智能指针

C++11 引入了 3 个智能指针类型：

1. `std::unique_ptr<T>` ：独占资源所有权的指针。
2. `std::shared_ptr<T>` ：共享资源所有权的指针。
3. `std::weak_ptr<T>` ：共享资源的观察者，需要和 std::shared_ptr 一起使用，不影响资源的生命周期。

std::auto_ptr 已被废弃。

std::unique_ptr

简单说，当我们独占资源的所有权的时候，可以使用 std::unique_ptr 对资源进行管理——离开 unique_ptr 对象的作用域时，会自动释放资源。这是很基本的 RAII 思想。

std::unique_ptr 的使用比较简单，也是用得比较多的智能指针。这里直接看例子。

1. 使用裸指针时，要记得释放内存。

```
{
    int* p = new int(100);
    // ...
    delete p;  // 要记得释放内存
}
```

1. 使用 std::unique_ptr 自动管理内存。

```
{
    std::unique_ptr<int> uptr = std::make_unique<int>(200);
    //...
    // 离开 uptr 的作用域的时候自动释放内存
}
```

1. std::unique_ptr 是 move-only 的。

```
std::unique_ptr<int> uptr = std::make_unique<int>(200);
std::unique_ptr<int> uptr1 = uptr;  // 编译错误，std::unique_ptr<T> 是 move-only 的

std::unique_ptr<int> uptr2 = std::move(uptr);
assert(uptr == nullptr);
```

1. std::unique_ptr 可以指向一个数组。

```
{
    std::unique_ptr<int[]> uptr = std::make_unique<int[]>(10);
    for (int i = 0; i < 10; i++) {
        uptr[i] = i * i;
    }
    for (int i = 0; i < 10; i++) {
        std::cout << uptr[i] << std::endl;
    }
}
```

1. 自定义 deleter。

```
{
    struct FileCloser {
        void operator()(FILE* fp) const {
            if (fp != nullptr) {
                fclose(fp);
            }
        }
    };
    std::unique_ptr<FILE, FileCloser> uptr(fopen("test_file.txt", "w"));
}
```

std::shared_ptr

std::shared_ptr 其实就是对资源做引用计数——当引用计数为 0 的时候，自动释放资源。

```
{
    std::shared_ptr<int> sptr = std::make_shared<int>(200);
    assert(sptr.use_count() == 1);  // 此时引用计数为 1
    {
        std::shared_ptr<int> sptr1 = sptr;
        assert(sptr.get() == sptr1.get());
        assert(sptr.use_count() == 2);   // sptr 和 sptr1 共享资源，引用计数为 2
    }
    assert(sptr.use_count() == 1);   // sptr1 已经释放
}
// use_count 为 0 时自动释放内存
```

和 unique_ptr 一样，shared_ptr 也可以指向数组和自定义 deleter。

```
{
    // C++20 才支持 std::make_shared<int[]>
    // std::shared_ptr<int[]> sptr = std::make_shared<int[]>(100);
    std::shared_ptr<int[]> sptr(new int[10]);
    for (int i = 0; i < 10; i++) {
        sptr[i] = i * i;
    }
    for (int i = 0; i < 10; i++) {
        std::cout << sptr[i] << std::endl;
    }
}

{
    std::shared_ptr<FILE> sptr(
        fopen("test_file.txt", "w"), [](FILE* fp) {
            std::cout << "close " << fp << std::endl;
            fclose(fp);
        });
}
```

std::weak_ptr

std::weak_ptr 要与 std::shared_ptr 一起使用。 一个 std::weak_ptr 对象看做是 std::shared_ptr 对象管理的资源的观察者，它不影响共享资源的生命周期：

1. 如果需要使用 weak_ptr 正在观察的资源，可以将 weak_ptr 提升为 shared_ptr。
2. 当 shared_ptr 管理的资源被释放时，weak_ptr 会自动变成 nullptr。\

```
void Observe(std::weak_ptr<int> wptr) {
    if (auto sptr = wptr.lock()) {
        std::cout << "value: " << *sptr << std::endl;
    } else {
        std::cout << "wptr lock fail" << std::endl;
    }
}

std::weak_ptr<int> wptr;
{
    auto sptr = std::make_shared<int>(111);
    wptr = sptr;
    Observe(wptr);  // sptr 指向的资源没被释放，wptr 可以成功提升为 shared_ptr
}
Observe(wptr);  // sptr 指向的资源已被释放，wptr 无法提升为 shared_ptr
```

关键字

inline

> **1. 引入inline关键字的原因**

在c/c++中，**为了解决**一些频繁调用的小函数大量消耗栈空间（栈内存）的问题，特别的引入了inline修饰符，表示为内联函数。

栈空间就是指放置程序的局部数据（也就是函数内数据）的内存空间。

在系统下，栈空间是有限的，假如频繁大量的使用就会造成因栈空间不足而导致程序出错的问题，如，函数的死循环递归调用的最终结果就是导致栈内存空间枯竭。

> **2. inline使用限制**

inline的使用是**有所限制的**，inline只适合涵数体内代码简单的涵数使用，不能包含复杂的结构控制语句例如while、switch，并且不能内联函数本身不能是直接递归函数（即，自己内部还调用自己的函数）。

> **3. inline仅是一个对编译器的建议**

inline函数仅仅是一个**对编译器的建议**，所以**最后能否真正内联，看编译器的意思**，它如果认为函数不复杂，能在调用点展开，就会真正内联，并不是说声明了内联就会内联，声明内联只是一个建议而已。

6. inline 是一种“用于实现的关键字”

关键字inline 必须与**函数定义体**放在一起才能使函数成为内联，仅将inline 放在函数声明前面**不起任何作用**。

如下风格的函数Foo 不能成为内联函数：

```objectivec
inline void Foo(int x, int y); // inline 仅与函数声明放在一起

void Foo(int x, int y){}
```


而如下风格的函数Foo 则成为内联函数：

```objectivec
void Foo(int x, int y);

inline void Foo(int x, int y) {} // inline 与函数定义体放在一起
```

所以说，inline 是一种“**用于实现的关键字**”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了inline 关键字，但我认为**inline不应该出现在函数的声明中**。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：**声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。**

7. 慎用inline

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联是以**代码膨胀（复制）**为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。

如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

以下情况不宜使用内联：

（1）如果函数体内的代码**比较长**，使用内联将导致**内存消耗代价较高。**

（2）如果函数体内出现**循环**，那么执行函数体内代码的时间要比函数调用的开销大。类的构造函数和析构函数容易让人误解成使用内联更有效。要当心**构造函数和析构函数可能会隐藏一些行为**，如“偷偷地”执行了**基类或成员对象**的构造函数和析构函数。所以**不要随便地将构造函数和析构函数的定义体放在类声明中。**一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中）。

8.总结

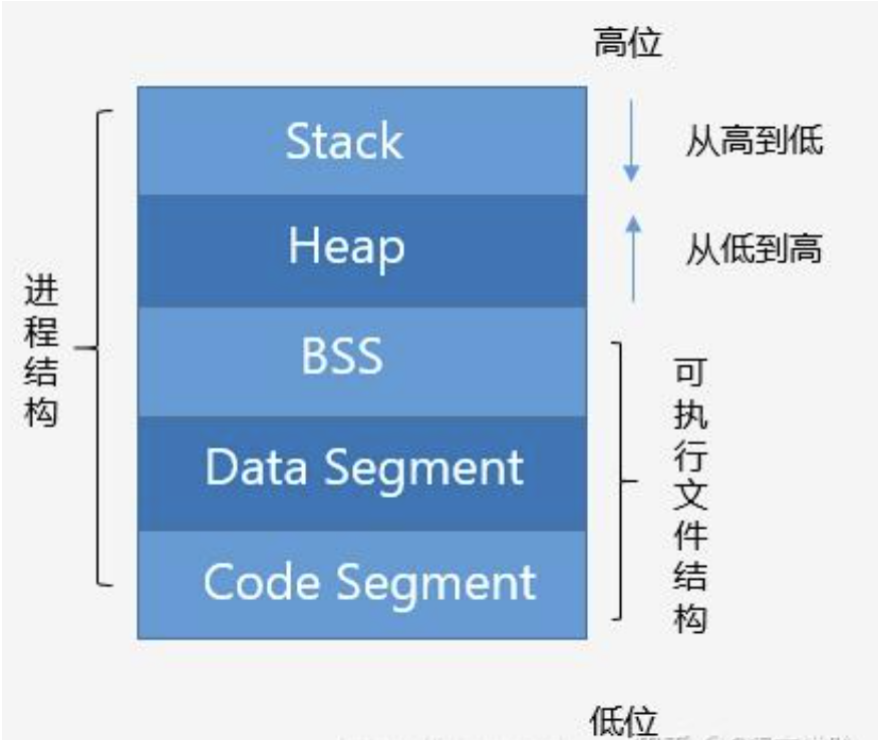
内联函数并不是一个增强性能的灵丹妙药。只有当**函数非常短小**的时候它才能得到我们想要的效果；但是，如果函数并不是很短而且在很多地方都被调用的话，那么将会使得可执行体的体积增大。

static

mutable

内存管理

内存分区



Code Segment⁺（代码区）

也称Text Segment，存放可执行程序的机器码。

Data Segment⁺（数据区）

存放已初始化的全局和静态变量， 常量数据（如字符串常量）。

BSS⁺（Block started by symbol）

存放未初始化的全局和静态变量。（默认设为0）

Heap⁺（堆）

从低地址向高地址增长。容量大于栈，程序中动态分配的内存在此区域。

Stack⁺（栈）

从高地址向低地址增长。由编译器自动管理分配。程序中的局部变量、函数参数值、返回变量等存在此区域。

malloc

内存对齐

为什么要内存对齐

1. **平台原因(移植原因)**：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
2. **性能原因**：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

- 假如没有内存对齐机制，数据可以任意存放，现在一个int变量存放在从地址1开始的联系四个字节地址中，该处理器去取数据时，要先从0地址开始读取第一个4字节块,剔除不想要的字节（0地址）,然后从地址4开始读取下一个4字节块,同样剔除不要的数据（5，6，7地址）,最后留下的两块数据合并放入寄存器。这需要做很多工作。
- 现在有了内存对齐的，int类型数据只能存放在按照对齐规则的内存中，比如说0地址开始的内存。那么现在该处理器在取数据时一次性就能将数据读出来了，而且不需要做额外的操作，提高了效率。

3、内存对齐规则

每个特定平台上的编译器✦都有自己的默认“对齐系数”（也叫对齐模数）。gcc中默认#pragma pack(4)，可以通过预编译命令#pragma pack(n)，n = 1,2,4,8,16来改变这一系数。

有效对其值：是给定值#pragma pack(n)和结构体中最长数据类型长度中较小的那个。有效对齐值也叫**对齐单位**。

了解了上面的概念后，我们现在可以来看看内存对齐需要遵循的规则：

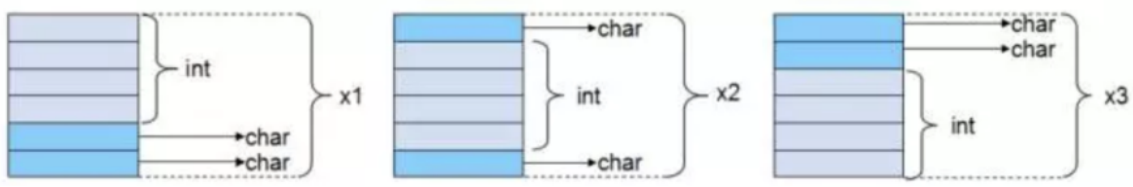
(1) 结构体第一个成员的**偏移量（offset）**为0，以后每个成员相对于结构体首地址的 offset 都是**该成员大小与有效对齐值中较小那个**的整数倍，如有需要编译器会在成员之间加上填充字节。

(3) **结构体的总大小**为 有效对齐值 的**整数倍**，如有需要编译器会在最末一个成员之后加上填充字节。

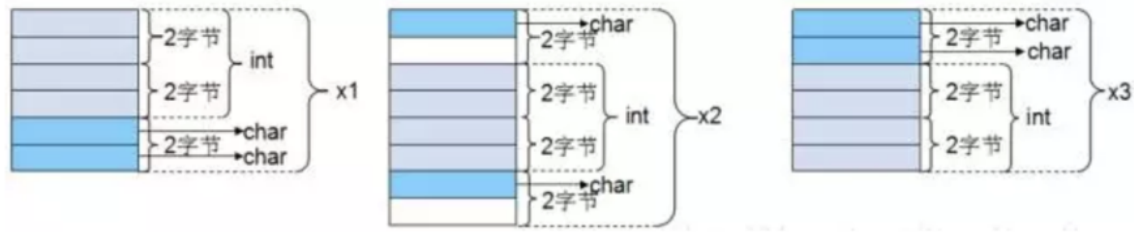
#pragma pack(n)

不同平台上编译器的 pragma pack 默认值不同。而我们可以通过预编译命令#pragma pack(n), n= 1,2,4,8,16来改变对齐系数。

例如，对于上个例子的三个结构体，如果前面加上#pragma pack(1)，那么此时有效对齐值为1字节，此时根据对齐规则，不难看出成员是连续存放的，三个结构体的大小都是6字节。



如果前面加上#pragma pack(2)，有效对齐值为2字节，此时根据对齐规则，三个结构体的大小应为6,8,6。内存分布图如下：



STL