

84 Track MEMORY ALLOCATIONS the Easy way in C++

内存是很重要的，计算机和内存是紧密相联的，如果你只有一个CPU而没有RAM没有内存，那么你什么都做不了。现在内存问题不像20年前那么严重了，我们的电脑有了8、16、32G的内存。

而写一个C++程序是比较难的，它会用到大量内存，所以了解你的程序什么时候分配内存，特别是堆内存是很有用的。如果你知道程序在哪里分配内存，你就有可能减少它，从而能优化你的程序并使其运行得更快，因为在堆上分配内存并不是最好的做法，尤其是在性能关键的代码中，除此之外，能够看到内存被分配到哪里，还可以帮助你更多地了解程序是如何工作的，即使所有正在运行的代码都是你写的，本节将讨论如何准确地看到每个分配从何而来，以及如何跟踪应用程序内内存的整体使用情况，只需编写代码而不依赖任何其它工具，所展示的东西都可以很容易地插入到你现有的应用程序中。

1. 重载new操作符

```
C++

void* operator new(size_t size)
{
    std::cout << "Allocating " << size << " bytes\n";
    return malloc(size);
} // breakpoint

struct Object
{
    int x, y, z;
};

int main()
{
    Object* obj = new Object;
    std::unique_ptr<Object> obj2 = std::make_unique<Object>();
}
```

```
template <class _Ty, class... _Types, enable_if_t<!is_array_v<_Ty>, int> = 0>
_NODISCARD _CONSTEXPR23 unique_ptr<_Ty> make_unique(_Types&&... _Args) { // ma
    return unique_ptr<_Ty>(new _Ty(_STD forward<_Types>(_Args)...)); ≤ 9ms elapsed
}
```

查看发现直接堆分配实例化和unique_ptr都会通过调用new在堆上分配对象。

可以看到，这样在new函数中插入一个断点来精确追踪内存分配的来源有多有用。

2. 重载delete操作符

```

#include <iostream>
#include <memory>
#include <string>

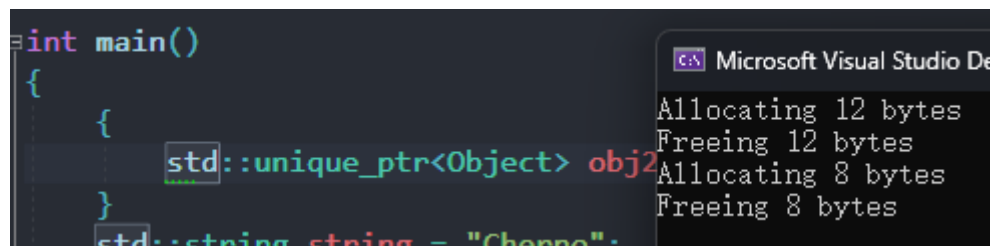
void* operator new(size_t size)
{
    std::cout << "Allocating " << size << " bytes\n";
    return malloc(size);
}

void operator delete(void* memory, size_t size)
{
    std::cout << "Freeing " << size << " bytes\n";
    free(memory);
}

struct Object
{
    int x, y, z;
};

int main()
{
    {
        std::unique_ptr<Object> obj2 = std::make_unique<Object>();
    }
    std::string string = "Cherno";
}

```



```

int main()
{
    {
        std::unique_ptr<Object> obj2 = std::make_unique<Object>();
    }
    std::string string = "Cherno";
}

```

Microsoft Visual Studio De

Allocating 12 bytes
Freeing 12 bytes
Allocating 8 bytes
Freeing 8 bytes

现在使用这两个函数，我们可以创建内存分配跟踪器，我们可以维护一个 *allocation metrics*，可以确切地知道有多少内存被使用，有多少内存被分配，有多少内存被释放等，这是非常有用的。

```
struct AllocationMetrics
{
    uint32_t TotalAllocated = 0;
    uint32_t TotalFreed = 0;

    uint32_t CurrentUsage() { return TotalAllocated - TotalFreed; }
};

static AllocationMetrics s_allocation_metrics;

void* operator new(size_t size)
{
    s_allocation_metrics.TotalAllocated += size;
    return malloc(size);
}

void operator delete(void* memory, size_t size)
{
    s_allocation_metrics.TotalFreed += size;
    free(memory);
}

struct Object
{
    int x, y, z;
};

static void PrintMemoryUsage()
{
    std::cout << "Memory Usage: " << s_allocation_metrics.CurrentUsage() << " bytes\n";
}

int main()
{
    PrintMemoryUsage();
    std::string string = "Cherno";
    PrintMemoryUsage();
    {
        std::unique_ptr<Object> obj2 = std::make_unique<Object>();
        PrintMemoryUsage();
    }
    PrintMemoryUsage();
}
```

```
int main()
{
    PrintMemoryUsage();
    std::string string = "Cherno";
    PrintMemoryUsage();
    {
        std::unique_ptr<Object> obj2 = std::make_unique<Object>();
        PrintMemoryUsage();
    }
    PrintMemoryUsage();
}
```

Microsoft Visual Studio Debug

Memory Usage: 0 bytes
Memory Usage: 8 bytes
Memory Usage: 20 bytes
Memory Usage: 8 bytes

C:\Dev\Small String\Debug
To automatically close the console window when debugging stops, please select an option from the Debug menu or press Alt+F5.
Press any key to close this window.

当然你也可以使用VS内置的内存分配跟踪分析工具，还有Valgrind等一些很好的现成的工具，但本节只是一个快速简单的方法，可以看到你的代码中发生了什么并跟踪它