

Memory Management & Smart Pointers

Proto Team 技术培训系列

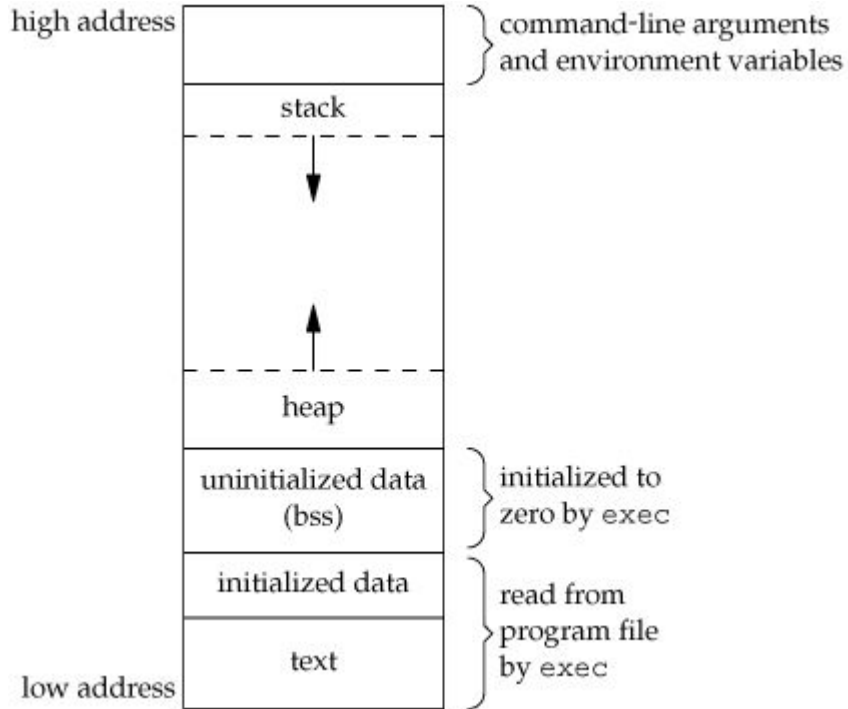
References

[Back to Basics: C++ Smart Pointers - David Olsen - CppCon 2022](#)

[CppCon 2019: Arthur O'Dwyer “Back to Basics: Smart Pointers”](#)

Memory Management

Process Memory



```
#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;
}
```

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>

```
#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 (dereference) : `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 `delete`, 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

```
1 #include <iostream>
2 #include <memory>
3
4 class MyObject {
5 public:
6     MyObject(int value): value_{value} { std::cout << "MyObject created" << std::endl; }
7     ~MyObject() { std::cout << "MyObject destroyed" << std::endl; }
8
9     int value() {return value_;}
10 private:
11     int value_;
12 };
13
14 int main() {
15     std::unique_ptr<MyObject> object(new MyObject(100));
16     std::cout << object->value() << std::endl;
17 }
```

unique_ptr – 转移Owner

```
1 #include <iostream>
2 #include <memory>
3
4 class MyObject {
5 public:
6     MyObject(int value): value_{value} { std::cout << "MyObject created" << std::endl; }
7     ~MyObject() { std::cout << "MyObject destroyed" << std::endl; }
8
9     int value() const {return value_;}
10 private:
11     int value_;
12 };
13
14 void printObject(std::unique_ptr<MyObject> o)
15 {
16     std::cout << o->value() << std::endl;
17 }
18
19 int main() {
20     std::unique_ptr<MyObject> object = std::make_unique<MyObject>(100);
21     printObject(std::move(object));
22 }
```

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

```
18 class ObjectManager {  
19 public:  
20     void addObject(int value)  
21     {  
22         objects_.emplace_back(std::make_unique<MyObject>(value));  
23     }  
24     void removeObject(int value)  
25     {  
26         auto predicate = [&value](const std::unique_ptr<MyObject>& object) {  
27             return object->value() == value;  
28         };  
29         objects_.erase(std::remove_if(objects_.begin(), objects_.end(), predicate), objects_.end());  
30     }  
31  
32     MyObject* getObject(int value)  
33     {  
34         auto predicate = [&value](const std::unique_ptr<MyObject>& object) {  
35             return object->value() == value;  
36         };  
37  
38         auto found = std::find_if(objects_.begin(), objects_.end(), predicate);  
39  
40         if (found != objects_.end()) {  
41             return found->get();  
42         }  
43  
44         return nullptr;  
45     }  
46  
47     int size() const {return objects_.size();}  
48     void printInfo() const {std::cout << "Object Manager with " << size() << " objects" << std::endl;}  
49 private:  
50     std::vector<std::unique_ptr<MyObject>> objects_{};  
51 };  
52
```

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

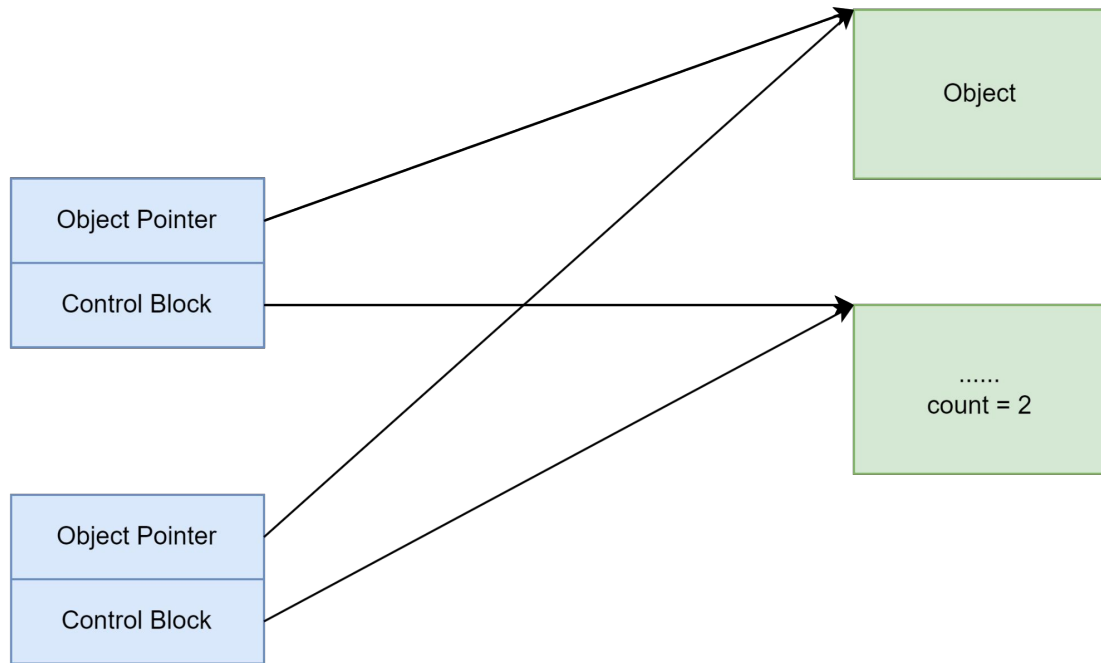
拥有所指向内存

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



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`。

shared_ptr – 基础用法

<https://godbolt.org/z/Tf64jM3af>

Thread Safe

<https://godbolt.org/z/ebno5j6ov>

```
17 int main()
18 {
19     auto a = std::make_shared<MyObject>(42);
20
21     std::thread t([](std::shared_ptr<MyObject> b){
22         std::shared_ptr<MyObject> c = b;
23         std::cout << "thread 1:" << c->value() << std::endl;
24     }, a);
25
26     {
27         std::shared_ptr<MyObject> d = a;
28         a.reset((MyObject*)nullptr);
29         std::cout << d->value() << std::endl;
30     }
31
32     t.join();
33 }
```

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

<https://godbolt.org/z/eevoaMxxE>

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