C++ 整理

1. C++ 编译过程

1.1 预编译

- 宏函数：在预编译时把所有宏名用宏体来替换
- define

1.2 编译

- 内联函数：在编译时进行代码插入，省区函数调用开销，提高效率
- const
- 命名倾轧（name mangling）：同名函数重载，在编译阶段更改函数名来区分参数不同的同名函数
- 虚函数表
- 数组指针
- 静态存储区的内容
- const_cast, reinterpret_cast, static_cast

1.3 汇编

1.4 链接

2. 内存管理

2.1 全局/静态存储区

- 静态全局变量，静态局部变量，全局变量：程序结束回收内存；
- 类静态成员变量：当超出类作用域时回收内存；

2.2 常量存储区

存放const常量，不允许修改

2.3 自由存储区

由malloc分配的内存块

2.4 栈

由操作系统自动分配释放，存放函数的参数值，存放在一级缓存

- 局部变量：出了作用域回收内存

2.5 堆

由程序员分配释放，存放在二级缓存

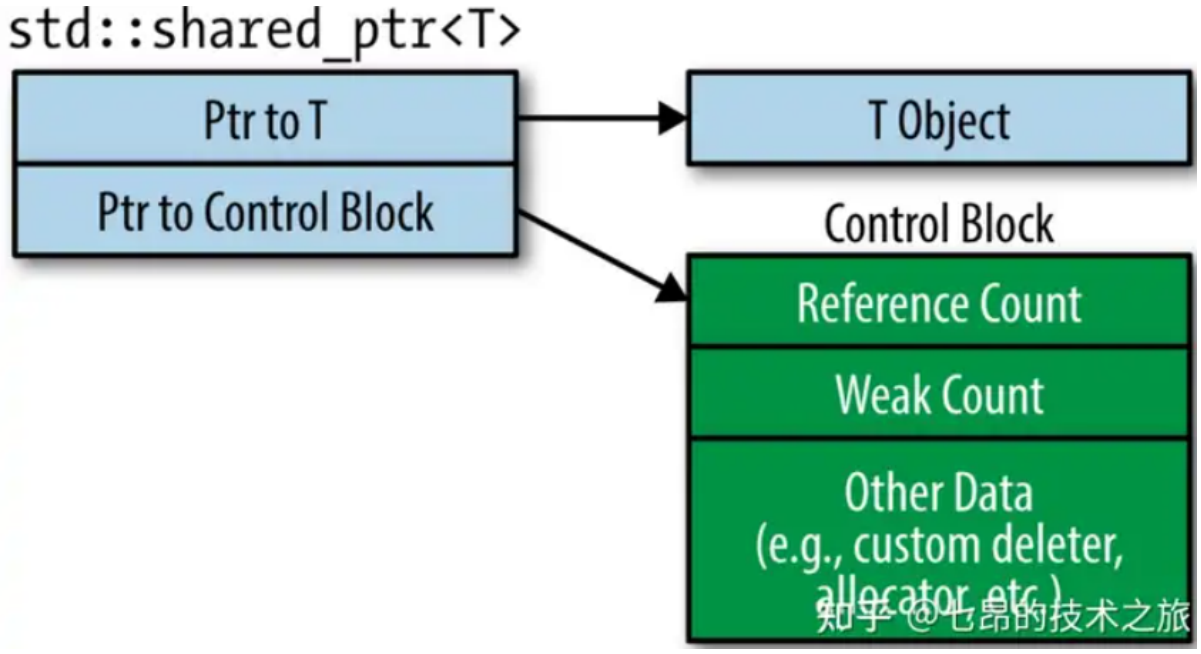
- new：new分配的内存块存放在堆上

3. 智能指针

[万字长文全面详解现代C++智能指针：原理、应用和陷阱 - 知乎 (zhihu.com)]

3.1 shared_ptr底层原理

```
element_type*    _M_ptr;         // Contained pointer.
__shared_count<_Lp>  _M_refcount;    // Reference counter.
```



`std::shared_ptr`在内部维护一个**引用计数**，其只有两个指针成员，一个指针是所管理的数据的地址；还有一个指针是控制块的地址，包括引用计数、weak_ptr计数、删除器(Deleter)、分配器(Allocator)。因为不同shared_ptr指针需要共享相同的内存对象，因此**引用计数的存储是在堆上的**。而unique_ptr只有一个指针成员，指向所管理的数据的地址。因此一个shared_ptr对象的大小是raw_pointer大小的两倍。

3.2 unique_ptr

独占性的实现：

```
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
```

4. vector

寻找最小元素

```
*min_element(sweetness.begin(),sweetness.end());
```

统计和：

```
accumulate(sweetness.begin(),sweetness.end(), 0)
```

5. 虚函数和多态

虚函数表指针（vptr）：当类中存在虚函数时，会自动创建隐形的虚函数表指针，该指针是成员变量，占用类对象的内存空间；

虚函数表（vtbl）：当类中存在至少一个虚函数时，在**编译**期间会自动创建虚函数表，在编译链接生成可执行文件后，类和其对应的虚函数表会被保存到可执行文件中，虚函数表会存储在全局/静态存储区里，可执行文件执行时一并被加载到内存中；

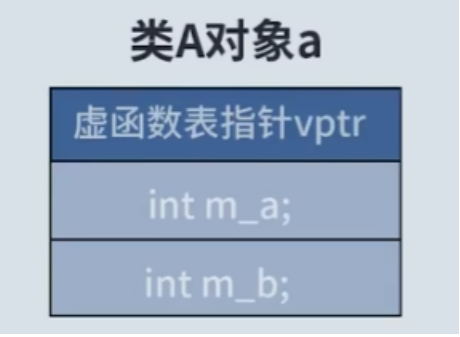
vptr被赋值的时机以及vptr和vbtl的关系：在**编译**期间，编译器会向类的构造函数中安插为vptr赋值的语句。当创建该类对象时会调用该类的构造函数，因为构造函数中有给vptr赋值的语句从而使vptr指向类的vtbl

```
//伪代码如下
A(){
    vptr = &A::vftable;  //使vptr指向类A的vtbl
    //......
}
```

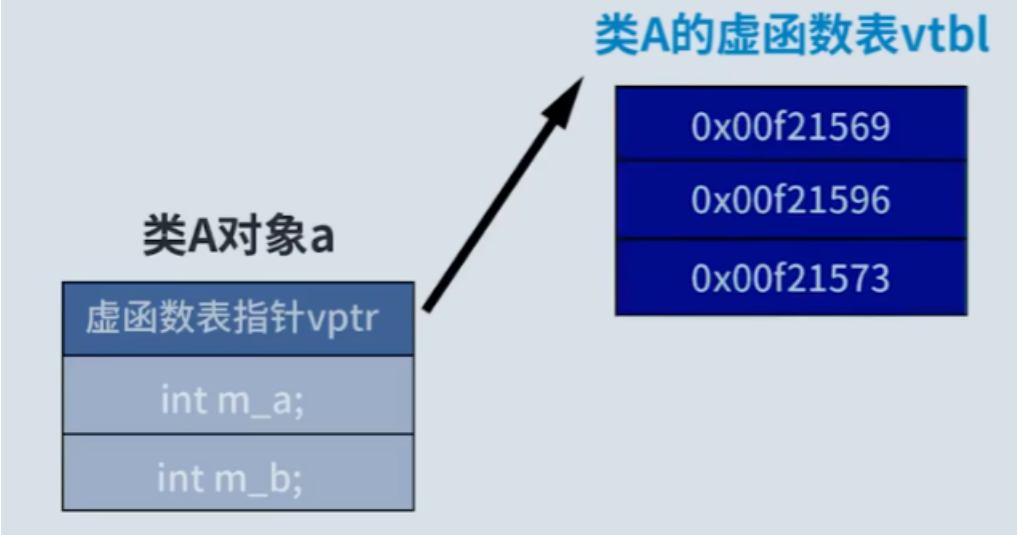
类对象在内存中的布局：

```
class A
{
public:
    virtual ~A(){}
    void func1(){}
    void func2(){}
    virtual void vfunc1(){}
    virtual void vfunc2(){}
private:
    int m_a;
    int m_b;
}
```

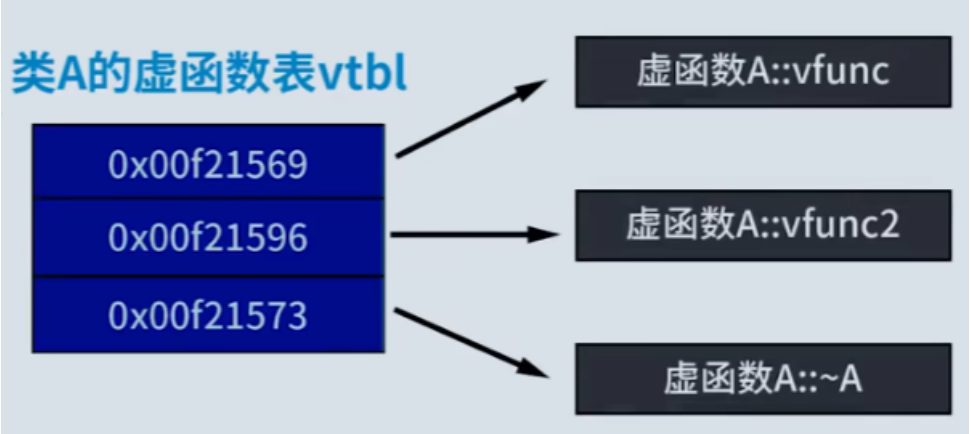
当生成A对象时，就可以看一看类A对象在内存中的布局：



编译器会向其中插入vptr，同时，vptr会指向类A的vtbl



虚函数表的三个指针分别指向类的三个虚函数vfunc1，vfunc和~A，它们都是类A的组成部分，不占用类A对象的内存空间



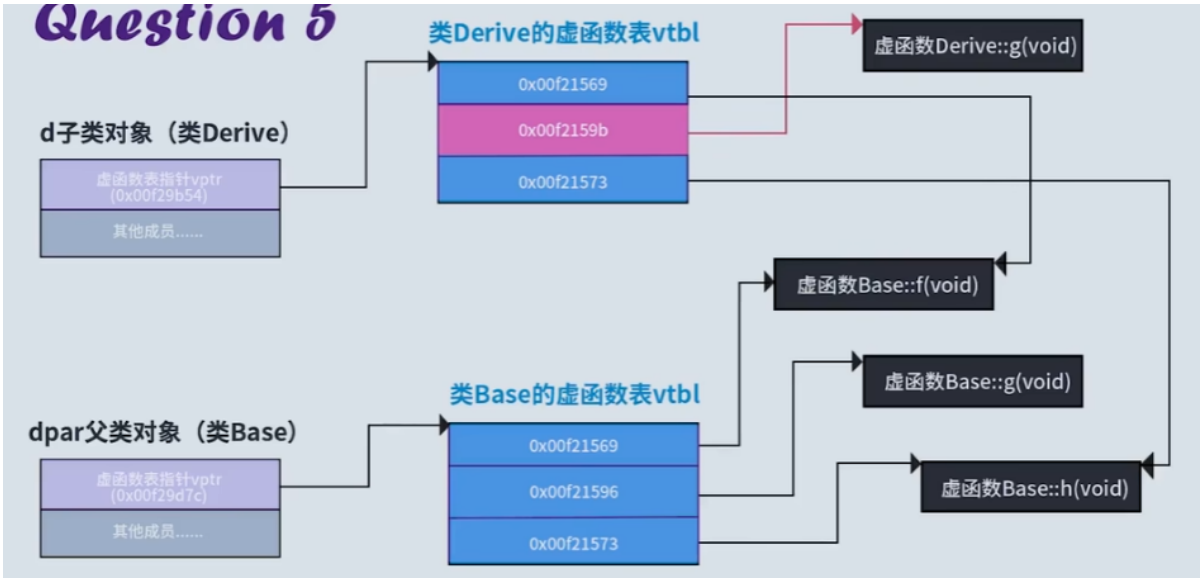
构造函数不能为虚函数：如果构造函数时虚函数，那么调用构造函数就需要去找vptr，而此时vptr还没有初始化，因此**构造函数不可以是虚函数。**

析构函数常常是虚函数：

- 若析构函数是虚函数，delete 时，基类和子类都会被释放；
- 若析构函数不是虚函数，delete 时，只有基类会被释放，而子类没有释放，存在内存泄漏的隐患。

多态性：函数重载是**静态多态**，**动态多态**是指父类有一个虚函数，子类中有该虚函数的重写，当通过父类指针new一个子类对象如 $\mathrm{Base}^{*}\ \mathrm{pBase} = \mathrm{new}\ \mathrm{A};$ ，或通过父类引用来绑定一个子类对象时，如果用这个父类指针来调用虚函数那么调用的其实是子类的虚函数。 动态多态必须存在**虚函数**，没有**虚函数**绝对不可能存在**动态多态。**

虚函数的工作原理和多态性的体现：



6. C++编译过程

编译的过程简单来说就是将程序的源代码翻译成CPU能直接运行的机器代码，编译是以**单个源文件为单位**，这也是为什么当修改已经编译好的项目的某些文件时，重新编译只有这些修改过的文件会被编译的原因。

编译后会变为.o文件，被称为目标文件（Object File），目标文件是一个二进制文件，Linux格式是**ELF**（Executable and Linkable Format，翻译：可执行并链接的格式），Windows格式是**PE**（Portable Executable）。

在编译一个文件时，它对于所调用的其他文件的函数的存在完全不知道，因此编译器只能将这两个函数的跳转地址暂时设为0，在链接过程中再去修正，为了让链接器能够定位到这些需要被修正的地址，.o文件中包括一个**重定位表**（Relocation Table）

在编译时，**最先做的是预处理（.i文件）**，该阶段会把所有的预处理语句进行处理（带#的指令），比如当看到#inlcude时，它指定了想要包含的文件，预处理器会打开该文件并将所有内容粘贴过来成为一个**翻译单元**

7. C++链接过程

链接器**解析符号引用**，遇到一个标识符时，它会首先在当前作用域内查找该名称。如果在当前作用域内找不到，编译器会向外层作用域逐级查找，直到找到为止。如果在全局作用域内仍未找到，链接器会报错。

静态链接：指将编译后的所有目标文件，连同用到的一些静态库、运行时库组合拼装成一个独立的可执行文件

动态链接：

静态链接和动态链接的区别：静态链接会将编译产生的所有目标文件连同用到的库合并成一个独立的可执行文件，其中会修正模块间函数的跳转地址（重定位），在**技术上更快**，因为链接器实际可以执行链接时优化，而且因为已经在链接过程中链接好了所以启动更快，但是静态链接也会导致exe文件过大且**一旦发生修改就要全部重新链接**，动态链接实际上将链接的整个过程推迟到程序加载的时候，当我们运行程序时，操作系统会将程序的数据，代码连同它用到的一系列动态库先递归的加载到内存，当动态库内

存地址被确定，就会修正动态库中的函数跳转地址（重定位），它在数据段专门预留一片区域用来存放函数的跳转地址，被称为全局偏移表（GOT, Global Offset Table）

问：所有的函数都经过链接器吗?

答：不，如果是宏函数或者内联函数就不需要，宏函数在预处理阶段展开，内联函数当编译器认为可以展开会在编译阶段展开，所以链接器就不会去探测这些函数

8. static

- **类或结构体之外使用static**

静态变量或函数意味着，当需要将这些函数或变量与实际定义的符号链接时，链接器**不会在这个翻译单元的作用域之外寻找对应符号定义**，它只会在它被声明的C++文件中被看到

- **类或结构体内使用static**

静态在类或结构体中意味着特定的东西。如果和变量一起使用，意味着这个静态变量在类的所有实例中只有一个实例，所以通过类实例来引用静态变量是没有意义的。要注意的是，**静态函数无法访问非静态变量**，因为静态函数没有this指针

- **局部作用域使用static**

声明一个变量需要考虑两种情况——变量的生存期和变量的作用域，生存期是指变量实际存在的时间，作用域是指我们可以访问变量的范围，静态局部变量允许声明一个变量，它的生存期基本相当于整个程序的生存期，但是它的作用范围被限制在某个局部作用域中

```
void Function(){
    static int i = 0;
    i++;
    cout<< i << endl;
}

int main(){
    Function(); //1
    Function(); //2

    return 0;
}
```

9. mutable

mutable指某些东西可以改变，一般和const或lambda表达式一起使用

- **和const一起使用**

指的是某种const但它实际上可以改变

通常，`const`成员函数承诺不会修改对象的状态，这意味着在这类函数内部不能修改成员变量的值。然而，有时候我们需要在`const`成员函数中修改一些成员变量，而这些变量的修改并不影响对象的逻辑状态或外部可观察行为。这类成员变量通常用于**缓存、懒惰计算或是跟踪函数调用**次数等功能。

```
class Rectangle {
public:
    Rectangle(double width, double height) : width_(width), height_(height),
area_(0.0), areaValid_(false) {}

    double area() const {
        if (!areaValid_) {
            // 可以在const函数内修改mutable成员
            area_ = width_ * height_;
            areaValid_ = true;
        }
        return area_;
    }

private:
    double width_;
    double height_;
    mutable double area_;  // 使用mutable修饰
    mutable bool areaValid_;
};
```

- **和lambda一起使用**

在C++中，lambda表达式默认捕获的变量是`const`的，这意味着你不能修改这些捕获的变量。当你在lambda表达式后添加`mutable`关键字时，它允许你在lambda函数体内修改通过值捕获的变量。

```
int x = 4;
auto lambda = [x]() mutable { x++; };  // 现在编译通过，可以在lambda内部修改x
```

需要注意的是，这里修改的`x`是捕获到lambda表达式内部的一个**副本**，原始的`x`并没有被修改。如果你想修改原始的`x`，你需要通过引用捕获它：

```
int x = 4;
auto lambda = [&x]() { x++; };  // 通过引用捕获x，然后在lambda内部修改它
lambda();
std::cout << x;  // 输出5，因为原始的x被修改了
```

`mutable`关键字在lambda表达式中的**使用场景**包括：

- 当你需要在lambda表达式内部改变通过值捕获的变量的状态时。
- 当使用的lambda表达式需要模拟有状态的函数对象，而该状态仅在lambda表达式内部有意义时。

10. extern

- **使用`extern`声明全局变量**

在一个文件中定义一个全局变量时，可以在其他文件中使用`extern`关键字来声明这个变量，而不需要重新定义它。这样，多个文件就可以共享和访问这个全局变量。

```
// file1.cpp
int globalVar = 42;

// file2.cpp
extern int globalVar;  // 声明，不是定义
```

- **使用`extern`声明函数**

对于函数，`extern`是**默认**的，所以你通常不需要显式地使用`extern`关键字来声明函数。函数的声明告诉编译器函数的存在，它的定义可以在程序的任何地方出现

```
// file1.cpp
void myFunction() {
    // 函数实现
}

// file2.cpp
void myFunction();  // 可以省略extern，因为对函数而言它是默认的
```

- **extern "C"**

extern还有另一个用法是与"C"一起使用，形成**extern "C"**。这是用于C++代码调用C语言代码的情况，它告诉编译器在连接时使用C语言的名称修饰规则（Name Mangling），而不是C++的。这对于在C++项目中调用C语言库非常有用。

```
extern "C" {
    #include "c_header.h"  // 假设这是一个C语言的头文件
}

extern "C" void cFunction();  // 告诉编译器这是一个C语言函数
```

12. explicit

c++默认支持一次隐式转换

```
class Entity{
private:
    string name;
    int age;
public:
    Entity(string i_name) : name(i_name), age(-1){}
    Entity(int i_age) : name("Unknown"), age(i_age){}
};

void PrintEntity(const Entity& entity){
    //do something
}

int main(){
```

```
    Entity a("Sam");
    Entity a(23);
    PrintEntity(Entity(22));
    PrintEntity(Entity("Sam"));

    //可以使用c++默认的一次隐式转换
    Entity a = "Sam";
    Entity a = 23;
    PrintEntity(22);
    PrintEntity(std::string("Sam"));
    //PrintEntity("Sam")  会报错，因为做了两次隐式转换，const char数组 --> string --> Entity
}
```

explicit则**禁用了该功能**

如果在**构造函数**前添加该关键词，则无法隐式调用该构造函数，必须显示调用

11. 函数指针

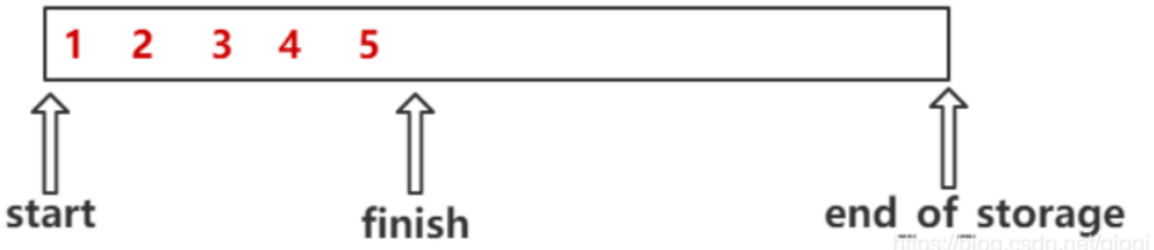
函数指针可以用于将一个函数作为参数传递给另一个函数，常用于回调函数，回调函数允许一个函数在特定事件或条件发生时由另一个函数调用。

12. STL 容器的底层实现

C++STL的容器的底层实现详解_c++ stl 各种容器底层实现-CSDN博客

12.1 vector

动态数组，一个指针指向start，一个指针指向当前存储数据的末尾，一个指针指向capacity（）的末尾，当数组满时发生扩容，扩容大小为原来的1.5倍或2倍




```
iterator start;                 // 指向底层空空间的起始位置
iterator finish;                // 指向最后一个有效元素的下一个位置，没有元素时与start在同一位置
iterator end_of_storage;        // 指向空间的末尾
```

12.2 deque

C++ STL deque 容器底层实现原理（深度剖析） - 知乎 (zhihu.com)

和 vector 容器采用连续的线性空间不同，**deque 容器存储数据的空间是由一段一段等长的连续空间构成，各段空间之间并不一定是连续的，可以位于在内存的不同区域。**

为了管理这些连续空间，**deque 容器用数组（数组名假设为 map）存储着各个连续空间的首地址。**

也因此，它的迭代器就比较复杂：

```
template<class T,...>
struct __deque_iterator{
    ...
    T* cur;
    T* first;
    T* last;
    map_pointer node;//map_pointer 等价于 T**
}
```

- cur：指向当前正在遍历的元素；
- first：指向当前连续空间的首地址；
- last：指向当前连续空间的末尾地址；
- node：它是一个二级指针，用于指向 map 数组中存储的指向当前连续空间的指针。

```
iterator start;
iterator finish;
map_pointer map;
```

start 迭代器记录着 map 数组中首个连续空间的信息，finish 迭代器记录着 map 数组中最后一个连续空间的信息。另外需要注意的是，和普通 deque 迭代器不同，start 迭代器中的 cur 指针指向的是连续空间中首个元素；而 finish 迭代器中的 cur 指针指向的是连续空间最后一个元素的下一个位置。

12.3 list

离散存储

12. 指针和引用的区别

13. new和malloc的区别，第三种分配内存的方法（placement new）

13.1 new

- **类型安全**：`new`是C++中引入的操作符，它知道它所分配的对象的类型，并返回该类型的正确指针，无需进行类型转换。
- **构造函数和析构函数**：`new`在分配内存的同时会调用对象的构造函数，为对象进行初始化。相应地，`delete`操作符会调用对象的析构函数。
- **异常处理**：如果`new`无法分配所请求的内存，它会抛出一个`std::bad_alloc`异常，而不是像`malloc`那样返回`nullptr`。
- **用法**：`new`的用法更符合C++的面向对象特性。

13.2 malloc

- **C语言遗留**：`malloc`是从C语言继承来的函数，它仅分配内存，不调用构造函数。
- **返回类型**：`malloc`返回一个`void*`类型的指针，需要显式转换为需要的类型。
- **错误处理**：如果内存分配失败，`malloc`返回`nullptr`。
- **内存释放**：使用`malloc`分配的内存应通过`free`函数释放，而不是`delete`。

```
MyClass* obj = (MyClass*)malloc(sizeof(MyClass));
```

- malloc只分配内存，不调用构造函数，如果使用malloc创建类或结构体需要使用placement new

```
MyClass* obj = (MyClass*)malloc(sizeof(MyClass));
if (!obj) {
    // 处理内存分配失败的情况
}
new (obj) MyClass();
obj->~MyClass();  // 显式调用析构函数
free(obj);        // 释放内存
```

13.3 placement new

placement new 在已分配的特定内存创建对象；其既可以在栈上生成对象，也可以在堆上生成对象

```
//在栈上分配
char mem[100];

cout << (void*)mem << endl;
A* p = new (mem)A;

p->~A();

//在堆上分配
char* buffer = new char[sizeof(MyClass)];  // 分配足够的原始内存
MyClass* obj = new (buffer) MyClass();  // 在buffer指向的内存上构造对象
```

13.4 malloc底层实现原理

14. 字节对齐

内存对齐的目的：使得内存获取数据更快，**空间换时间的思想**

15. 缓存算法

- **LRU（最近最久未使用，Least Recently Used）算法**

LRU算法基于这样一种假设：如果数据最近被访问过，那么将来被再次访问的概率也很高。因此，LRU算法会优先淘汰那些最长时间未被访问的数据。

实现方法：

- 一个常见的LRU缓存实现方式是使用一个哈希表加上一个双向链表。哈希表用于保证查找速度，双向链表用于记录元素的访问顺序，链表头部是最近访问的，尾部是最久未访问的。
- 当一个新数据被访问时，如果缓存未满，将其添加到链表头部；如果缓存已满，则删除链表尾部的数据，并将新数据添加到头部。
- 如果缓存中的数据被再次访问，这个数据会被移动到链表头部。

应用场景： 适用于最近的数据访问模式较为频繁的情况。

```
struct DLinkedNode{
    int key, val;
    DLinkedNode* prev, * next;
    DLinkedNode() : key(0), val(0), prev(nullptr), next(nullptr){}
    DLinkedNode(int i_key, int i_val) : key(i_key), val(i_val), prev(nullptr),
next(nullptr){}
};

class LRU{
public:
    LRU(int i_capacity){
        capacity = i_capacity;
```

```
        size = 0;
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head->next = tail;
        tail->prev = head;
    }

    int get(int i_key){
        if(umap.find(i_key) != umap.end()){
            MoveToHead(umap[i_key]);
            return umap[i_key]->val;
        }
        else return -1;
    }

    void put(int key, int value){
        if(umap.find(key) != umap.end()){
            umap[key]->val = value;
            MoveToHead(umap[key]);
        }
        else{
            DLinkedNode* newNode = new DLinkedNode(key, value);
            AddToHead(newNode);
            if(size == capacity){
                DLinkedNode* TailNode = DeleteTail();
                umap.erase(TailNode->key);
            }
            else{
                size++;
            }
        }
    }

    void AddToHead(DLinkedNode* node){
        node->next = head->next;
        node->prev = head;
        head->next->prev = node;
        head->next = node;
    }

    void MoveToHead(DLinkedNode* node){
        node->prev->next = node->next;
        node->next->prev = node->prev;
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
        node->prev = head;
    }

    DLinkedNode* DeleteTail(){
        DLinkedNode* TailNode = tail->prev;
        TailNode->prev->next = tail;
        tail->prev = TailNode->prev;

        return TailNode;
    }
```

```
private:
    int capacity, size;
    DLinkedNode* head, * tail;
    unordered_map<int, DLinkedNode*> umap;
}
```

- **FIFO（First In First Out）算法**

FIFO算法是最简单的缓存算法，按照数据进入缓存的顺序来淘汰数据。最先进入缓存的数据，当缓存满时，也将是最先被淘汰的。

实现方法：

- 可以使用一个队列来实现FIFO缓存。新访问的数据被添加到队列的末尾。
- 当缓存达到最大容量时，队列头部的数据（即最早进入缓存的数据）会被移除。

应用场景： FIFO算法实现简单，但不考虑数据的访问模式，因此适用性较为有限，可能用在一些对缓存淘汰策略要求不高的场景。

- **LFU（最近最少使用算法，Least Frequently Used）算法**

LFU算法根据数据的访问频次来进行淘汰，优先淘汰访问频次最低的数据。与LRU不同，LFU关注的是访问的频率。

实现方法：

- LFU缓存的实现较为复杂，通常需要维护一个按访问频率组织的数据结构，比如最小堆，以及一个哈希表来存储频率。
- 每次访问数据时，数据的访问频率会更新，数据结构也随之调整以保持正确的顺序。
- 当缓存满时，频率最低的数据会被淘汰。

应用场景： 适用于需要根据长期访问模式来优化的场景，但由于其实现复杂度较高，不适用于访问模式快速变化的环境。

- **ARC（自适应缓存替换，Adaptive Replacement Cache）算法**

ARC算法结合了LRU和LFU的特点，自适应地调整对最近使用和频繁使用的数据的偏好。它维护两个LRU列表，一个记录最近访问的数据，另一个记录频繁访问的数据，并动态地调整这两个列表的大小。这种自适应特性使得ARC能够在不同的工作负载下都表现出良好的性能。

- **2Q算法**

2Q算法是一种结合了FIFO和LRU优点的缓存替换算法。它将缓存分为两个队列，一个是FIFO队列，另一个是LRU队列。新访问的数据首先进入FIFO队列，如果数据在FIFO队列中被再次访问，则将其移动到LRU队列。当需要淘汰数据时，首先从FIFO队列中淘汰，这种方法既考虑了数据的新鲜度也考虑了数据的访问频率。

16. 哈希冲突

链地址法：

开放地址法：

再哈希法：

公共溢出区

17.宏函数和内联函数

- ## 内联函数

内联函数是C++的一个特性，它向编译器建议在每个调用点上“内联地”展开函数体，而不是进行常规的函数调用。这意味着编译器会在**编译时**尝试将函数调用替换为函数体本身的副本。这样做的主要目的是**减少函数调用**的开销，尤其是对于小型、频繁调用的函数。

特点：

- **类型安全**：内联函数是类型安全的，它们遵循C++的类型检查。
- **自动处理**：内联是一种请求，编译器可以选择忽略这个请求。编译器会根据函数的复杂性和调用上下文来决定是否内联。
- **构造和析构**：适用于构造函数和析构函数，可以管理对象的生命周期。
- **调试友好**：虽然内联函数使得调试更复杂，但它们通常比宏更容易调试，因为它们遵循正常的作用域和类型规则。

```
inline int Max(int x, int y) {
    return (x > y) ? x : y;
}
```

- ## 宏函数

宏定义是预处理器的指令，用于在编译之前替换代码中的文本。宏可以用来定义常量、表达式、代码片段等。由于宏的替换发生在预处理阶段，所以它不受C++语言规则的约束，也不进行类型检查。

特点：

- **文本替换**：宏仅仅是对文本的简单替换，并不是函数调用。
- **无类型安全**：宏不进行类型检查，易于引发类型相关的错误。
- **作用域**：宏没有作用域的概念，它们在定义后直到文件结束或被 `#undef` 指令取消定义之前都是有效的。
- **调试困难**：宏扩展可能导致代码难以理解和调试，因为宏展开后的代码可能与原始代码大不相同。

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

- ## 内联函数 vs. 宏定义

- **编译器优化**：内联函数的内联是由编译器决定的，编译器可以根据情况进行优化；而宏定义在预处理阶段就进行了文本替换，不受编译器控制。
- **类型检查和安全性**：内联函数在编译时进行**类型检查，更安全**；宏定义不进行类型检查，可能会引入错误。

- **作用域和命名空间**：内联函数遵循C++的作用域和命名空间规则，而宏定义没有作用域概念，可能会**导致命名冲突。**
- **调试**：**内联函数比宏定义更容易调试**，因为宏展开可能会让代码变得难以理解。

18. Union

Union一次只占用一个成员的内存，以结构体做对比，一个声明有4个浮点数的结构体，它的大小是4*4 = 16字节，当不断向类或结构体添加更多成员时，其大小会不断增长

Union不能使用虚函数，当想要给同一个变量取不同名字时Union非常有用

```
struct Vector2{
    float x, y;
};

struct Vector3{
    float x, y, z;
};

struct Vector4{
    union{
        struct{
            float x, y, z, w;
        };
        struct{
            vector2 a, b; //a和x、y共享数据，b和z、w共享数据
        };
    };
};

void PrintVector2(const Vector2& vector){
    std::cout << vector.x << ", " << vector.y << std::endl;
}

int main(){
    Vector4 vector = { 1.0f, 2.0f, 3.0f, 4.0f};
    PrintVector2(vector.a); //1.0, 2.0
    PrintVector2(vector.b); //3.0, 4.0
    vector.z = 400.0f;
    vector.x = 111.0f;
    PrintVector2(vector.a); //111.0, 2.0
    PrintVector2(vector.b); //400.0, 4.0

    return 0;
}
```

19. 类型转换

C++是强类型系统，所以类型不能随意转换，C++可以为简单类型进行隐式类型转换，比如int，double等之间的转换，但是对于类类型等的转换则需要强制类型转换

c++类型强制类型转换

- static_cast
- reinterpret_cast
- dynamic_cast
- const_cast

```
class Base{
public:
    Base(){}
    virtual ~Base(){}
};

class Derived : public Base{
public:
    Derived(){}
    ~Derived(){}
};

class AnotherClass : public Base{
public:
    AnotherClass(){}
    ~AnotherClass(){}
};
```

- **dynamic_cast:**

  当想做特定类型的类型转换时，使用起来更像函数，不像编译时进行的类型转换，而是在运行时计算，因此，它有一定的运行成本。

  dynamic_cast是专门用于沿继承层次结构进行的强制类型转换，只支持含有虚函数的类进行类型转换，可以用它来检查一个对象是否是给定类型。转换失败会返回nullptr

```
class Entity{
public:
    virtual void DoSomething(){}
};

class Player : public Entity{

};

class Enemy : public Entity{

};

int main(){
    Entity* player = new Player();
    Entity* enemy = new Enemy();

    Player* p1 = dynamic_cast<Player*>(player); //成功，拿到转换结果
```

```
    Player* p2 = dynamic_cast<Player*>(enemy);  //失败，返回nullptr

    return 0;
}
```

问：dynamic_cast怎么知道的?

答：因为它储存了运行时类型信息（RTTI，Runtime Type Information），RTTI存储了所有类型的运行时信息

20. 类与结构体

C++类和结构体几乎没有区别，只有一个关于可见度的小区别，即类的成员默认为private，结构的成员默认为public。但是即使技术上讲它们没有太大区别，但是实际的使用场景却不一样，比如当只是来包括一堆变量来表示某种结构时可能更倾向于使用struct，而当需要封装某些功能时更倾向于使用class

21. ENum

一种命名值的方法，当有需求使用整数来表示某些状态或数值时，希望能给对应整数指定名称以便代码更易于阅读

- ## 基本枚举

基本的`enum`类型在C++中定义了一组命名的整数常量。例如：

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

- ## 枚举的使用

你可以像使用普通整数类型一样使用枚举类型的变量：

```
Color myColor;
myColor = RED;
if (myColor == RED) {
    // Do something
}
```

- ## 基于作用域的枚举（C++11）

C++11引入了一种新的枚举类型称为“基于作用域的枚举”（也称为`enum class`），它提供了更好的类型安全。与传统的`enum`相比，`enum class`的枚举值不会隐式转换为整数，而且它们的名字需要以枚举类型为前缀：

```
enum class Color {
    RED,
    GREEN,
    BLUE
};

Color myColor = Color::RED;  // 注意作用域解析符(::)
```

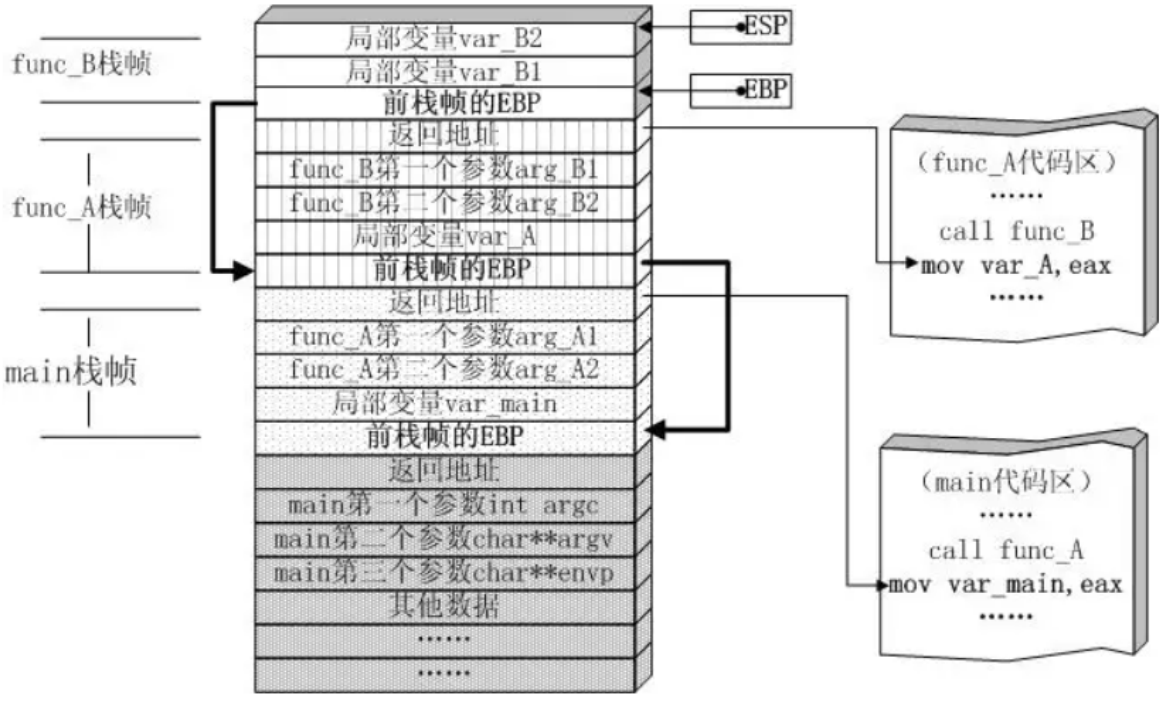
- **注意事项**

- 传统的`enum`类型的枚举值可能会导致名字冲突，因为它们实际上是在封闭作用域中定义的整数常量。
- `enum class`提供了更好的封装，枚举值必须使用作用域解析符进行访问，但这也意味着它们的使用稍微繁琐一些。

21. 函数调用

C/C++函数调用的压栈模型_c++函数调用的压栈过程-CSDN博客

- **参数入栈**：将参数从右向左依次压入系统栈中
- **返回地址入栈**：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
- **代码区跳转**：处理器从当前代码区跳转到被调用函数的入口处
- **栈帧调整**：具体包括：
    - 保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）
    - 将当前栈帧切换到新栈帧。（将ESP值装入EBP，更新栈帧底部）
    - 给新栈帧分配空间。（把ESP减去所需空间的大小，抬高栈顶）



22 线程

```cpp
#include <iostream>
#include <thread>

static bool s_Finished = false;

void DoWork(){
    using namespace std::literals::chrono_literals;

    while(!s_Finished){
        std::cout<<"Working...\n";
        std::this_thread::sleep_for(1s);
    }
}

int main(){
    std::thread worker(DoWork);  //创建worker这个线程，它的任务是执行DoWork里的代码

    std::cin.get();
    s_Finished = true;

    worker.join();    //等待worker这个线程运行完，在这之前阻塞
    std::cin.get();
}
```

23. 左值和右值

23. 1完美转发

```cpp
void process(int& value) { /* 对左值的处理 */ }
void process(int&& value) { /* 对右值的处理 */ }

template <typename T>
void forwarder(T&& arg) {
    process(std::forward<T>(arg));
}
```

24. 移动语义

移动语义本质上允许我们移动对象

```cpp
class String{
public:
    String() = default;
    String(const char* string){
        cout<<"Created!\n";
        m_size = strlen(string);
        m_Data = new Char[m_size];
```

```
        memcpy(m_Data, string, m_size);
    }
    //移动构造函数
    String(String&& other) noexcept{
        cout<<"Moved!\n";
        m_size = other.m_size;
        m_Data = other.m_Data;

        othrt.m_size = 0;
        other.m_Data = nullptr;  //因为移动构造函数结束后会调用other的析构函数，所以要先将
other的指针成员变量置空
    }

    //拷贝构造函数
    String(const String& other){
        cout<<"Copied!\n";
        m_size = other.m_size;
        m_Data = new Char[m_size];
        memcpy(m_Data, other.m_Data, m_size);
    }

    ~String(){
        cout<<"Destroyed!\n";
        delete[] m_Data;
    }

    void Print(){
        for(uint32_t i = 0; i<m_size; i++){
            cout<<m_Data[i];
        }
        cout<<"\n";
    }

private:
    char* m_Data;
    uint32_t m_size;
}

class Entity{
public:
    Entity(const String& name):m_Name(name){}
    Entity(String&& name):m_Name(move(name){}

    void PrintName(){
        m_Name.Print();
    }
private:
    String m_Name;
}

int main(){
    Entity entity(String("Samuel"));
    entity.PrintName();

    cin.get();
}
```

25. 初始化列表

初始化列表必须按照声明的顺序写，否则**可能导致依赖性**等问题。原因可参考下面代码：

```
#include <iostream>

class MyClass {
public:
    int num1;
    int num2;

    MyClass() : num2(10), num1(num2 + 1) { // 注意，这里先初始化了num2，然后是num1
        // 构造函数体
    }

    void print() {
        std::cout << "num1: " << num1 << ", num2: " << num2 << std::endl;
    }
};

int main() {
    MyClass obj;
    obj.print(); //打印结果：num1: -858993459, num2: 10

    return 0;
}
```

在这个例子中，虽然初始化列表试图先初始化`num2`为10，然后初始化`num1`为`num2 + 1`，按理来说`num1`应该是11。但是，因为`num1`在`num2`之前声明，所以`num1`**会先被初始化。由于**`num2`**此时还没有被初始化**，`num1`的初始化实际上依赖了一个未定义的值（`num2`的值）。这将导致`num1`的值是未定义的，程序的行为是不可预测的。

使用初始化列表的好处：

1. **提高效率**：
    - 对于非POD（Plain Old Data，即没有构造函数、析构函数和虚函数的简单数据类型）类型的成员变量，使用初始化列表可以直接调用构造函数来初始化成员变量，避免了先默认构造然后再赋值的额外开销。这在成员变量类型为类类型且没有默认构造函数或者构造代价较高时尤其重要。
2. **构造函数的直接初始化**：
    - 使用初始化列表可以直接调用成员变量的构造函数进行初始化，包括调用非默认构造函数。这在需要传递参数给成员变量的构造函数或初始化const和引用类型成员时尤其重要，因为这些类型的成员只能在初始化列表中被初始化。
3. **避免二次初始化**：
    - 如果不使用初始化列表，类的成员变量将首先被默认初始化，然后在构造函数体中被赋予新值。这意味着成员变量实际上被初始化了两次。对于某些类型的成员变量（如类对象），这可能导致不必要的性能开销。
4. **初始化const和引用成员**：

- const成员和引用成员必须在构造函数的初始化列表中初始化，因为它们一旦被默认初始化后就不能再被赋值。初始化列表提供了一种在对象创建时立即设置这些成员值的方法。

5. **清晰的依赖关系表达**：
   - 初始化列表清楚地表明了成员变量的初始化顺序和依赖关系，增加了代码的可读性。它允许开发者一目了然地看到哪些成员变量是被直接初始化的，哪些是通过构造函数参数初始化的。
6. **支持非默认构造类型的成员**：
   - 某些类型的对象，如某些STL容器，可能没有默认构造函数，或者默认构造函数不是最优选择。使用初始化列表可以直接调用这些成员的非默认构造函数，提供更多的灵活性和控制。

26. const

const定义之后的变量存放在内存的哪个区域

1. **全局或静态`const`变量**：通常存储在程序的数据段（data segment），特别是在它的只读部分（如`.rodata`区域），因为它们是全局的或静态的，其生命周期贯穿整个程序执行过程。
2. **局部`const`变量**：通常存储在栈（stack）上，就像其他局部变量一样。尽管它们被声明为`const`，但它们的存储位置并没有变化，仍然是在每次函数调用时分配在栈上。
3. **`const`表达式**：编译器可能会在编译时对`const`表达式进行优化，将它们直接替换为具体的值，而不分配任何存储空间。这种情况下，`const`变量实际上存在于程序的代码段（code segment）中，作为指令的一部分。
4. **`const`成员变量**：如果是类中的`const`成员变量，它们的存储位置取决于对象的存储位置。如果对象是全局的或静态的，`const`成员变量可能存储在数据段；如果对象是局部的，`const`成员变量则存储在栈上；如果对象是通过`new`分配的，则`const`成员变量存储在堆（heap）上。
5. **编译期`const`变量**：在某些情况下，如果`const`变量在编译时就已知，并且从未通过地址取用，编译器可能会将这些变量视为编译期常量，不为它们分配存储空间，而是在需要它们的地方直接替换为其值。