全局重载

扩展讨论

自定义内存管理方式

- Example : 4K Buffer

重载 new 和 delete

- Examples: Arena，调试和追踪

全局new和delete

小对象内存优化

- Example: String

Smart Pointers

裸指针（raw pointer）

- 强大，什么都能干
- 危险，使用不当会产生错误及危害
  1. 内存泄漏
  2. 悬挂指针和野指针
  3. 安全漏洞
  4. 难以调试和维护
- **不能**完整表达编程者的意图

Smart Pointer

- 行为像 Pointer
  - 指向一个对象
  - 可以被解引用（derefernce）：sp->something(), *sp
- 但是变得 Smart
  - 自动释放资源
  - 意图更加清晰
  - 多线程安全（shared_ptr）
  - 无需显式调用delelte，同时也尽量不要调用new

unique_ptr

拥有所指向内存

唯一拥有所指向的对象

会自动销毁所指向对象并释放内存

不可拷贝，只支持移动

unique_ptr – API

- 构造函数
  - `std::unique_ptr(T *ptr)`:将指针`ptr`指向的对象所有权转移给`unique_ptr`。
    i. 尽量使用`std::make_unique()`方法
  - `std::unique_ptr(std::nullptr_t)`:创建一个空的`unique_ptr`。
  - `std::unique_ptr(std::unique_ptr &&other)`:将`other`的所有权转移给`unique_ptr`。
- 析构函数
  - 在`unique_ptr`对象销毁时，会自动调用`delete`来释放其所指向的对象。
- 成员函数
  - `operator bool()`:返回`unique_ptr`是否指向一个有效的对象。
  - `reset()`:将`unique_ptr`指向一个新的对象，并释放其原来的所有权。
  - `release()`:返回`unique_ptr`所指向的对象，并将`unique_ptr`置空。
  - `operator*()`:返回`unique_ptr`所指向的对象的引用。
  - `operator->()`:返回`unique_ptr`所指向的对象的指针。

unique_ptr – 基础用法

```cpp
#include <iostream>
#include <memory>

class MyObject {
public:
    MyObject(int value): value_{value} { std::cout << "MyObject created" << std::endl; }
    ~MyObject() { std::cout << "MyObject destroyed" << std::endl; }

    int value() {return value_;}
private:
    int value_;
};

int main() {
  std::unique_ptr<MyObject> object(new MyObject(100));
  std::cout << object->value() << std::endl;
}
```

https://godbolt.org/z/TbfcT869W

unique_ptr – 转移Owner

```cpp
#include <iostream>
#include <memory>

class MyObject {
public:
    MyObject(int value): value_{value} { std::cout << "MyObject created" << std::endl; }
    ~MyObject() { std::cout << "MyObject destroyed" << std::endl; }

    int value() const {return value_;}
private:
    int value_;
};

void printObject(std::unique_ptr<MyObject> o)
{
    std::cout << o->value() << std::endl;
}

int main() {
    std::unique_ptr<MyObject> object = std::make_unique<MyObject>(100);
    printObject(std::move(object));
}
```

unique_ptr – 转移所有权

- 显式转移

  std::move

- 作为参数传递

  所有权从调用者转移给函数

  Note：reference不会转移控制权

- 函数返回一个unique_ptr

  所有权从函数转移到调用者

https://godbolt.org/z/xzK5qohrx

unique_ptr – 不转移所有权

如果只想使用对象，但是不想获得控制权

1. unique_ptr &
2. Raw pointer
3. Value reference (recommended)

https://godbolt.org/z/M8hKh7evs

unique_ptr – 容器

```cpp
class ObjectManager {
public:
    void addObject(int value)
    {
        objects_.emplace_back(std::make_unique<MyObject>(value));
    }
    void removeObject(int value)
    {
        auto predicate = [&value](const std::unique_ptr<MyObject>& object) {
            return object->value() == value;
        };

        objects_.erase(std::remove_if(objects_.begin(), objects_.end(), predicate), objects_.end());
    }

    MyObject* getObject(int value)
    {
        auto predicate = [value](const std::unique_ptr<MyObject>& object) {
            return object->value() == value;
        };

        auto found = std::find_if(objects_.begin(), objects_.end(), predicate);

        if (found != objects_.end()) {
            return found->get();
        }

        return nullptr;
    }

    int size() const {return objects_.size();}
    void printInfo() const {std::cout << "Object Manager with " << size() << " objects" << std::endl;}
private:
    std::vector<std::unique_ptr<MyObject>> objects_{};
};
```

https://godbolt.org/z/9T9vPs43b

shared_ptr

拥有所指向内存

共享拥有所指向的对象

会自动销毁所指向对象并释放内存

需要所有共享者共同参与

可以拷贝

unique_ptr

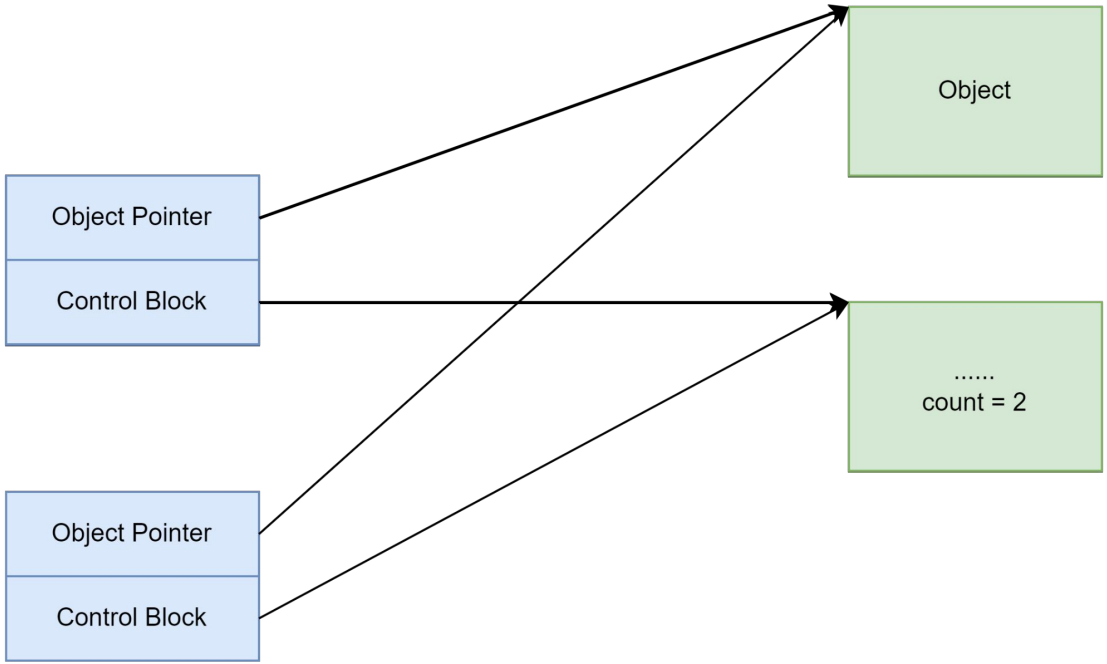
拥有所指向内存

唯一拥有所指向的对象

会自动销毁所指向对象并释放内存

不可拷贝，只支持移动

shared_ptr 图示



使用make_shared可以
优化内存分配

shared_ptr API

构造函数

- `shared_ptr(T* ptr)`:构造一个新的 shared_ptr，指向给定的指针。
- `shared_ptr(const shared_ptr& other)`:拷贝构造 – 增加引用计数
- `shared_ptr(shared_ptr&& other)`:移动构造 – 转移ownership
- `shared_ptr(unique_ptr<T>&& other)`： 从`unique_ptr`构造，转移`ownership`

析构函数

- `~shared_ptr()`:析构 shared_ptr。

方法

- `reset()`:重置 shared_ptr 指向的对象。
- `T* get()`:返回shared_ptr 的裸指针。
- `long use_count()`:返回shared_ptr 的引用计数。
- `operator*()`:返回 shared_ptr 指向的对象。
- `operator->()`:返回 shared_ptr 指向的对象的指针。

静态成员函数

- `make_shared()`:创建一个新的 shared_ptr。

没有release()方法

shared_ptr – 基础用法

```cpp
void printObject(std::shared_ptr<MyObject> o)
{
    std::cout << "In function use_count():" << o.use_count() << std::endl;
    o->print();
}


int main()
{
    auto a = std::make_shared<MyObject>(42);
    std::cout << "use_count() after created: " << a.use_count() << std::endl;

    printObject(a);
    std::cout << "use_count() after function: " << a.use_count() << std::endl;

    a.reset();

    std::cout << "The End." << std::endl;
}
```

https://godbolt.org/z/Tf64jM3af

Thread Safe

```cpp
int main()
{
    auto a = std::make_shared<MyObject>(42);

    std::thread t([](std::shared_ptr<MyObject> b){
        std::shared_ptr<MyObject> c = b;
        std::cout << "thread 1:" << c->value() << std::endl;
    }, a);

    {
        std::shared_ptr<MyObject> d = a;
        a.reset();
        std::cout << d->value() << std::endl;
    }

    t.join();
}
```

线程本身的逻辑正确性还是需要程序自己保证，shared_ptr只是保证共享指针自身在线程间的状态是一致的。

https://godbolt.org/z/ebno5j6ov

weak_ptr

- 必须和shared_ptr配套使用，其不具有 对象的ownership

  访问时计数器不会改变

- 不能直接dereference

  要先转换成一个shared_ptr访问元素

  从某种意义上来说，不是一个真正的pointer

- 应用场景有限

Weak_ptr 基本使用

```cpp
void printObject(std::weak_ptr<MyObject> o)
{
    std::cout << "In function use_count():" << o.use_count() << std::endl;
    if(auto o1 = o.lock())
    {
        o1->print();
    }
}

int main()
{
    auto a = std::make_shared<MyObject>(42);
    std::cout << "use_count() after created: " << a.use_count() << std::endl;
    //a.reset();
    printObject(a);

    a.reset();

    std::cout << "The End." << std::endl;

}
```

Weak_ptr
的推荐用法

https://godbolt.org/z/d4bxKfzP8

解决循环引用

```cpp
class A {
public:
  A() { std::cout << "A created" << std::endl; }
  ~A() { std::cout << "A destroyed" << std::endl; }

  std::shared_ptr<B> b;
};

class B {
public:
  B() { std::cout << "B created" << std::endl; }
  ~B() { std::cout << "B destroyed" << std::endl; }

  std::shared_ptr<A> a;
};

int main() {
  auto a = std::make_shared<A>();
  auto b = std::make_shared<B>();

  // 循环引用
  a->b = b;
  b->a = a;

  std::cout << "Game over." << std::endl;
}
```

https://godbolt.org/z/qE9GT4MeE

指南

- **使用智能指针表示所有权**：智能指针用于帮助确保程序不会出 现内存和资源泄漏，并且具有异常安全性。它 们在标准库的 `std` 命名空间中定义，是实现资源获取即初始化（RAII）编程习惯的关键。

- **优先使用 `unique_ptr` 而非 `shared_ptr`**：`unique_ptr` 是独占所有 权的智能指针，适用于管理 单一对象。相比之下 ，`shared_ptr` 允许多个指针共享同一 资源，但会增加复 杂性。

- **使用 `make_unique` 和 `make_shared`**：这些函数用于 创建智能指针，避免手 动管理内存。`make_unique` 用于创建 `unique_ptr`，`make_shared` 用于创建 `shared_ptr`。

- **尽量避免使用 `new` 和 `delete`**：在现代 C++ 中，应该少用 `new` 和 `delete`，而使用智能指 针来管理资源。

- **在函数间传递所有权时使用 `unique_ptr`**：通过传递或返回 `unique_ptr`，可以在函数之 间传递资源的所有 权。

扩展内容

- Custom deleters
- Casts

  dynamic_pointer_cast, static_pointer_cast, …
- shared_from_this & std::enable_shared_from_this

Smart pointer 自身占用内存情况

https://godbolt.org/z/YEssvsrsE

RAII

RAII是什么

- **R**esource **A**cquisition **I**s **I**nitialization

  资源获取即是初始化

  在对象的构造函数中 获取资源，而在析构函数中 释放资源，从而确保 资源的正确管理。

RAII的特点

- 资源获取和释放绑定到对象的生命周期

  在对象的构造函数中获取资源，而在析构函数中释放资源，确保资源的获取和释放与对象的生命周期绑定。这确保了在对象超出范围时资源被正确释放。

- 异常安全性

  如果在对象构造的过程中发生异常，析构函数会被自动调用，从而确保资源得以释放，避免资源泄漏。

- 简化资源管理

  程序员不需要显式地调用资源的分配和释放函数，因为这些操作都被封装在对象的构造和析构函数中。

- 可组合性

  允许资源管理的嵌套和组合，对象可以包含其他对象作为其成员，从而形成一个资源管理的层次结构。

- 代码清晰度高

Resources

资源	创建方式	销毁方式
内存	new	delete
文件	fopen()	fclose()
网络连接	socket()	close()
线程	thread()	join()
互斥锁	mutex()	unlock()
条件变量	condition_variable()	notify_one()
信号量	semaphore()	release()

Examples

- Smart pointers

  当对象离开作用域后，会自 动销毁

- ofstream

  在创建ofstream对象时打开文件，当ofstream对象离开作用域后，会自 动关闭

- QSignalBlocker

  在创建时关闭对象的信号，当离开作用域 时恢复 对象的信号处理

Example: C FILE wrapper

https://godbolt.org/z/o9G8b68K1

```
1. class FileHolder {
2. public:
3.     FileHolder(my::FILE* f) : m_f(f) {}
4.     ~FileHolder() { my::fclose(m_f); }
5.     operator my::FILE*() { return m_f; }
6.
7.     static auto make_holder(const char* name) {
8.         return FileHolder{my::fopen(name)};
9.     }
10. private:
11.     my::FILE* m_f;
12. };
13.
14. int main() {
15.     FileHolder h = FileHolder::make_holder("def");
16.
17.     std::cout << "The end" << std::endl;
18.
19.     return 0;
20. }
```

如果在16行插入
auto h1 = h;
会如何？

用unique_ptr方式实现FileHolder

直接使用unique_ptr

https://godbolt.org/z/6aG8j6ME6

包装unique_ptr

https://godbolt.org/z/jenvP8f71

扩展内容 – unique_resource

https://godbolt.org/z/8cc5cfWxs

Scope Bound Operation Pairs

- RAII的一种应用方式
- 更强调作用域，在脱离作用域后的动作比较重要，并不仅仅是清理工作
- 是代码逻辑的一部分

```cpp
#include <iostream>

// 自定义资源管理类
class ResourceGuard {
public:
    ResourceGuard() {
        std::cout << "Resource acquired." << std::endl;
    }

    ~ResourceGuard() {
        std::cout << "Resource released." << std::endl;
    }
};

int main() {
    {
        // 在作用域内创建 ResourceGuard 对象，构造函数中获取资源，析构函数中释放
        ResourceGuard guard;

        // 在这个作用域内，资源处于被管理状态
        // ...

    } // 离开作用域，ResourceGuard 对象被销毁，析构函数中释放资源

    // 在这个作用域外，资源已经被释放

    return 0;
}
```

Example: QSignalBlocker 实现示意

```cpp
class QSignalBlocker
{
public:
    explicit QSignalBlocker(QObject *object)
        : m_object(object)
    {
        if (m_object)
            m_object->blockSignals(true);
    }

    ~QSignalBlocker()
    {
        if (m_object)
            m_object->blockSignals(false);
    }

private:
    QObject *m_object;
};
```

```cpp
void MyClass::doSomething()
{
    // 创建一个QSignalBlocker对象，阻止myObject的信号发射
    QSignalBlocker blocker(myObject);

    // 在这个代码块中，myObject的信号将被阻止
    // 执行一些操作，不会触发myObject的信号处理函数

    // 代码块结束后，QSignalBlocker对象将被销毁
    // myObject的信号将恢复正常发射
}
```

Qt自身的实现方式更复杂一些，考虑了拷贝、移动、以及原始对象的block状态等。
https://codebrowser.dev/qt5/qtbase/src/corelib/kernel/qobject.h.html#557

扩展内容:ScopedTimer

```cpp
class ScopedTimer {
public:
    using ClockType = std::chrono::steady_clock;
    ScopedTimer(std::string_view func)
        : function_name_{func}, start_{ClockType::now()} {
            std::cout << "===Entering " << func << std::endl;
        }

    ScopedTimer(const ScopedTimer&) = delete;
    ScopedTimer(ScopedTimer&&) = delete;

    auto operator=(const ScopedTimer&) -> ScopedTimer& = delete;
    auto operator=(ScopedTimer&&) -> ScopedTimer& = delete;

    ~ScopedTimer() {
        using namespace std::chrono;
        auto stop = ClockType::now();
        auto duration = (stop - start_);
        auto ms = duration_cast<milliseconds>(duration).count();
        std::cout << ms << " ms " << function_name_ << '\n';
    }
private:
    std::string function_name_{};
    const ClockType::time_point start_{};
};
```

https://godbolt.org/z/Ev9Y34j7r

小结

1. 堆和栈
    a. 优先使用栈
    b. 对象在内存中的内存使用
    c. Align 和 padding

2. 智能指针是RAII在内存管理的应用
    a. unique_ptr和shared_ptr 都蕴含着对象的所有权
    b. unique_ptr几乎不带来性能的开销
    c. weak_ptr需要和shared_ptr配套使用

3. RAII是一种简单且强大的资源管理方式

THANKS

上海合见工业软件集团有限公司

Shanghai UniVista Industrial Software Group Co.,Ltd.