

使用IDE

VS文档: <[Visual Studio 文档](#)|[Microsoft Learn](#)>

图标	说明	图标	说明
	命名空间		方法或函数
	类		运算符
	接口		属性
	结构		字段或变量
	Union		事件
	枚举		返回的常量
	TypeDef		枚举项
	模块		映射项
	扩展方法		外部声明
	委托		错误
	异常		模板
	映射		Unknown
	类型转发		

生成配置

Debug和Release版本

- Debug调试版本：包括了调试信息，并且不做任何优化，便于程序员调试程序
- Release发布版本：编译器进行了各种优化，使得程序在代码大小和运行速度上都是最优的，以使用户使用，但不能进行调试

只有DEBUG版的程序才能设置断点、单步执行、使用 TRACE/ASSERT等调试输出语句。
release不包含任何调试信息，所以体积小、运行速度快

对于x86来说，Debug和Release编译生成的obj和链接后可执行文件会分别放在Debug和Release文件夹中；而x64则是将Debug和Release分别放在名为x64的文件夹中

项目结构

如何组织一个项目？

新建的vs工程的配置文件主要包括两部分：Solution（解决方案）和Project（工程）配置文件

一个解决方案里可能包含多个工程。**每个工程是一个独立的软件模块，比如一个程序、一个代码库等。**这样的好处是解决方案可以共享文件和代码库

解决方案文件

VS 采用两种文件类型（.sln 和 .suo）来存储解决方案设置，下面提到的除了 .sln 之外的文件都放在 .vs/解决方案名/版本号/ 中

- *.sln Visual Studio.Solution 环境提供对项目、项目项和解决方案项在磁盘上位置的引用，可以将它们组织到解决方案中。比如是生成Debug还是Release，是通用CPU还是专用的等。*.sln 文件可以在开发小组的开发人员之间共享。*.sln 就是打开文件的索引，正确引导用户进入环境、进入工程
- *.suo Solution User Operation 解决方案用户选项，记录所有将与解决方案建立关联的选项，以便在每次打开时，它都包含用户所做的自定义设置。比如说VS窗口布局、项目最后编译的而又没有关掉的文件在下次打开时用，打的断点等。注意：*.suo 文件是用户特定的文件，不能在开发人员之间共享

sdf 和 ipch文件与VS提供的智能感知、代码恢复、团队本地仓库功能有关，如果不需要，可以禁止，就不会产生sdf 和 ipch这两个文件了，VS重新加载解决方案时速度会加快很多。另外这两个文件会导致VS工程变得很大，如果此时用git进行管理，git中的管理文件也会变得很大

- *.sdf 文件：SQL Server Compact Edition Database File（.sdf）文件，是工程的信息保存成了数据库文件。sdf文件是VS用于intellisense的

- 若没有参加大型的团队项目，不会涉及到高深的调试过程，这个文件对于用户来说是没什么用的，可以放心删除。若后来又需要这个文件了，只打开工程里的 `.sln` 文件重新编译链接就ok了
- 同时我们注意到，当我们打开工程的时候还会产生一个 `*.opensdf` 的临时文件，不需关心，该文件关闭工程就会消失，是update `*.sdf` 文件的缓冲。如果完全不需要，也觉得sdf文件太大，那么可以：在Visual Studio里进入如下设置：进入“Tools > Options”，选择“Text Editor > C/C++ > Advanced”，然后找到“Fallback Location”。然后把“Always use Fallback Location”和“Do Not Warn if Fallback Location”设置成“True”。这样每次打开工程，不会再工程目录生成 `*.sdf` 文件了
- VS2015之后生成的 `*.db` 文件是sqlite后端用于intellisense的新数据库，相当于之前的 `*.sdf` SQL Server Compact数据库。它与VS2015提供的智能感知、代码恢复、团队本地仓库功能有关，VS重新加载解决方案时速度超快
- ipch文件夹：用来加速编译，里面存放的是precompiled headers，即预编译好了的头文件
- 禁止这两类文件生成的设置方法是：工具 -> 选项 -> 文本编辑器 -> C/C++ -> 高级，把回退位置和警告设置为true或者禁用数据库设为true，这样就不会产生那个文件了

上面的文件只是起一个组织的作用，将各个信息凝聚在一起，从而形成一个解决方案。不要随意的删掉着写看似没用的文件，删掉代码也不会丢失，但是，有时候环境配置好后，使用也比较方便，对于这两个文件，没必要动它。为了减少项目文件的大小，和这两个文件没有关系，但是如果操作不当，会导致解决方案打不开。那么解决办法就只有重建项目，然后导入代码文件了，只是会浪费一些时间而已，又要重新组织项目文件

工程配置文件

Project的配置文件种类主要包

括：`*.vcxproj`、`*.vcxproj.filters`、`*.vcxproj.user`、`*.props`。（注意区分`*.vcproj`和`*.vcxproj`的区别，前者是vs2008及以前版本的工程配置文件，后者是vs2010及以后的工程配置文件）

- `*.vcxproj` 文件是真正的项目配置文件，以**标准XML格式**的形式记录了工程的所有配置，如包含的文件名、定义的宏、包含的头文件地址、包含的库名称和地址、系统的种类等等。此外，还可以使用过滤条件决定配置是否有效

- *.vcxproj.filters 文件是项目下文件的虚拟目录，用于组织项目中源代码文件的视图层次结构的XML文件。它定义了 Visual Studio 中的解决方案资源管理器中如何显示和组织项目文件。该文件通常包含项目中的文件夹结构和源代码文件的过滤器（例如，源文件、头文件、资源文件等）。通过在 *.vcxproj.filters 文件中定义过滤器，可以在 Visual Studio 中更好地组织和浏览项目文件
- *.vcxproj.user 是XML格式的用户配置文件，用于保存用户个人的数据，比如配置 debug 的环境 PATH 等等。用于存储针对特定用户的项目设置。这些设置通常包括编译器选项、调试器设置、运行时环境等。每个用户在打开或修改项目时，可以在该文件中保存自己的首选项和个性化设置
- *.props 是属性表文件，用于保存一些配置，可以根据需求，导入到项目中使用。使用起来很灵活，比如使用一个开源库，我们新建一个工程，往往需要做不少配置，如果不用属性表文件的话，那么我们每次创建一个工程都要配置一遍，太浪费时间了。如果我们能将配置保存起来，每次新建项目将配置加进来就好了，属性表文件就很好的实现了这一点

IntelliSense

IntelliSense 是一项由 Microsoft 开发的智能代码补全和代码提示功能，旨在提高开发人员在集成开发环境（IDE）中编写代码的效率和准确性。它在多个 Microsoft IDE（如 Visual Studio、Visual Studio Code）和其他编辑器中得到广泛支持。

IntelliSense 使用静态代码分析、语义分析和用户输入上下文来为开发人员提供有关代码的实时信息和建议。它的主要功能包括：

1. 代码自动补全：IntelliSense 会根据正在输入的代码上下文，提供相关的代码补全选项。它可以自动完成代码片段、类、函数、变量等，并显示对应的参数列表和函数签名。
2. 代码导航：IntelliSense 可以帮助开发人员快速浏览代码库，并提供与代码相关的导航功能。这包括跳转到定义、查看函数调用层次结构、查找引用等。
3. 实时错误检查：IntelliSense 可以在代码编写过程中进行实时的语法和语义错误检查，并显示相应的错误和警告。这样可以帮助开发人员及早发现和修复问题，提高代码质量。
4. 文档注释：IntelliSense 可以显示与代码相关的文档注释、函数说明和参数描述，使开发人员能够更好地理解代码的含义和使用方式。
5. 提示和上下文帮助：IntelliSense 可以根据用户输入的上下文，提供有关可用选项的提示和帮助。它可以显示函数签名、参数类型、属性和方法列表等信息，以便开发人员更准确地编写代码

C++对C语言缺陷的修正

C++版本

C with classes -> C++1.0 -> ... -> C++98 (C++标准第一个版本, 引入STL库) -> ... -> C++11 (增加了许多特性, 使得C++更像是一种新语言) -> ... C++20 (自C++11以来最大的发行版, 引入了许多新的特性)

命名空间 *namespace*

在大型的Project中, 同一个作用域中可能存在大量同样命名的变量/函数/结构体等, C编译器无法解决这种冲突, C++通过命名空间解决了该冲突

定义命名空间

```
1 namespace wjF {
2     int rand = 1; // 定义变量
3     int Swap (int* left, int* right) { // 定义函数
4         int tmp = *left;
5         *left = *right;
6         *right = tmp;
7     }
8     struct Node { // 定义结构体或类
9         struct Node* next;
10        int val;
11    };
12 }
```

- 命名空间嵌套

```

1 namespace N1 {
2     // 各种定义...
3     namespace N2 {
4         // 各种定义...
5     }
6 }

```

- 同一个工程中允许多个相同名称的命名空间，编译器最后会合成为同一个命名空间

命名空间的展开和使用

- 不展开，加命名空间/指定空间访问

```

1 // 指定空间访问的操作符为 ::
2 std::cout << "Hello World!" << std::endl;

```

- 部分展开

```

1 using std::cout;
2 using std::endl;
3 std::cout << "Hello World!" << std::endl;

```

- 完全展开

```

1 using namespace std;

```

- 建议：项目中，尽量不要展开std库。可以指定空间访问+展开常用的；日常练习可以展开

缺省参数/默认参数 *Default Parameter*

缺省参数概念

缺省参数是声明或定义函数时为函数的参数指定一个默认值。在调用该函数时，若没有指定实参则采用该形参的默认值，否则使用指定的实参

- 半缺省参数必须是位于左边，且不能间隔着给

```

1 void Func(int a, int b, int c=30); // 正确
2 void Func(int a=10, int b=20, int c); // 错误，必须是在左边
3 void Func(int a, int b=20, int c); // 错误，不能间隔着给

```

- 缺省参数不能在函数声明和定义中同时出现。当分离定义时，以声明为准，因为在汇编形成符号表时以声明中的定义为准
- 缺省值必须是常量或者全局变量，不能是局部变量

```

1 // wz、def和ht的声明必须出现在函数之外
2 sz wd = 80;
3 char def = ' ';
4 sz ht();
5 string screen(sz = ht(), sz = wd, char = def);
6 string window = screen(); //调用screen(ht(), 80, ' ');
7 void f2() {
8     def = '*'; //重新赋值，改变默认实参的值
9     sz wd = 100; //局部变量构成隐藏，但没有改变默认值
10    window = screen(); //调用screen(ht(), 80, '*');
11 }

```

缺省参数分类

- 全缺省参数

```

1 void Func(int a=10, int b=20, int c=30);

```

- 半缺省参数

```

1 void Func(int a, int b=20, int c=30);

```

声明问题

在给定的namespace种一个形参只能被赋予一次默认实参，因此函数的后续声明只能为之前没有默认值的形参添加默认实参，而且该形参右侧的所有形参必须都有默认值

通常应该在函数声明中指定默认实参，并将该声明放在头文件中

函数重载 *Function Overloading*

函数重载概念

- 函数重载允许在**同一作用域**中声明几个功能类似的同名函数。函数重载很方便，就像在使用同一个函数一样
- C++的名字查找发生在类型检查之前，所以函数重载必须是在同一作用域，否则会构成同名隐藏
- 函数重载类型
 - 参数个数不同

```
1 int func(int a, double b);  
2 int func(int a, double b, char c);
```

- 参数类型不同

```
1 int func(int a, char b);  
2 int func(double a, double b);
```

- 参数类型顺序不同

```
1 int func(int a, double b);  
2 int func(double a, int b);
```

- 注意：只有返回值不同是不构成函数重载的，因为编译器是根据传递的实参类型推断想要的是哪个函数，所以若只有返回类型不同则无法识别

函数匹配

函数匹配 function matching/重载匹配 overload resolution

- 编译器找到最佳匹配 best match，并生成调用该函数的代码
- 无匹配错误 no match

- 二义性调用 ambiguous call

C++支持函数重载的原理 -- 符号修饰Name-decoration/符号改编 Name-mangling

只有声明的函数或变量在本目标文件中是没有分配虚拟地址的，只有在定义之后才会分配内存地址。在编译器的链接过程中，会去找总符号表（同名的cpp文件和其头文件生成一份符号表），里面记录着不同函数的地址。只有声明的函数可以通过定义文件的符号表找到自己的地址，若没有找到就会报链接错误（编译阶段只报语法错误）

在Linux中利用 `objdump -S` 指令，可以发现在Linux的gcc编译器中，C语言编译器直接用函数名作为其符号表的命名，比如 `<Func>`；而C++编译器则会进行函数名修饰，比如分别为 `<_Z4Funcid>` 和 `<_Z4Funcii>`。修饰规则比较复杂，每种编译器在不同系统的修饰规则也不同。关于GCC的基本C++修饰规则可以看自我修养P88

binutils工具包里面了一个 `c++filt` 工具来解析被修饰过的名称

```
1 $ c++filt _Z4Funcid
2 Func(int, double)
```

符号修饰规则不仅用于函数重载，对于全局（静态）变量和（局部）静态变量也要进行符号修饰防止冲突

因为不同的编译器使用的符号修饰规则是不同的，所以不同的编译器编译产生的ELF文件是无法通过链接器连接到一块的，因为在符号表里找不到需要的符号，这是导致不同编译器之间不能互操作的主要原因之一

C和C++互相调用库 `extern "C"`

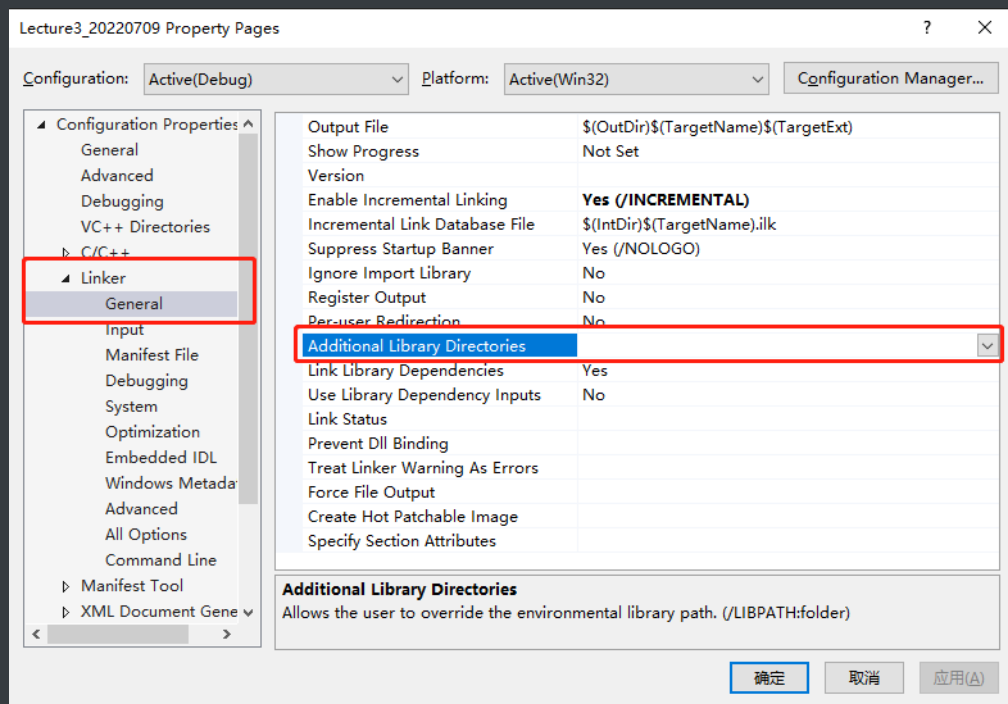
- 生成静态库.lib和动态库.dll
- C++调用C库
 - C++中在调用C的头文件时使用 `extern "C"`：告诉C++的编译器，这里面的函数使用C的库实现的，用C的规则去链接查找它们

```

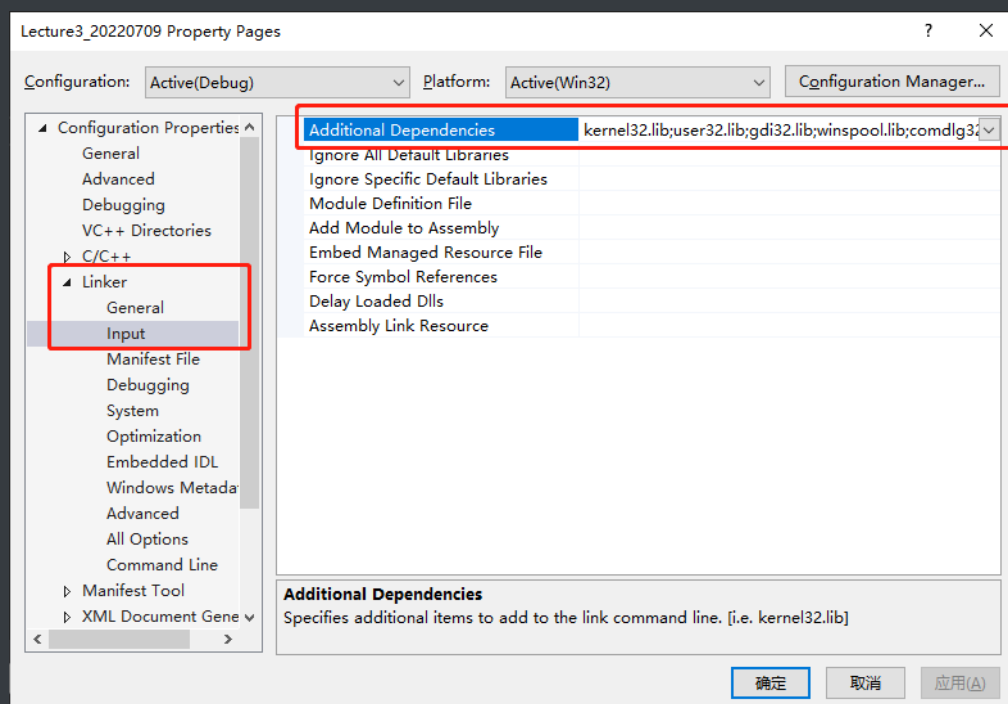
1 extern "C" {
2     #include "....h"
3     // ...
4 }
5 extern "C" int var; //单独声明某个符号为C语言符号

```

■ 附加库目录



■ 附加依赖项



一种更通用的方式是在系统头文件中添加 `__cplusplus` 条件编译，这样可以让C++能调用C，也能让C调用C++。比如说C语言共享库 `string.h` 中的 `memset` 函数

```
1  #ifdef __cplusplus
2  extern "C" {
3  #endif
4  void *memset(void *, int, size_t);
5  #ifdef __cplusplus
6  }
7  #endif
```

引用 Reference

注意：这部分特指左值引用

引用的概念

引用不是新定义一个变量，而是给已存在变量取了一个别名，编译器不会为引用变量开辟内存空间，它和它引用的变量共用同一块内存空间

& 操作符：和类型在一起的是引用，和变量在一起的是取地址

引用特性

- 引用在定义时必须初始化
- 一个变量可以有多个引用
- 引用一旦已经引用了一个实体后，不能再引用其他实体
- 除了const和类以外，其他的引用必须严格类型匹配

```

1  int a = 1;
2  int& b = a; // 必须初始化
3
4  int& c = a; // 同一个变量可以有多个引用
5
6  int x = 10;
7  b = x; // 将会同时赋值给b和a，因为b和a是同一个地址

```

使用场景

■ 函数传参

函数传参本质上和变量的初始化过程是一样的，都是把一个右值传给左值

- 传值传参 passed by value 需要拷贝。C中常用指针来传递参数，从而使用输出的目的，但是比如在链表中要改变指针自身的时候要传二级指针就很麻烦
- 引用传参 passed by reference/传引用调用 called by reference
 - 输出型参数，突破函数只能返回一个值的限制

```

1  typedef struct SeqList {
2      //...
3  }SL;
4  void SLPushBack(SL& s, int x) {
5      //...
6  }
7  int main() {
8      SL sl;
9      SLPushBack(sl); // 不用传指针了
10 }

```

- 大对象传参，减少拷贝，提高效率
- 做返回值
 - 传值返回（见C语言中的函数栈帧），小对象放寄存器，大对象放上层栈帧

做法是将return的值装载到一个寄存器中，并将寄存器中保存的值给原函数中的接收变量。如果是将临时变量z设置为静态变量z，即 `static int z`。那么z会被保存到静态区中，并不会被销毁。但编译器仍然会选择将z的值加入到寄存器中生成临时拷贝后返回给上层

- 传引用返回的问题

- `ret`的结果是未定义的，栈帧调用结束时，系统会清理栈帧并置成随机值，那么这里`ret`的结果就是随机值。因此该程序使用引用返回本质是不对的，越界后结果没有保证。因此传引用返回的前提是出了函数作用域，返回对象就销毁了，那么一定不能用引用返回，一定要用传值返回。

- 修改方式

- 将Count中的int放到静态区中，这样Count调用结束，栈帧销毁后，静态区中的int也不会被销毁
- `malloc`出来的内存是在堆上也不会被销毁，因此传引用返回可以应用到顺序表等数据结构中提高效率

```
1  // 错误
2  int& Count() {
3      int n = 0;
4      n++;
5      // ...
6      return n;
7  }
8  // 放到静态区中传引用返回
9  int& Count() {
10     static int n = 0;
11     n++;
12     // ...
13     return n;
14 }
15 int main() {
16     int ret = Count();
17     return 0;
18 }
```

- 传引用返回的优势
 - 输出型返回对象，调用者可以修改返回对象，比如 `operator[]`
 - 减少拷贝

例子：传引用与传值在类中的应用

```
1 // 实例
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5 class A {
6 public:
7     A(int a = 0) {
8         _a = a;
9         cout << "A(int a = 0)->" << _a << endl;
10    }
11    // A aa2(aa1);
12    A(const A& aa) {
13        _a = aa._a;
14        cout << "A(const A& aa)->" << _a << endl;
15    }
16    ~A() {
17        cout << "~A()->" << _a << endl;
18    }
19 private:
20     int _a;
21 };
22
23 void func1(A aa) {} //传值传参
24 void func2(A& aa) {} //传引用传参
25
26 A func3() {
27     static A aa(3);
28     return aa;
29 }
```

```

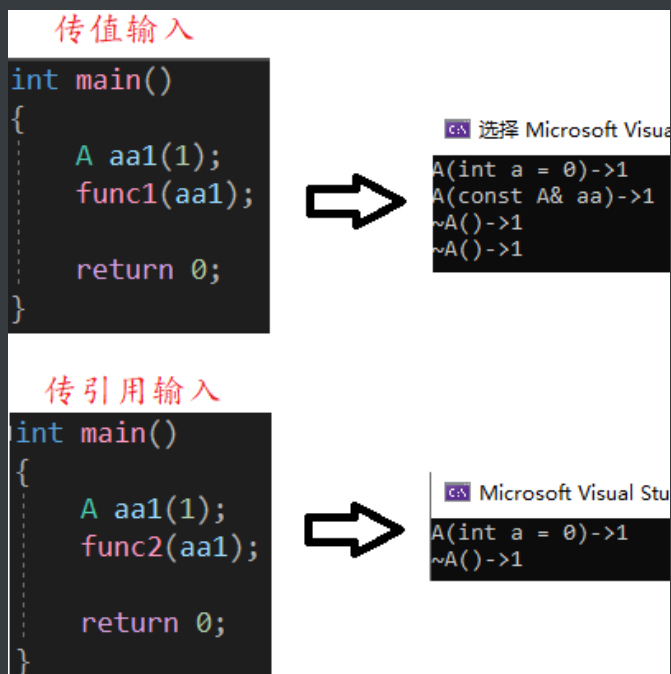
30 A& func4() {
31     static A aa(4);
32     return aa;
33 }

```

- 中间临时量 temporary object 问题，具体看《程序员的自我修养--链接、装载与库》
10.2.3函数返回值传递机制

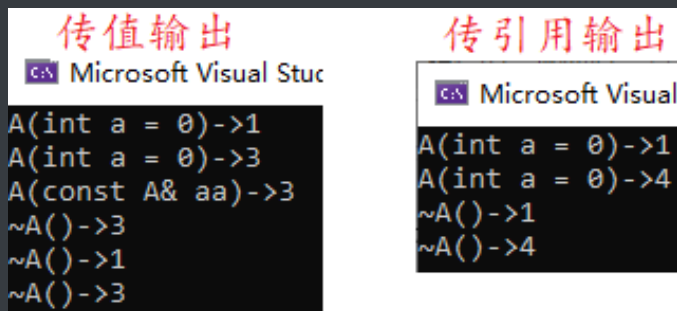
- 函数的返回对于1-8字节的小对象，直接通过eax寄存器存放的临时量返回。注意这个临时对象编译器默认将其设置为const
- 对于大于8字节的大对象，会在上层栈帧开辟temp空间进行返回

- 输入



- 传值输入：需要调用拷贝函数，压栈时创建临时的形参，将实参拷贝给形参后调用形参，调用拷贝构造函数有栈帧开销和拷贝临时量的空间浪费，从上图试验中可以看出，传值需要多调用拷贝和析构，开销很大
- 传引用输入：不需要调用拷贝构造函数，直接传递引用
- 结论：传引用和传值对于内置类型效果不明显，都需要4个或8个字节。但对自定义类型效果明显，传引用不需要调用拷贝构造进行拷贝

- 输出



- 传值输出：需要调用拷贝构造函数拷贝生成一个临时量，然后将临时量的值返回给外层函数的接收变量，栈帧销毁后一块销毁，即传值返回不会直接返回原来栈帧中的对象，而是返回对象的拷贝。调用拷贝构造函数有栈帧开销和拷贝临时量的空间浪费
- 传引用输出：不需要调用拷贝构造函数，直接传递引用
- 结论
 - 调用拷贝构造函数有栈帧开销
 - 自定义类型有时需要深拷贝
 - 自定义类型往往比较大

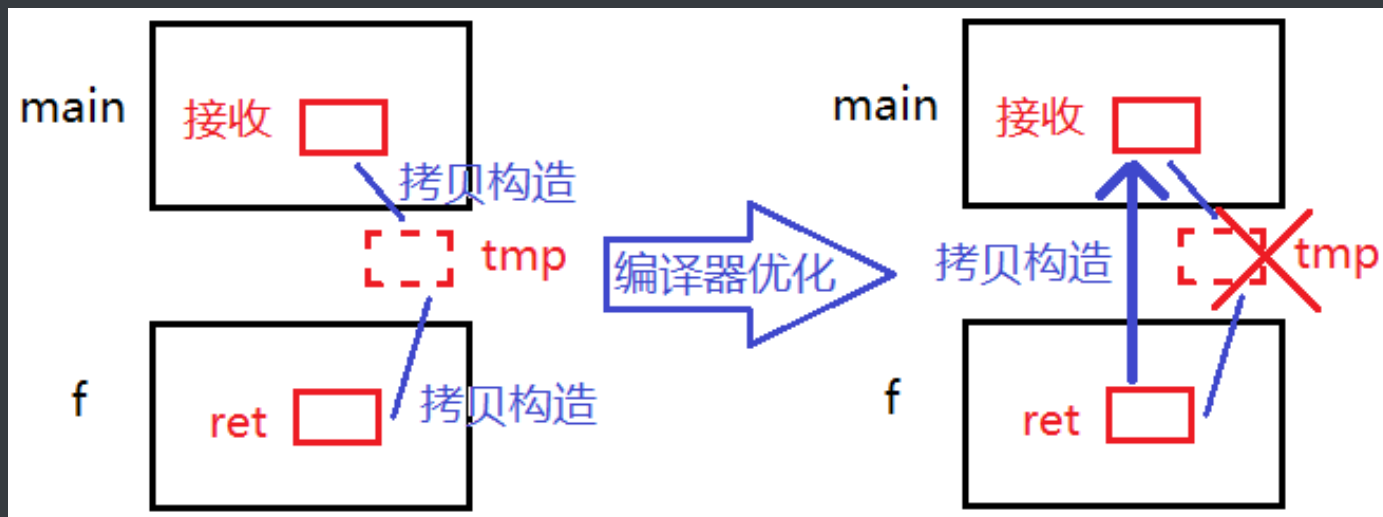
编译器对传值传参和传引用传参的优化

这部分在C++2022.08.05的课程有讲

编译器在release下的优化是取消了中间临时变量，直接将要返回的值拷贝给接收变量，Linux可以通过 `-fno-elide-constructors` 关闭编译器的所有优化

但是不能因此而取消中间临时变量，因为根据建立栈帧的顺序，ret必须要紧跟着func栈帧才能被找到，并且进行优化

```
1 int ret1 = func(x); //只有直接用变量来接收编译器才会进行优化
2 int ret2;
3 // ...
4 ret2 = func(y); //这时候就不会优化的
```

指针与引用的关系

- 指针和引用用途基本是相似的，但指针更强大也更危险
- 使用场景
 - 链表定义中不能使用引用，因为引用必须要初始化赋值且引用不能更改指向
- 语法特性及底层原理
 - 语法角度来看引用没有开空间，指针开了4或8字节空间
 - 底层原理来看，引用底层是用指针实现的

内联函数 *Inline*

- 定义：以 `inline` 修饰的函数叫做内联函数，编译时C++编译器会在调用内联函数的地方展开，没有函数调用建立栈帧的开销，而是直接替换成了一些机器代码，因此内联函数提升程序运行的效率
- 使用场景：堆排序和快速排序中需要被频繁调用的Swap函数

C语言中用宏函数来避免建立和销毁栈帧

- 宏的优点：复用性变强、宏函数提高效率，减少栈帧开销、不用进行传入参数的类型检查（用enum）
- 宏的缺点：可读性差，复杂不易编写、没有类型安全检查、不方便调试（预处理阶段就被替换掉了）

```
1 #define ADD(a, b) ((a) + (b))
2 // 记住两个特殊场景
3 ADD(1, 2) * 3;
4 ADD(a | b, a&b);
```

inline特性

- inline是一种以空间换时间的做法，若编译器将函数当成内联函数处理，在编译阶段，会用函数体替代函数调用。缺陷是可能会使文件目标变大；优势是少了调用开销，提高程序运行效率
- inline对于编译器而言只是一个建议，不同编译器关于inline实现机制可能不同
- inline不建议声明和定义分离，分离会导致链接错误。inline不会被放进编译生成的符号表里，因为inline定义的函数符号是不会被调用的，只会被展开使用。推荐声明+定义全部写在头文件中（这也符合模块化设计声明和定义分离的设计初衷，便于使用）

C++的强制类型转换

<https://learn.microsoft.com/zh-cn/cpp/cpp/casting-operators?view=msvc-170>

为什么C++需要四种类型转换

C++继承了C语言的隐式类型转换和显式类型转换体系，可以看C.md类型转换部分

这里有一个经典的错误可以看Cpp.md的string类模拟实现部分的insert部分：[经典隐式类型转换错误](#)

C++兼容了C隐式类型转换和强制类型转换，但希望用户不要再使用C语言中的强制类型转换了，希望使用规范的C++显式强制类型转换

强制类型转换的形式为 `cast_name<type>(expression);`

static_cast 用于非多态类型转换

<https://learn.microsoft.com/zh-cn/cpp/cpp/static-cast-operator?view=msvc-170>

```
1 double d = 12.34;
2 int a = static_cast<int>(d);
3
4 void *p = &d;
5 double *dp = static_cast<double*>(p); //规范使用void*转换
```

任何具有明确定义的类型转换，只要不包含底层const，都可以使用 `static_cast`

`static_cast` 是一种**相对安全**的类型转换运算符，它可以将一种类型转换为另一种类型。`static_cast` 可以执行隐式类型转换，例如将整数类型转换为浮点类型，也可以执行显式类型转换，例如将指针类型转换为整数类型。`static_cast` 进行类型转换时会执行一些类型检查和转换，以确保类型转换是合法的

举两个内存池项目中的例子

1. 有 `T *obj = nullptr;` 和 `char *_memory = nullptr;`，`obj = (T *)_memory` 可以通过编译，而 `obj = static_cast<T *>(_memory)` 会报错：`invalid 'static_cast' from type 'char' to type 'TreeNode'`
2. `void *next = *(static_cast<void **>(_freeList))` 从 `*` 转换成 `**` 是可以的

`reinterpret_cast` 用于对位进行简单的重新解释

C++类型转换之`reinterpret_cast` - 叫啥名呢的文章 - 知乎 <https://zhuanlan.zhihu.com/p/33040213>

`reinterpret_cast` 可以将一个指针或引用类型强制转换为另一个指针或引用类型，而**不进行任何类型检查或转换**。`reinterpret_cast` 主要用于以下情况：

1. 将指针或引用类型转换为另一个指针或引用类型，以便可以在它们之间进行转换
2. 在某些特殊情况下，当需要使用指针或引用类型表示不同的对象类型时，可以使用 `reinterpret_cast` 进行类型转换。这通常发生在涉及底层硬件或操作系统接口的代码中
3. 在某些情况下，`reinterpret_cast` 也可以用于类型擦除，即将模板类型擦除为一个没有模板参数的类型，以便可以在运行时处理它们

`reinterpret_cast` 的危险性 ⚠️：需要注意的是，使用 `reinterpret_cast` 进行类型转换时必须非常小心。由于它不执行任何类型检查或转换，如果类型转换不正确，可能会导致未定义的行为或错误的结果。因此，应该尽可能避免使用 `reinterpret_cast`，而优先考虑使用其他更安全的类型转换运算符，例如 `static_cast` 或 `dynamic_cast`

`const_cast`

通过 `const_cast` 去除`const`属性 `cast away the const`，但是在去除`const`属性后再进行写就行未定义的行为了

```
1  const char *pc;
2  char *p = const_cast<char*>(pc); //pc现在是非常量了
3  *pc = 2;
```

`const_cast` 常用于有函数重载的上下文中

`dynamic_cast` 用于多态类型的转换

`dynamic_cast` 和前面的三类用于增强C++规范的类型转换不同，它专用于区分父类指针是指向子类还是指向父类，即用于将一个父类对象的指针/引用转换为子类对象的指针或引用，即动态转换

父类对象是无论如何都不允许转子类的，但允许父类的指针或引用转换为子类对象的指针或引用

- 向上转型：子类对象指针或引用指向父类对象或引用，也就是切片，不需要进行转换

```
1  class A {};
```

```
2  class B : public A {};
```

```
3  B bb;
```

```
4  A aa1 = bb;
```

```
5  A& ra1 = bb; //向上转换切片，所以没有产生中间变量，也就不需要const了
```

- 向下转型：父类对象指针或引用指向子类对象或引用
 - 父类指针强制转子类有越界的风险，因为指针相当于规定了能看到的内存空间为多大，而父类的大小小于等于子类

- 只能用于父类含有虚函数的多态类
- 这时候只有使用 `dynamic_cast` 才是安全的。安全是什么意思？
 - 如果指针是指向子类，那么可以转换是安全的
 - 如果指针是指向父类，那么不能转换，转换表达式返回 `nullptr`

RTTI思想

C++是一种静态类型 `statically typed` 语言，其含义是在编译阶段进行类型检查 `type checking`。对象的类型决定了对象所能参与的运算，因此类型检查的含义是编译器负责检查类型是否支持要执行的运算。当然前提是编译器必须要知道每一个实体对象的类型，所以这要求在使用变量前必须要声明其类型

Run-time Type Identification 运行时类型识别

C++通过以下方式来支持RTTI

- `typeid` 运算符：获取对象类型字符串
- `dynamic_cast` 运算符：父类的指针是指向父类对象还是子类对象
- `decltype` 运算符：推导一个对象类型，这个类型可以用来定义另一个对象

C++的IO流

C语言的IO函数

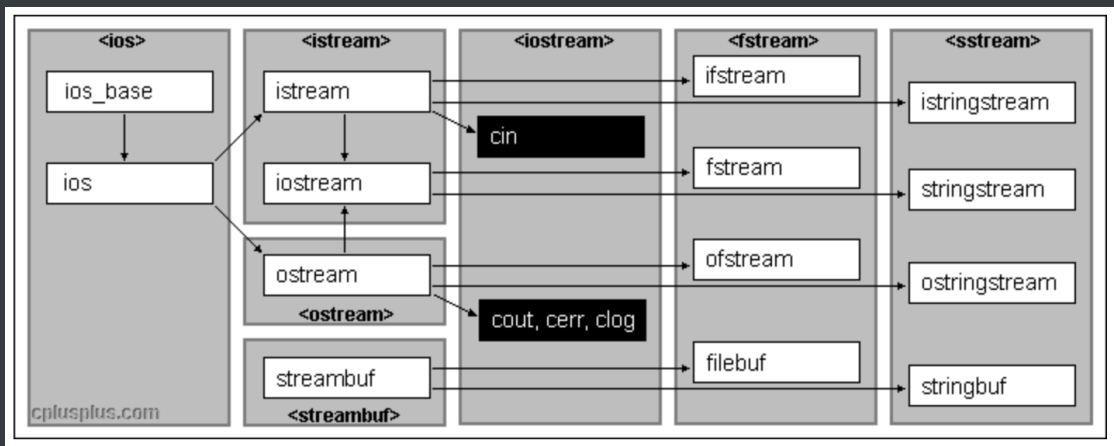
- `scanf()/printf()` 处理终端/控制台IO
- `fscanf()/fprintf()` 处理文件IO
- `sscanf()/sprintf()` 处理字符串IO

C语言是面向对象的，只能指针内置类型

C++IO流设计

什么是流 `Stream`？流是对有序连续且具有方向性的数据的抽象描述。C++流是指信息从外设向计算机内存或从内存流出到外设的过程

为了实现这种流动，C++实现了如下的标准IO库继承体系，其中 `ios` 为基类，其他类都是直接或间接派生于它



- `<istream>` 和 `<iostream>` 类处理控制台终端
- `<fstream>` 类处理文件
- `<sstream>` 类处理字符串缓冲区
- C++暂时还不支持网络流的IO库

C++标准IO流

C++标准库提供了4个全局流对象 `cin`，`cout`，`cerr`，`clog`。从下图可以看到 `std::cin` 是一个 `istream` 类的全局流对象

```
object
std::cin
extern istream cin;
```

C++对这部分的设计并不是特别好，因此这几个对象在输出上基本没有区别，只是应用该场景略有不同

使用 `cin` 进行标准输入即数据通过键盘输入到程序中；使用 `cout` 进行标准输出，即数据从内存流向控制台(显示器)；使用 `cerr` 用来进行标准错误的输出；使用 `clog` 进行日志的输出

- cin 为缓冲流，相当于是一个 char buff[N] 对象，键盘输入的数据保存在缓冲区中，当要提取时，是从缓冲区中
- 空格和回车都可以作为数据之间的分隔符，所以多个数据可以在一行输入，也可以分行输入。但若是字符型和字符串，则空格（ASCII码为32）无法用 cin 输入。字符串中也不能有空哦个，回车符也无法读入

```

1 // 2022 11 28
2 // 输入多个值，默认都是用空格或者换行分割
3 int year, month, day;
4 cin >> year >> month >> day;
5 scanf("%d%d%d", &year, &month, &day);
6 scanf("%d %d %d", &year, &month, &day); //不需要去加空格
7
8 //20221128 无空格
9 scanf("%4d%2d%2d", &year, &month, &day); //scanf可以通过加宽度直接处理
10 //cin在这种情况下反而会比较麻烦
11 string str;
12 cin >> str;
13 year = stoi(str.substr(0, 4));
14 month = stoi(str.substr(4, 2));
15 day = stoi(str.substr(6, 2));

```

- cin, cout 可以直接输入和输出内置类型数据，因为标准库已经将所有内置类型的输入和输出全部重载了

std::ostream::operator<<

<ostream> <iostream>

C++98 C++11

```

ostream& operator<< (bool val);
ostream& operator<< (short val);
ostream& operator<< (unsigned short val);
ostream& operator<< (int val);
ostream& operator<< (unsigned int val);
ostream& operator<< (long val);
arithmetic types (1) ostream& operator<< (unsigned long val);
ostream& operator<< (long long val);
ostream& operator<< (unsigned long long val);
ostream& operator<< (float val);
ostream& operator<< (double val);
ostream& operator<< (long double val);
ostream& operator<< (void* val);
stream buffers (2) ostream& operator<< (streambuf* sb );
ostream& operator<< (ostream& (*pf)(ostream&));
manipulators (3) ostream& operator<< (ios& (*pf)(ios&));
ostream& operator<< (ios_base& (*pf)(ios_base&));

```

- 对于自定义类型，可以重载运算符 << 和 >>

```
function
```

```
std::operator>> (string)
```

```
istream& operator>> (istream& is, string& str);
```

如上图是对string类的 >> 运算符重载，意思是支持了 istream& is 和 string& str 之间的 >> 运算

这部分可以参考Date类中的 << 重载

■ OJ中的输入与输出

```
1 // 单个元素循环输入
2 while(cin>>a) {
3     // ...
4 }
5 // 多个元素循环输入
6 while(c>>a>>b>>c) {
7     // ...
8 }
9 // 整行接收
10 while(cin>>str) {
11     // ...
12 }
```

OJ中有时可能会出现用 cin 和 cout 效率过不了的问题，可以考虑采用 printf 和 scanf 。这是因为C++为了要兼容C语言，内部需要采取和C语言输入输出函数同步顺序的一些处理，因此可能会导致效率的下降

因此还是建议在OJ中用 printf 和 scanf

istream 类型对象转换为逻辑条件判断值

考虑有多组测试用例的情况，比如上面的代码中输入多组日期，那么要用


```
1 char buff[128];
2 while (scanf("%s", buff) != EOF) {}
3 // or
4 while (cin >> str) {} //如何实现逻辑判断?
```

对于 `scanf()` 而言，这是一个函数调用，返回的是接收到的值的数目，因此很好理解，若写入失败（满了）或按 `ctrl+z`，就返回EOF，ASCII码中 `EOF==0`，因此while循环会被终止，可是 `cin` 是一个全局流对象，为什么它也等价于一个逻辑值，从而可以作为while循环的判断条件呢？

内置类型是可以隐式类型转换成自定义类型的，编译器会自动调构造和拷贝构造（有些编译器自动优化为只有自定义），反过来自定义类型也可以转换为内置类型，但是此时需要显式重载 `operator TYPE()`，比如重载 `operator bool()` 将自定义类型转换成bool内置类型

`cin >> str` 的返回值是 `istream` 对象，因此在`istream`类内部实现了 `operator bool()`，因此当 `while (istream& obj)` 可以进行逻辑判断

`operator bool()` 全部是复用的ios基类的实现

C++文件IO流

C++用文件IO流的使用步骤

1. 定义一个文件流对象
 - `std::ifstream` 只输入用
 - `std::ofstream` 只输出
 - `std::fstream` 既输入又输出
2. 使用文件流对象的成员函数打开一个磁盘文件，使得文件流对象和磁盘文件之间建立联系
3. 使用提取和插入运算符对文件进行读写操作，或使用成员函数进行读写
4. 关闭文件 `close()`

打开模式，注意**C++**根据文件内容的数据格式分为二进制文件和文本文件

- `std::ios::in` 读

- `std::ios::out` 写
- `std::ios::app` 追加
- `std::ios::binary` 二进制流
- `std::ios::ate` 输出在末尾
- `std::ios::trunc` 截断

一个 `std::fstream` 的demo

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6      std::fstream file("data.txt", std::ios::in | std::ios::out |
7                          std::ios::app); // 打开文件
8
9      if (file.is_open()) { // 确保文件成功打开
10         file << "Hello, World!" << std::endl; // 写入数据到文件
11         file.seekg(0); // 将文件指针移动到文件开头
12
13         std::string line;
14         while (std::getline(file, line)) { // 读取文件内容
15             std::cout << line << std::endl; // 输出每行内容
16         }
17         file.close(); // 关闭文件
18     } else {
19         std::cout << "Failed to open the file." << std::endl;
20     }
21
22     return 0;
23 }
```

stringstream

istringstream、ostringstream 和 stringstream，分别用来进行流的输入、输出和输入输出操作

类与对象（上） -- 类成员

类的定义

```
1 // .h 文件中的声明
2 class className { // 类名
3     // class默认成员访问权限为private
4     int attribute = 1; // 类属性和方法用 "." 操作符调用，即
    className.attribute
5     void Method_declaration(); // 定义为类的一部分的函数称为成员函数 member
    fucntion 或 类方法 method
6 }
7 // .cpp 文件中的类方法定义
8 void className::Method_definition() { /*...*/ }
```

类的定义原则

- 类函数的声明必须要在类内部，而类函数的声明则既可以在类内部也可以在类外部
- 小函数若想成为inline，直接在类里面定义即可。但具体是否会作为inline处理取决于编译器
- 若是大函数，应该声明和定义分离。一般情况下都使用这种定义方式

类的作用域

变量搜索遵循局部优先，因此在c++文件中定义类函数时要用 `::` 指定类的作用域

类里面用 `typedef` 或者 `using` 起别名，但要注意用来定义类型的成员必须先定义后使用

类的命名规范

- 单词和单词之间用驼峰法
- 函数名、类名等所有单词首字母大写 `DataMgr`
- 变量首字母小写，后面单词首字母大写 `dataMgr`
- 类的属性变量在开头或结尾使用 `_`（STL库）或者驼峰（公司规范居多） `_dataMgr`

```
1 class Date {
2 public:
3     void Init(int year)
4         _year = year;
5         // 若属性不写成_year, 则可能会出现 year = year 的混淆
6 private:
7     int _year;
8 }
```

类的访问限定符及封装

访问限定符 Access Modifier

```
1 class A {
2 public:
3     void PrintA()
4         cout << _a << endl;
5 private:
6     char _a;
7 };
```

- 分类
 - public 公有
 - protected 保护
 - private 私有
- 访问限定符的说明

- `public`修饰的成员在类外可以直接访问
- `protected`和`private`修饰的成员在类外不能直接被访问
- 访问权限作用域从该访问限定符出现的未知开始直到下一个访问限定符出现时为止
- 若后面没有访问限定符，作用域就到 `}` 为止
- 三者都是只对类作用域外有效，即类内无论是什么限定符都可以互相取
- `struct`定义的类默认访问权限是`public`，而`class`定义的类默认访问权限则是`private`
- C++中`struct`和`class`的区别：C++需要兼容C语言，所以C++中`struct`可以当成结构体使用。但C++中的`struct`既可以定义变量，也可以定义函数。因此C++中将`struct`升级成了`class`。区别是`struct`定义的类默认访问权限是`public`，而`class`定义的类默认访问权限则是`private`。

封装 Encapsulation

隐藏对象的属性和实现细节，即隐藏成员变量。仅对外公开接口来和对象进行交互（即开放成员函数接口）

```
1  cout << st.a[st.top] << endl; // 这样调栈顶数据是错误的，因为类属性受保护。
2  // 而且很有可能出错，因为使用者并不知道底层的实现，top到底指向哪一个数据？
3  cout << st.Top() << endl; // 这样是正确的，要使用给的函数接口
```

C语言没办法封装，可以规范的使用函数访问数据，也可以不规范的直接访问数据；C++使用封装，必须规范使用函数访问数据，不能直接访问数据

注意下面一个例子，在类内声明，在类外定义也是可以无视限定符的

```
1 class A {
2 public:
3     void Print(); //类成员声明
4 private:
5     int _a;
6 };
7
8 void A::Print() {
9     cout << _a << endl;
10 } //类成员定义
```

类的实例化 *Instantiation*

- 类的声明不会占用物理空间
- 一个类可以实例化出多个对象，实例化出的对象占用实际的物理空间，存储类成员变量

类对象模型

类对象的可能存储方式

- ☐ 对象中包含类的各个成员：问题是每个对象中成员变量都需要调用同一份函数，导致大量的冗余代码
- ☐ 实例化的每个对象成员变量都是独立空间，是不同变量。但是每个对象调用的成员函数都是同一个。没有被采用，在虚表和多态部分采用
- ☒ 公共代码区
 - 只保存非静态的成员变量，成员函数存放在公共的代码段
 - 编译链接时就根据函数名去公共代码区找到函数的地址，然后call函数地址
 - 下面代码可以顺利运行，说明没有对空指针解引用，所以实际使用的是公共代码区

```

1  class A {
2  public:
3      void func() {
4          cout << "void A::func()" << endl;
5      }
6  };
7
8  int main() {
9      A* ptr = nullptr;
10     ptr->func();
11     return 0;
12 }

```

类的内存对齐规则

- 属性和C语言中自定义类型的对齐规则一样
- 方法属于公共区代码，所以不计入类的大小。但是即使属性和方法都为空，也会为其分配1字节占位。不存储实际数据，标识对象存在

*this*指针

this指针的引出

```

1  class Date {
2  public:
3      void Init(int year, int month, int day) {
4          _year = year;
5          _month = month;
6          _day = day;
7      }
8      void Print() {
9          cout << _year << "-" << _month << "-" << _day << endl;
10     }
11 private:
12     int _year;

```

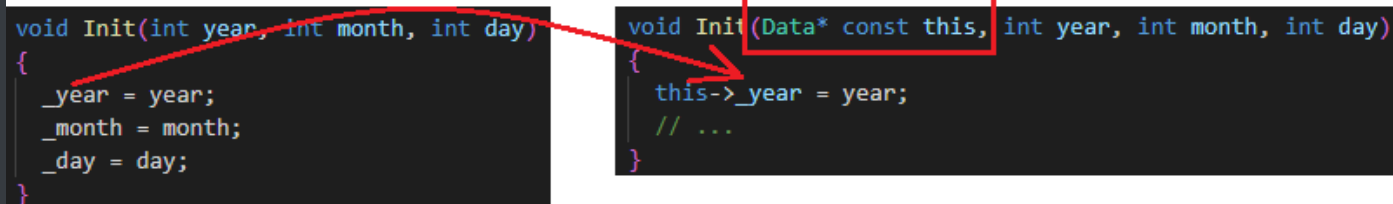
```
13     int _month;
14     int _day;
15     int a;
16 }
```

```
1 Date d1;
2 d1.Init(2022, 7, 17);
3 Date d2;
4 d2.Init(2022, 7, 18);
5 d1.Print();
6 d2.Print();
```

产生两个对象d1和d2，但是方法中并没有区分作用的对象是哪一个，函数是怎么知道该设置哪一个对象呢？

类中的方法会在第一个形参处放置一个额外的隐式this指针用来访问调用它的那个对象，当然下图中右边的类方法是不对的，因为不允许显式地给出this指针，这里只是为了说明用

编译器负责把调用者的地址传给this



```
void Init(int year, int month, int day)
{
    _year = year;
    _month = month;
    _day = day;
}

void Init(Data* const this, int year, int month, int day)
{
    this->_year = year;
    // ...
}
```

this指针的特性

- this指针是所有**非静态**成员函数的第一个隐形形参
- this指针是一指针常量：语法规则不能修改this指针（`T* const this`），因为this指针是被const保护的只读，但可以修改this指针指向的内容
- 也不能在实参和形参位置显式地传递this指针。但可以在类函数的内部使用（不使用也会自动加）

- this指针本质上是成员函数的形参，当对象调用该成员函数时，将对象地址作为实参传递给this形参，所以对象中不存储this指针。VS下将this指针优化到了寄存器中，因为在频繁使用时可以提高效率
- 问题：this指针可以为空吗？可以！参考下面两段对比代码

```
1  class A {
2  public:
3      void Print()
4          cout << "Print()" << endl;
5  private:
6      int _a;
7  };
8
9  int main {
10     A* p = nullptr;
11     p->Print();
12     return 0;
13 }
```

不会崩溃，虽然p是空指针，但并没对它解引用，因此程序正常运行

```
1  class B {
2  public:
3      void Print()
4          // 相当于是 cout << this->_b << endl;
5          cout << _b << endl;
6  private:
7      int _b;
8  };
9
10 int main {
11     B* p = nullptr;
12     p->Print();
13     return 0;
14 }
```

程序会崩溃，因为对空指针解引用了！

const成员函数

介绍

this指针是一个 `T* const this` 的指针常量，我们不能修改this指针本身，也就说不能莫名其妙把调用者改成其他人。但是仍然可以通过this指针来修改调用者的属性，比如通过 `this->_year` 就可以获取 `_year` 属性了

那么如果我们想保持类属性不变怎么办呢，一个方法当然是可以直接把类属性设置为const，但是这样任何其他的方法也就不能修改属性了，不够灵活。可以通过将某个特定方法设置为const，即const成员函数 `const member function` 来达成，此时的this指针就相当于变成了 `const T* const this`

权限放大问题

```
1  class Date {
2  Public:
3      // ...
4      void Date::Print() {
5          cout << _year << "/" << _month << "/" << _day << endl;
6      }
7      bool Date::operator<(const Date& d) {
8          return !(*this >= d);
9      }
10 }
11 int main() {
12     Date d1(2022, 7, 25);
13     const Date d2(2022, 7, 26);
14
15     d1.Print();
16     // d2.Print(); // 编译报错，权限放大
17     d1 < d2;
18     // d2 < d1; // 编译报错，权限放大
19 }
```

- `Date* const this` `this`指针是一个指针常量，指针本身不能被修改，但可以修改指针指向的内容。当传入一个`const Date d2` 常量时，权限就被放大了
- `d2 < d1`，因为 `<` 运算符重载时，第一个参数是`this`指针常量，和上面的错误是一样的，都是权限放大
- 若要支持以上的调用，我们必须将`this`指针设置为常量指针。`void Date::Print()` `const` 修饰的`this`指向的内容，也就是保证了成员函数内部不会修改成员变量`const`对象和非`const`对象都可以调用这个成员函数（权限平移）

```
1 void Date::Print() const {
2     cout << _year << "/" << _month << "/" << _day << endl;
3 }
4 bool Date::operator<(const Date& d) const {
5     return !(*this >= d);
6 }
```

static成员

概念

有些时候类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持关联。比方说银行账户类会规定基准利率，这个基准利率是对每个账户对象一致成立的，所以它属于这个类的共性，因此没有必要让每个类对象都存储它。当修改它的时候应该要对所有的类对象都成立

静态成员就是专门给这个类访问，**静态成员变量一定要在类外进行初始化**，注意在类外定义的时候必须要用 `::` 指定类域，否则编译器找不到

```
1 class A {
2     public:
3         A() { ++_scount; }
4         A(const A& t) { ++_scount; }
5         ~A() { --_scount; }
6         static int GetACount() { return _scount; } // 静态成员函数，没有this指针，不能访问非静态成员
7     private:
8         static int _scount; // 静态成员变量声明
```

```

9  };
10
11  // 在类外面定义初始化静态成员变量
12  int A::_scount = 0;
13
14  int main() {
15      A a1;
16      A a2;
17      cout << a1.GetACount() << endl; // 用静态成员函数来取
18  }

```

特性

- 静态成员为所有类对象所共享，属于整个类，也属于这个类的所有对象，并不单独属于某个具体的对象，这点在 `std::shared_ptr` 的设计中是一个坑，存放在静态区
- 和友元一样，静态成员变量**必须在类外定义和初始化**（但是静态成员函数可以在类内定义），定义时不添加static关键字，类中只是声明，如 `int A::scount = 0;`，若是模板的话要把模板也给带上。初始化列表不能用来初始化静态成员
- 静态成员变量也受访问限定符的限制
- 当static是公有的时候，类静态成员可用 `类名::静态成员` 或者 `对象.静态成员` 来访问。但是若私有，则只能通过 `类名::静态成员` 访问或静态成员函数来获取，这是因为类静态成员不属于某个类对象，而是存放于静态区中，属于类域
- **静态成员函数没有隐藏的this指针，不能访问任何非静态成员**，也不能声明const成员函数，也不能在static函数体内使用this指针
- 静态变量不能给缺省值，只能在类外面给初始值，但 `const int` 类型的静态变量可以给缺省值，比如哈希桶中的素数size扩容就用到了这个特性

一个应用：设计一个只能在栈上定义对象的类

若把构造函数定义为public，那么类对象可以定义在任何地方。但若把构造函数定义为private，然后给一个静态函数，里面定义类后返回

静态成员函数没有this指针，不需要访问对象来调用。若不用static，就会产生调用成员函数需要现创建对象的矛盾，因为调用成员函数需要this指针指向对象

```
1  class StackOnly{
2  public:
3      static StackOnly CreateObj(){
4          StackOnly so;
5          return so;
6      }
7  private:
8      StackOnly(int x = 0, int y = 0)
9          :_x(x)
10         , _y(0)
11     {}
12     int _x = 0;
13     int _y = 0;
14 };
```

调用问题

- 静态成员函数可以调用非静态成员函数吗？不能，因为静态成员函数没有this指针
- 非静态函数可以调用类的静态成员函数吗？可以，因为静态成员属于整个类和类的所有对象

友元 *Friend*

有一些函数虽然不属于类，但类也要用到他们并且允许他们访问类的非公有成员从而实现某些功能。友元提供了一种突破封装的方式，在一些使用场景下提供了便利。但是友元会增加耦合度，破坏封装，所以要尽可能少地使用友元

友元函数

友元函数可以直接访问类的私有成员，它是定义在类外部的普通函数，不属于任何类，但需要在类的内部声明，声明时需要加 `friend` 关键字

注意：友元在类内部的声明不是普通的声明，相当于只是制定了访问的权限，还需要在头文件的**类外部再次声明**然后在c文件中定义

- 运用场景

- 重载运算符： `operator<<` 、 `operator>>` 当定义为类成员函数时，`cout`的输出流对象在和隐含的`this`指针抢占第一个参数的位置。为了保证第一个形参为`cout`，需要将该运算符重载定义为全局函数，但此时又会导致它是类外成员而无法取得类属性成员，因此要借助友元函数帮助
- 多个类之间共享数据，如有一个函数 `void func(const A& a, const B& b, const C& c)`，设置`func`同时成为A、B、C类的友元函数以获取多个类之中的数据
- 说明
 - 友元函数可以访问类的所有成员，包括私有保护成员，但不是类的成员函数，所有它没有`this`指针
 - 友元函数不能用 `const` 修饰，因为在类中 `const` 修饰的是`this`指针
 - 友元函数可以在类定义的任何地方声明，不受类访问限定符限制
 - 一个函数可以是多个类的友元函数
 - 友元函数的调用与普通函数的调用原理相同

友元类

- 友元类的所有成员函数都可以是另一个类的成员函数，都可以访问另一个类中的非公有成员
- 特性
 - 友元关系是单向的，不具有交换性
 - 友元关系不能传递
 - 友元关系不能继承

类与对象（中） -- 默认成员函数

类的6个默认成员函数

分类

对于一个空类，编译器会自动生成6个不会显示实现的默认成员函数

- 初始化和清理
 - Constructor 构造函数主要完成初始化工作
 - 大部分类都不会让编译器默认生成构造函数，都要自己写。显式地写一个全缺省函数，非常好用
 - 特殊情况下才会默认生成。比如用两个栈生成一个队列时
 - 每个类最好都要提供默认构造函数
 - Destructor 析构函数
 - 申请了内存资源的类需要显式写析构函数，如Stack, Queue，否则会造成内存泄漏
 - 不需要显式地写析构函数
 - 一些没有资源需要清理的类比如Date
 - 或者是MyQueue这样的类中会调用自定义结构的默认析构函数完成清理
- 拷贝复制
 - Copy constructor 拷贝构造
 - 和析构一样，若申请了内存资源则需要显式定义
 - 不需要显式地写拷贝构造
 - 如Date这种默认生成会完成浅拷贝
 - 或者是MyQueue这样的类中会调用自定义结构的默认拷贝复制函数完成任务
 - Assignment operator overloading （拷贝）赋值运算符重载：和拷贝构造一样也会面临自定义结构的浅拷贝问题，因此自定义结构也需要显式定义
- 取地址即const取地址重载：主要是普通对象和const对象取地址，这两个很少会自己实现，默认生成的够用了

三/五法则

- 分配了动态内存的构造函数必然要实现析构函数，因此也必然要实现拷贝构造和重载赋值运算符
- 需要拷贝操作的类也需要赋值操作，反之亦然

构造函数 *Constructor*

概念

- 构造函数自动初始化类的**非static成员属性**。构造函数的名字与类名相同，没有返回值。创建类类型对象时由编译器自动调用，以保证每个数据成员都有一个合适的初始值，并且在对象整个生命周期内只调用一次
- 构造函数的任务不是构造对象和开辟栈帧空间，而是对实例化对象进行初始化

特性

```
1  // 函数重载，给默认值
2  Date() {
3      _year = 1;
4      _month = 0;
5      _day = 0;
6  }
7  Date(int year, int month, int day) {
8      _year = year;
9      _month = month;
10     _day = day;
11 }
12 // 调用
13 Date d1(2022, 9, 15);
14 Date d2; // 赋默认值
15 -----
16 // 或者是直接给缺省参数，这样实现比较好
17 Date(int year=1, int month=0, int day=0) {
18     _year = year;
19     _month = month;
```



```
20     _day = day;
21 }
```

- 函数名与类名相同
- 无返回值
- 对象实例化时编译器自动调用对应的构造函数
- 构造函数可以重载
- 因为构造函数无论如何要取得类属性的控制权，所以它不能被声明为**const**

定义构造函数

若没有定义，就自动生成空的默认构造；若定义了非默认构造就不会生成默认构造，必须手动定义一个默认构造，或者用 **default** 强制生成

- 若类中没有显式定义构造函数，则C++编译器会自动生成一个**无参的默认构造函数**（空默认构造）default constructor，一旦用户显式定义编译器将不再生成
 - C++把类型分成了两类
 - 内置类型/基本类型：int, double, char, pointer, ...
 - 自定义类型：struct, class, ...
- **C++设计的缺陷**：默认生成的构造函数对**内置类型不做处理**；而自定义类型成员会去调用它的默认构造函数

```
1  class Time {
2  public:
3      Time() { // 构造函数
4          cout << "Time()" << endl;
5          _hour = 0;
6          _minute = 0;
7          _second = 0;
8      }
9  private:
10     int _hour;
11     int _minute;
12     int _second;
```

```

13 };
14 class Date {
15 public:
16     // 没有自己写的构造函数，编译器会自动生成
17     void Print()
18         cout << _year << " " << _month << " " << _day << endl;
19 private:
20     int _year; // 内置类型
21     int _month; // 内置类型
22     int _day; // 内置类型
23     Time _t; // 构造类型
24 };
25 // 用默认构造函数调用
26 Date d;

```

Name	Value
d	{_year=-858993460 _month=-858993460 _day=-858993460 ...}
_year	-858993460
_month	-858993460
_day	-858993460
_t	{_hour=0 _minute=0 _second=0}

通过调试可以发现，内置类型的 `_year`、`_month`、`_day` 都是随机值，没有被初始化，但自定义类型 `_t` 被初始化了

- 这个缺陷在C++11中打了补丁：可以为内置类型成员变量在类中声明时给默认值（缺省参数）来修正

```

1 // 一定要注意，这个不是给初始值，而是为了修正构造函数不处理内置类型的缺陷给的缺省值!!!
2 private:
3     int _year=1;
4     int _month=0;
5     int _day=0;
6     Time _t; // Time* _t 指针也是内置类型，不会被初始化

```

- 默认构造函数并不仅仅指默认生成的构造函数，而是有三类都可以称为默认构造函数。其特点是不传参数就可以调用的，且只能存在一种
 - 我们不写，由编译器自动生成的

- 我们自己写的全缺省构造函数
- 我们自己写的无参构造函数（空参数列表）
- 若写了一个其他的非默认构造函数，那么一定要自己实现上面的三个默认构造函数之一。因为编译器不会自动生成默认构造函数，此时编译器会报如下的错误

没有合适的默认构造函数可用

- 注意：如何调用默认构造函数？不要加（）

```
1 // 假设有Sales_data类
2 Sales_data obj(); // 意思是声明一个返回类型为Sales_data的obj函数
3 Sales_data obj; // 意思是默认构造实例化Sales_data的一个对象obj
```

初始化列表在构造函数中的角色

构造函数体赋值/函数体内初始化

```
1 class Time {
2 public:
3     Time(int hour = 0) { // 全缺省的默认构造函数，否则会报“没有可用的默认构造函数”的错误
4         _hour = hour;
5     }
6 private:
7     int _hour;
8 };
9 class Date() {
10 public:
11     Date(int year, int hour) {
12         _year = year;
13         Time t(hour);
14         _t = t;
15         // 自定义结构Time成员hour是私有的，无法直接被Date取到
16         // 只能通过Date的构造函数中新建一个Time类后再赋值给Date的属性_t
17     }
18 private:
```

```

19     int _year;
20     Time _t;
21 };
22 int main() {
23     Date d(2022, 1);
24     return 0;
25 }

```

- 初始化是在初始化列表中完成的。构造函数体中的语句是赋值，而不是初始化。初始化只能有一次，而构造函数体内可以多次赋值
- 若类成员中包含本身没有默认构造函数的自定义结构时初始化会很麻烦，如上所示，自定义结构Time成员hour是私有的，无法直接被Date取到，只能通过在Date的构造函数中新建一个Time类后再赋值给Date的属性 _t （前提是自定义类有自己的默认构造函数）
- 通过上述方式初始化起始本质也是通过初始化列表初始化的，相当于绕了一个大圈子

初始化列表 Initializaiton list

- 示例

```

1  class Time {
2  public:
3      Time(int hour = 0) { // 全缺省的默认构造函数，否则会报“没有可用的默认构造函数”的错
4          _hour = hour;
5      }
6  private:
7      int _hour;
8  };
9  class Date
10 {
11 public:
12     // 初始化列表可以认为是类成员定义的地方
13     Date(int year, int hour, int& ref)
14         : _year(year)
15         , _t(hour)
16         , _ref(ref)

```

```

17         , _n(10)
18     {}
19 private:
20     // 声明
21     int _year = 0; // 内置对象给缺省值，该缺省值是给初始化列表的，此时内置类
        型若没有显式给值就会用这个缺省值
22     Time _t; // 自定义类型成员
23     int& _ref; // 引用成员变量
24     const int _n; // const成员变量
25 };
26
27 int main() {
28     int y = 0;
29     Date d(2022, 1, y); // 对象整体定义
30 }

```

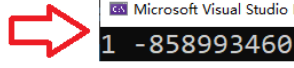
- 初始化列表可以认为是成员变量初始化的地方。初始化列表是被自动调用的，即使是完全空的构造函数也会自动调用，因此可以认为构造函数里的是二次赋值，private里的只是声明，真正的初始化是在初始化列表之中。也就是说实例化的过程是：类属性声明（若有缺省值就直接初始化） -> 初始化列表赋值 -> 构造函数内部赋值（若有的话）
- 注意点
 - 每个成员变量在初始化列表中只能出现一次，因为初始化只能有一次
 - 当类中包含以下成员时，其**必须**放在初始化列表位置进行初始化
 - 自定义类型成员，且该类没有默认构造函数时，否则会编译错误
 - 引用成员变量，引用在定义时必须初始化
 - const成员变量，const只有一次定义机会，之后不能重新赋值
 - 推荐使用初始化列表进行初始化
 - 有默认构造函数的自定义类型也推荐用初始化列表。通过调试可以发现，有默认构造函数的自定义结构通过构造函数初始化时也是借助构造函数，然后再赋值。不如直接使用列表初始化，只需要调用一次构造函数即可。
 - 内置类型也推荐使用初始化列表，当然内置类型在函数体内初始化也没有明显的问题

- 统一的建议：能使用初始化列表就使用初始化列表来初始化，基本不会有什么问题，肯定比在函数体内好
- 尽量使用初始化列表初始化
- 成员变量在类中的声明次序就是其在初始化列表中的初始化顺序，与其在初始化列表中的先后次序无关

```
#include <iostream>
using namespace std;

class A
{
public:
    A(int a)
        : _a1(a)
        , _a2(_a1)
    {}
    void Print()
    {
        cout << _a1 << " " << _a2 << endl;
    }
private:
    int _a2;
    int _a1;
};

int main()
{
    A aa(1);
    aa.Print();
}
```



可以发现上面程序的结果是1和随机值。这是因为根据声明，`_a2` 在初始化列表中应该首先被定义，其定义值为 `_a1`，然而此时 `_a1` 还没有被定义，所以其为随机值

委托构造

C++11的新特性：委托构造 delegating constructor 调用类的其他构造函数

析构函数 *Destructor*

概念

析构函数与构造函数功能相反，析构函数不是完成对对象的销毁。局部对象销毁工作是由编译器来完成的。而对象在销毁时会自动调用析构函数，完成对象中非static资源的清理工作

特性

- 析构函数名是在类名前加上字符 `~`
- 无参数无返回值类型，无参数所以析构函数不能重载

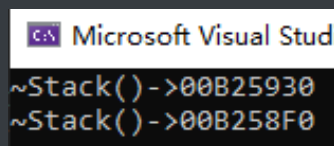
- 一个类只能有一个析构函数。若无显式定义，系统会自动生成默认的析构函数
- 对象生命周期结束时，C++编译系统自动调用析构函数
- 若析构函数中没有很多工作时，有些编译器会把它优化掉
- 默认生成的析构函数和构造函数类似
 - **内置类型不处理**，但这里不是设计bug，而是系统无法处理。因为系统无法判断当前指针是否是动态开辟内存出来的还是有其他作用
 - **自定义类型成员会去调用它的析构**

```
1  class Stack {
2  public:
3      Stack(int capacity = 4) {
4          _array = (DataType*)malloc(sizeof(DataType) * capacity);
5          if (NULL == _array) {
6              perror("malloc fail!\n");
7              return;
8          }
9          _size = 0;
10         _capacity = capacity;
11     }
12     void Push(DataType data) {
13         _array[_size] = data;
14         _size++;
15     }
16     ~Stack() { // Destructor
17         cout << "~Stack()->" << _array << endl;
18         free(_array);
19         _size = _capacity = 0;
20         _array = nullptr;
21     }
22 private:
23     DataType* _array;
24     int _size;
25     int _capacity;
26 };
27 class MyQueue {
```

```

28 public:
29     void Push(int x){}
30 private:
31     size_t _size = 0;
32     Stack _st1;
33     Stack _st2;
34 };
35 // 调用
36 int main {
37     Myqueue q;
38     return 0;
39 }

```



在析构函数 `~Stack()` 中打印了一下，可以发现在自定义类型Stack中调用了两次析构函数

- 若类中没有动态开辟内存时，析构函数可以不写，直接使用编译器生成的默认析构函数；由动态开辟内存时则一定要写，否则会造成内存泄漏

拷贝构造函数 *Copy constructor*

概念


```

1  int a = 1;
2  int b = a; // 普通变量的复制
3
4  Date d1(2022, 7, 23);
5  Date d2(d1); // 类对象的拷贝
6  Date d3 = d1; // 也可以这么拷贝
7
8  // 拷贝构造函数
9  Date(const Date& d) {
10     _year = d._year;
11     _month = d._month;
12     _day = d._day;
13 }

```

和普通变量一样，有时候想要创建一个一模一样的新对象。为达成这种性质，需要给类添加拷贝构造函数

特性

- 拷贝构造函数是构造函数的一个重载形式
- 拷贝构造函数的参数只有一个且必须是类类型对象的引用，使用传值方式编译器直接报错，因为会引发无穷递归调用

```
illegal copy constructor: first parameter must not be a 'Date'
```

- 以传值方式传参需要开辟临时空间，从而拷贝传入的参数
- 但是为了拷贝传入的参数，本身又需要调用拷贝构造函数，从而形成了无穷递归

先做下图试验，通过调试可以发现：对于两个普通的函数 `get1`, `get2`，当其实参采用类传值传参时会调用类的拷贝构造函数，而传引用则不会。这说明传参的时候为了创建类实参的临时拷贝，本身就需要调用类的拷贝构造函数

```
Date(Date& d) // Wrong copy constructor
{
    cout << "Date(Date& d) " << endl;
    _year = d._year;
    _month = d._month;
    _day = d._day;
}

void get1(Date d){}
void get2(Date& d){}

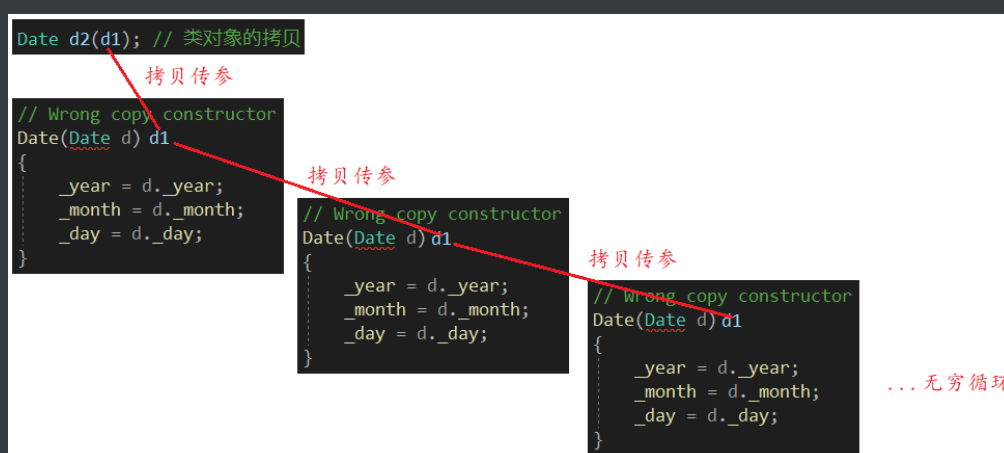
int main()
{
    Date d1(2022, 7, 23);
    get1(d1);
    get2(d1);

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Date(Date& d)
D:\C_Learning\Lecture\cpp_fundamental\Le
with code 0.
To automatically close the console when
le when debugging stops.
Press any key to close this window . . .
```

也就是说，传参要调用拷贝构造函数，我们必须拷贝它的实参来创建一个临时变量，但为了拷贝实参，又需要再次调用拷贝构造函数，无限循环



- 添加 `const` 将引用设置为只读，防止误操作对实参的改变

默认拷贝构造的浅拷贝问题

若未显式定义，编译器会生成默认的拷贝构造函数，或者称为合成拷贝构造函数 `synthesized copy constructor`。默认的拷贝构造函数对象将非static成员的内存存储按字节序完成拷贝，这种拷贝叫做**浅拷贝/位拷贝 shallow copy/bit-wise copy**

浅拷贝问题指的是在对象的拷贝操作中，仅仅进行成员变量的简单拷贝，**而没有对成员变量中指针指向的动态内存进行深度复制**的情况。这可能导致多个对象共享同一块内存，进而在释放内存时出现问题

- 对于内置类型不显式定义也可以，但若有内存开辟则拷贝的是指向开辟空间的指针，那么浅拷贝会存在以下问题
 - 一个对象修改会应先给另一个对象

- 会析构两次，即free同一块内存空间两次，程序崩溃

调用默认拷贝构造函数，可以发现两个字符串指向同一个地址

名称	值	类型
s1	{_str=0x00a9f880 "hello world" _size=11 _capacity=11 }	wjfs:string
_str	0x00a9f880 "hello world"	char *
_size	11	unsigned int
_capacity	11	unsigned int
s2	{_str=0x00a9f880 "hello world" _size=11 _capacity=11 }	wjfs:string
_str	0x00a9f880 "hello world"	char *
_size	11	unsigned int
_capacity	11	unsigned int

- 解决方法：自己实现深拷贝 Deep copy/值拷贝 Member-wise copy，深拷贝的实现见string章节

拷贝构造函数的典型调用场景

拷贝构造函数一般作为拷贝初始化使用

- 用 = 来拷贝初始化，即使用已存在对象创建新对象
- 将一个对象作为实参传递给一个**非引用类型**的形参
- 从一个返回类型为**非引用类型**的函数返回一个对象
- 用初始化列表来初始化一个数组中的元素或一个聚合类中的成员
- 容器使用 push 或 insert 成员（注意emplace是直接构造）

通过 explicit 禁止拷贝构造

explicit 声明的**某种特定类型**的构造函数只能用于直接初始化，不能用于拷贝初始化（即拷贝构造）。不过这种情况是比较少见的

```
1 explicit Sales_data(const std::string &s): bookNo(s) {}
2 Sales_data item1(null_book); // 正确，直接初始化
3 Sales_data item2 = null_book; // 错误，explicit禁止了拷贝构造
```

赋值运算符重载 *Assignment Operator Overloading*

赋值运算符重载

```
1 Date& operator=(const Date& d) {  
2     // this出了类域还存在  
3     if (this != &d) { // 避免自己给自己赋值  
4         _year = d._year;  
5         _month = d._month;  
6         _day = d._day;  
7     }  
8     return *this; // 返回this指向的值为了支持连续赋值 d2 = d1 = d3  
9 }
```

赋值运算符重载的特性

- 参数类型：const Date& **传引用**提高效率，且不修改Date类，因此用const保护
- 返回值类型：返回引用可以避免传值拷贝，提高返回效率，有返回值的目的是为了支持连续赋值。如果传指针返回，外面接受还需要解引用，非常别扭
- 要检查是否自己给自己赋值（不写也不会报错，只是为了提高效率）
- 返回*this：要符合连续赋值的含义
- 赋值运算符只能重载成类的成员函数不能重载成全局函数，会被覆盖。赋值运算符如果不显式实现，编译器会生成一个默认的。此时用户再在类外自己实现一个全局的赋值运算符重载，就和编译器在类中生成的默认赋值运算符重载冲突了，故赋值运算符重载只能是类的成员函数

默认赋值运算符重载的浅拷贝问题

用户没有显式实现时，编译器会生成一个默认赋值运算符重载，进行浅拷贝，会产生和默认拷贝构造函数的浅拷贝一样的问题。若有自定义类型成员就必须显式实现

而且相比于默认构造函数的浅拷贝，赋值的对象是一个已经存在的对象，所以默认赋值运算符重载还会引起**内存泄漏**的问题

拷贝构造和赋值运算符重载的浅拷贝问题可以看[string类模拟实现](#)

赋值运算符重载和拷贝构造的区分

```
1  class A;
2  A a1;
3  A a2(a1); // 拷贝构造
4  A a3 = a1; // 拷贝构造
5  A a4;
6  a4 = a1; // 赋值
```

拷贝构造是针对还没有定义的对象，而赋值是已经存在的对象

阻止拷贝

构造函数如果显式定义了就不会生成默认构造函数，但拷贝构造与赋值运算符重载函数不同，即使已经显式定义了拷贝构造，编译器仍然会生成默认拷贝构造和默认的赋值运算符重载。这种特性有时候会带来一些麻烦，因此有时候有必要阻止它们生成

```
1  istream& (const istream&) = delete;
```

比如说istream禁止了默认拷贝构造，因为不应该有多个对象写同一个输入流

C++11: delete关键字

在函数的参数列表后面加上delete关键字来定义为删除的函数 deleted function

和default不同，delete可以用于任何函数，因为这样可以避免函数重载的多重匹配问题

注意：不允许将析构函数声明为delete

默认的拷贝控制成员可能是delete

若一个类有数据成员不能默认构造、拷贝、赋值或销毁，则对应的成员函数将被定义为delete

- 若类的某个成员的拷贝构造函数是删除的或不可访问的，则类的默认拷贝构造函数被定义为delete

- 若类的某个成员的赋值运算符是删除的或不可访问的或者类有一个const的或者引用成员，则类的默认赋值运算符被定义为delete

C++11之前：声明为private

取地址及const取地址操作符重载

- 它们是默认成员函数，我们不写编译器也会自动生成。自动生成就够用了，所以一般是不需要我们自己写的
- 非常特殊的场景：不想让别人取到这个类型对象的地址（或者设置为私有直接报错）

```
1 Date* Date::operator&() {
2     return nullptr;
3 }
4 const Date* Date::operator&() const {
5     return nullptr;
6 }
```

类与对象（下）——特殊类

运算符重载

以Date类的实现为例，详细实现见 [Lecture6_class4_date_20220724](#)

运算符重载 Operator Overloading

- 运算符重载是为了让自定义类型对象也可以使用运算符像内置对象那样进行运算
- 重载的运算符是具有特殊名字的函数：它们的名字由关键字 operator 和其后要定义的运算符共同组成
- 除了 `.* :: sizeof ? : .` 5个运算符外其他的运算符都可以被重载
- 重载运算符函数的参数数量与该运算符作用的运算对象数量一样多

- 当一个重载的运算符是成员对象时，this绑定到左侧运算对象

其他运算符重载的复用实现

- 几乎所有的类的运算符重载都只需要自己实现 `= == >` 或者 `= == <`，其他的比较运算符重载（不是加减乘除！）都可以复用这几个运算符
- Date类的 `+=` 复用（`-=` 同理）
 - 用`+=`复用`+`
 - 用`+`复用`+=`：缺点式拷贝次数过多，一直在调用`+`中的`=`

```
1  Date& Date::operator+=(int day) {
2      // 要改变自身，所以可以直接对自身操作，采用传引用返回
3      if (day < 0)
4          return *this -= -day;
5      _day += day;
6      while (_day > GetMonthDay(_year, _month)) {
7          _day -= GetMonthDay(_year, _month);
8          ++_month;
9          if (_month == 13) {
10             _year++;
11             _month = 1;
12         }
13     }
14     return *this;
15 }
16 Date Date::operator+(int day) const {
17     // Date ret(*this); // 拷贝构造
18     Date tmp = *this; // 注意这里也是拷贝构造，两个已经存在的值才是赋值。两种写法等价
19     tmp += day; // 用 += 复用
20     return tmp; // 传值拷贝构造
21 }
```

根据加法的性质，两数相加会生成临时变量，并将临时变量赋给接收值，因此需要新建一个tmp临时变量并采用传值返回。同时拷贝构造所产生的临时变量在栈帧销毁后也被销毁了，所以只能传值返回，不能传引用，否则会引起野指针问题

- 一个类需要重载哪些运算符要看哪些运算符对这个类型有意义，比如Date类除法和乘法就没有意义
- 前置++和后置++重载：特殊处理，使用函数重载，默认无参为前置，有参为后置

```
1  Date& operator++() { // 前置
2      *this += 1;
3      return *this;
4  }
5  Date operator++(int) { // 后置，传的参数没有效果，仅仅是用来区分
6      Date tmp(*this);
7      *this += 1;
8      return tmp;
9  }
```

复用+=，因为前置++会直接改变*this，因此可以传引用返回，而后置++是先用再++，因此要保留一个*this的备份tmp，所以要传值返回tmp

运算符重载和函数重载的应用（以cout为例）

回顾C++IO流的内容，<< 流输出运算和 >> 流输入，因为C++库里写好了运算符重载。cout 对象能做到自动识别类型，这是因为所有内置类型构成了函数重载

为了支持对自定义类型Date类的流输入和流输出，我们需要分别为 istream 和 ostream 类重载针对Date类的 >> 和 << 运算符重载

```
1  class Date {
2      // 友元函数 -- 这个函数内部可以使用Date对象访问私有保护成员
3      friend ostream& operator<<(ostream& out, const Date& d);
4      friend istream& operator>>(istream& in, Date& d);
5      //...
6  }
7  //流输出重载
```



```

8  inline ostream& operator<<(ostream& out, const Date& d) {
9      //无法访问私有，使用友元
10     //out << _year << "-" << _month << "-" << _day << endl;
11     out << d._year << "-" << d._month << "-" << d._day << endl;
12     return out;
13 }
14 //流提取重载
15 inline istream& operator>>(istream& in, Date& d) {
16     in >> d._year >> d._month >> d._day;
17     assert(d.CheckDate());
18     return in;
19 }

```

实习的细节

- 因为是针对 `istream` 和 `ostream` 类运算符重载，所以第一个参数必须是 `istream& in` 和 `ostream& out`，所以这个重载不能写在 `Date` 类里面，因为在 `Date` 里的话默认第一个参数是 `this` 指针，因此一定要在 `Date` 类外实现
- 在 `Date` 类外实习的话又会产生取不到私有的 `_year` 等 `Date` 类成员的问题
- 所以最后的解决方案是将重载函数设置为 `Date` 类的友元，以便让其取到 `Date` 类成员

类的类型转换

转换构造函数 `converting constructor` 用于**将其他类型转换为类的对象**，转换构造函数没有返回类型

类型转换运算符 `converting operator` 用于**将类的对象转换为其他类型**，类型转换运算符具有返回类型

转换构造函数

构造函数不仅可以构造与初始化对象，对于**单个参数**或者**除第一个参数无默认值其余均有默认值**的构造函数，还具有类型转换的作用。把这种特殊的构造函数称为**转换构造函数**

converting constructor

内置类型是可以隐式类型转换成自定义类型的，编译器会自动调用构造和拷贝构造（有些编译器自动优化为只有自定义）

- 转换构造函数是一种特殊的构造函数，用于将其他类型的对象转换为类的对象
- 它被用于隐式地执行类型转换，当需要将一个对象从一种类型转换为另一种类型时，编译器会自动调用适当的转换构造函数
- 转换构造函数在类内声明，没有返回类型，并且以类名作为函数名
- 虽然不可以指定返回类型，但是会隐式返回一个当前类

```
1  class MyInt {
2  public:
3      MyInt(int value) : m_value(value) {} // 转换构造函数将 int 类型转换为
        MyInt 类型
4      // ...
5  private:
6      int m_value;
7  };
8
9  int main() {
10     int num = 42;
11     MyInt myNum = num; // 隐式调用转换构造函数将 int 类型转换为 MyInt 类型
12     // ...
13 }
```

类型转换运算符

```
1  operator CONVERTED_TYPE() const;
```

- 类型转换运算符 converting operator 是一种特殊的成员函数，用于将类的对象转换为其他类型
- 它被用于隐式或显式地执行类型转换，当需要将一个对象从一种类型转换为另一种类型时，编译器会自动调用适当的类型转换运算符
- 类型转换运算符在类内声明，以 operator 关键字开头，后面跟着要转换的目标类型

- 类型转换运算符有返回类型，用于表示转换结果
- 类型转换运算符通常不应该改变待转换对象的内容，所以一般要被定义为const成员

```
1  class MyDouble {
2  public:
3      operator double() const { return m_value; } // 类型转换运算符将
        MyDouble 类型转换为 double 类型
4      // ...
5  private:
6      double m_value;
7  };
8
9  int main() {
10     MyDouble myNum(3.14);
11     double num = myNum; // 隐式调用类型转换运算符将 MyDouble 类型转换为
        double 类型
12     // ...
13 }
```

注意：Date d1(2022); 和 Date d2 = 2022; 虽然结果是一样的，但过程并不一样，前者是直接调用构造函数，而后者包含了先构造（创建临时变量）和拷贝构造（隐式类型转换）以及编译器的优化

什么是隐式调用 implicit call

隐式调用指的是编译器自动执行的函数调用，而无需显式地在代码中指定函数名和参数。它是通过一定的规则和转换机制来进行的，以便在需要的地方自动进行类型转换和函数调用

隐式调用可以发生在多种情况下，其中包括

1. **隐式类型转换**：当参数的类型与函数的参数类型不完全匹配时，编译器可以根据一组转换规则自动进行类型转换，以便使函数调用成为可能。例如，将一个整数传递给一个接受浮点数参数的函数，编译器可以隐式地将整数转换为浮点数

2. **拷贝构造函数的隐式调用**：当通过赋值操作符、函数返回值或者函数参数传递对象时，编译器会隐式调用拷贝构造函数来创建对象的副本。这样可以使得对象在不同的代码段中进行传递和复制
3. **隐式转换构造函数的调用**：当创建一个对象时，编译器会根据参数的类型选择适当的构造函数进行调用。如果构造函数具有一个参数且不是 `explicit` 的，那么编译器可以隐式地调用这个构造函数，以便根据参数类型创建对象

explicit 关键字：禁止单参数类的隐式类型转换

隐式类型转换往往给用户带来的不是帮助，而是各种bug。比如说若常用的string类的自动转换和转换为bool型的隐式类型转换经常会带来一些问题

因此可以通过 `explicit` 关键字来禁止这种**单参数**的隐式类型转换。只有在用户知道自己在做什么，也就是说用 `static_cast<>` 显示的给出类型转换时转换才会发生

`explicit` 只能用于有一个实参的构造函数，需要多个实参的构造函数不能用于执行隐式转换，所以无须将这些构造函数指定为 `explicit`

拷贝构造是一个实参的引用，拷贝构造函数本质上也是一个单参数的构造函数，所以当用 `explicit` 禁止了**某种参数类型**的单参数构造的时候也禁止了**这种参数类型**的拷贝构造的隐式调用

匿名对象 Anonymous object 及其单参数构造函数

```
Date(2000);
```

 生命周期只有这一行

可以在有仿函数参数的函数模板中使用

聚合类

定义

聚合类 Aggregate Class 是一种特殊的类类型，具有特定的属性和行为。聚合类的主要特点是

- 所有成员都是 `public`
- 没有定义任何构造函数、基类、虚函数或私有/受保护的静态数据成员

- 没有类内初始值

```
1 struct Data {  
2     int ival;  
3     string s;  
4 }
```

需要注意的是，从C++20起，聚合类的定义更加宽松，允许包含带有默认参数的非静态数据成员、内联函数等特性，但依然要满足成员都是公有的、没有用户定义的构造函数等要求

聚合类在C++中用途和优势

总之，聚合类在C++中用于表示简单的数据结构和对象，通过简洁的成员初始化语法和与C的兼容性，方便地进行数据操作、交互和持久化操作

- 快速的成员初始化：聚合类可以使用聚合初始化语法进行快速的成员初始化，无需显式定义构造函数。通过简洁的初始化语法，可以方便地初始化聚合类的成员变量。注意：**初始化的顺序一定要和聚合类成员声明的顺序一样**
- 适用于数据结构和简单对象：聚合类常用于表示简单的数据结构和对象，比如结构体。由于聚合类的成员都是公有的，可以直接访问和修改成员变量，方便进行数据操作和传递
- 与C兼容性：聚合类可以与C语言代码进行无缝交互，因为它们在内存布局上与C结构体相似。这使得聚合类在与现有的C库或C接口进行交互时很有用
- 便于序列化和持久化：聚合类的简单结构使得对其进行序列化和持久化操作变得简单。可以直接将聚合类的数据写入文件或进行网络传输

内部类 *Inner Class* (Java常用)

概念

如果一个类定义在另一个类的内部，这个内部类就叫做内部类。内部类是一个独立的类，它不属于外部类，更不能通过外部类的对象去访问内部类的成员。外部类对内部类没有任何优越的访问权限

特性

- 内部类受外部类的类域限制，也受访问限定符的限制
- 内部类天生是外部类的友元
- `sizeof(外部类)==外部类`，和内部类没有关系

特殊类设计

Cpp 第44节课，2小时左右开始

不能被拷贝的类

- C++98

```
1 class CopyBan {
2     // ...
3 private:
4     CopyBan(const CopyBan&);
5     CopyBan& operator=(const CopyBan&);
6     //...
7 };
```

- 将拷贝构造函数与赋值运算符重载只声明不定义
- 将拷贝构造和赋值运算符重载定义为私有，这样就不能在类外面实现了
- 如果非要说缺点的话，那就是还有一种情况，可以在类里重新定义拷贝构造
- C++11：使用 `delete` 关键字禁止编译器默认生成拷贝构造和复制运算符重载

```
1 class CopyBan {
2     // ...
3     CopyBan(const CopyBan&)=delete;
4     CopyBan& operator=(const CopyBan&)=delete;
5     //...
6 };
```

- 防拷贝应用

- `unique_ptr`
- `thread`
- `mutex`
- `istream`
- `ostream`

只能在堆上创建对象的类

创建类有三种构造方法+1种拷贝方法

```
1 HeapOnly obj1; //栈上
2 static HeapOnly obj2; //静态区
3 HeapOnly* obj3 = new HeapOnly; //堆上
4 HeapOnly obj4(obj1); //拷贝
```

要创建对象需要通过构造函数和拷贝构造，只要想办法把它们堵死到只能在堆上创建对象就可以

- 首先考虑一下能否通过析构函数私有来解决

```
1 class HeapOnly {
2 private:
3     ~HeapOnly() {}
4 private:
5     int _a;
6 }
7
8 int main() {
9     HeapOnly hp1; // 报错，自动调用析构
10    static HeapOnly hp2; // 报错，自动调用析构
11    HeapOnly* ptr = new HeapOnly; // 通过，自定义类型不会自动调用析构
12    // delete ptr; // 需要手动调用，此时报错
13    return 0;
14 }
```

hp1和hp2是可以的，因为它们强制调用析构函数，但是ptr是自定义类型，不会调用默认析构函数，所以反而避开了私有析构

- 实际采用的方法是
 - 将类的构造、拷贝构造声明成私有，防止别人调用拷贝在栈上生成对象。赋值拷贝可写可不写，因为都禁止了构造，也不会有对象被new出来
- 提供一个静态的成员函数，在该静态成员函数中完成堆对象的创建
 - 调用成员函数 CreateObject 需要现有类对象，然后类对象又只能通过成员函数得到，一个经典的先有鸡还是先有蛋问题
 - 因此将 CreateObject 设置为不需要没有this指针的静态成员函数，且构造函数不需要this指针就可以调用

```
1 class HeapOnly {
2 public:
3     static HeapOnly* CreateObject() {
4         return new HeapOnly // new出来的对象返回指针，不需要调拷贝构造
5     }
6 private:
7     HeapOnly() {}
8     HeapOnly(const HeapOnly&) = delete; //防拷贝
9     HeapOnly& operator=(const HeapOnly &hp) = delete; //禁止赋值拷贝
10 };
```

只能在栈上创建对象的类

栈的局部对象传值返回必然要调用拷贝构造，因为有编译器优化，把传值返回的拷贝构造省去了，相当于直接构造。要new对象必须要调用其构造函数，设置成私有后new就不能访问构造函数了

这里也不是移动构造，因为默认移动构造必须要不写构造函数

第二种思路是可以重载 operator new() 和 operator delete()，使无法从堆上创建对象。但这种写法有缺陷，无法阻止全局new对象和静态对象生成

```
1 class StackOnly {
```



```

2  public:
3      static StackOnly* CreateObject() {
4          return StackOnly(); //编译器优化，传值返回没有拷贝构造
5      }
6      ///// 不能禁拷贝构造，因为局部对象传值返回要调用
7      //StackOnly(const StackOnly&) = delete; //防拷贝
8      //StackOnly &operator=(const StackOnly &st) = delete;
9
10     //void* operator new(size_t size) = delete;
11     //void operator delete(void* p) = delete;
12 private:
13     StackOnly() {}
14 };
15
16 static StackOnly copy2(st1); //不禁拷贝就不能解决这种拷贝，这是一个缺陷

```

不能被继承的类

- C++98：父类构造函数私有化，这样子类就不能调用父类的构造函数，则无法继承
- C++11：用 `final` 关键字修饰父类，也就称为所谓的最终类

```

1  //C++98
2  class NonInherit {
3  public:
4      static NonInherit GetInstance() {
5          return NonInherit();
6      }
7  private:
8      NonInherit() {}
9  };
10 //C++11
11 class A final {};

```

只能创建一个对象的类（单例模式 Singleton）

[快速记忆23种设计模式 - 知乎 \(zhihu.com\)](#)

设计模式 Design Pattern：是一套被反复使用、多数人知晓的、经过分类的代码设计经验总结，具体可以看上面的文章。比较重要的有适配器模式、单例模式、迭代器模式、观察者模式和工厂模式

Java对设计模式的要求是比较高的，因为Java本身的库设计就用到了很多设计模式。而C++等语言则没有很普遍使用设计模式。设计模式是有益的工具，可以提高代码的可读性、可维护性和灵活性。然而，过度使用设计模式可能导致复杂性增加、过度工程化、不必要的复用、过度抽象化和性能损失等劣势。在应用设计模式时，应根据具体的问题和需求进行合理的权衡，避免滥用和不必要的复杂性

单例模式 Singleton Pattern：一个类只能创建一个对象。该模式可以保证系统中在当前进程中该类只有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息

单例模式在何时创建这个唯一的对象有区别，即

- 饿汉模式 Eager Initialization：不管将来用不用，程序启动时就创建一个唯一的实例对象
构造私有、禁止拷贝，在类外定义全局static成员并用new实例赋值给它用来保存唯一一个实例，类内必须定义static成员以让构造取到、提供一个Get方法生成一个实例

```
1  class Singleton {
2  public:
3      static Singleton* GetInstance() {
4          return _spInst;
5      }
6      void Print() {
7          cout << _spInst << endl;
8      }
9  private:
10     Singleton() {}
11     Singleton(const Singleton&) = delete; //防拷贝
12 }
```

```

13     int _a;
14     //static Singleton _sInst; //仅仅是声明，否则构造私有取不到，声明为类成员就能令定义时取到
15     static Singleton* _spInst; //仅仅是声明
16 };
17
18 //Singleton Singleton::_sInst; // 定义成全局的，而不是私有的，满足饿汉模式的要求（main函数之前就被初始化）
19 Singleton* Singleton::_spInst = new Singleton; //类外才是定义，生成唯一一个实例
20
21 int main() {
22     Singleton::GetInstance()->Print(); //只能这么取，因为不能实例化
23     return 0;
24 }

```

- 懒汉模式 Lazy Initialization：用到的时候在初始化创建（延迟加载）。若单例对象构造十分耗时或者会占用很多资源，而有可能这个对象程序运行时并不会用到。若此时在程序时也要对这个对象进行初始化就会令程序启动非常缓慢

```

1 static Singleton* GetInstance() {
2     if (_spInst == nullptr) { //第一次调用的时候创建对象
3         _spInst = new Singleton;
4     }
5     return _spInst;
6 }
7 Singleton* Singleton::_spInst = nullptr; //定义

```

实际中**懒汉比较常用**，因为饿汉有时候初始化速度太慢，并且当多个单例存在顺序依赖问题时，饿汉静态成员的初始化顺序是不确定的，而懒汉可以控制初始化顺序

单例不用析构，但是不属于内存泄漏，因为这个类一直在使用。不过可以在析构函数里写信息到文件中持久化，可以通过一个垃圾回收的内部类来实现

```

1  class Singleton {
2      // 实现一个内嵌垃圾回收类
3      class CGarbo {
4      public:
5          ~CGarbo(){
6              if (Singleton::_spInst)
7                  delete Singleton::_spInst;
8          }
9      };
10 }
11 // 定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数从而释放单例对象
12 static CGarbo Garbo;

```

复杂的const问题

因为函数的传参行为和const与变量、引用和指针的初始化是一样的，所以直接用函数传参的例子来总结const

```

1  // const与变量没有权限关系
2  // 因为变量都是拷贝，没有内存关系
3  void test_v1(const int i) { printf("%d\n", i); }
4  void test_v2(int ci) { printf("%d\n", ci); }
5
6  // const与引用有权限放大问题
7  // 只能是int或const int 赋值给 const int&, 不能const int赋值给int&
8  void test_r1(const int &ri) { printf("%d\n", ri); }
9  void test_r2(int &ri) { printf("%d\n", ri); }
10
11 // const与指针有权限放大问题
12 // 只能是int*或const int* 赋值给 const int*, 不能const int*赋值给int*
13 void test_p1(const int *pi) { printf("%d\n", *pi); }
14 void test_p2(int *pi) { printf("%d\n", *pi); }

```

```

15
16 int main()
17 {
18     int i = 1;
19     test_v1(i);
20
21     const int ci = 2;
22     test_v2(ci);
23
24     int *pi = &i;
25     const int *cpi2 = &i; //用const int*指向int* 权限缩小ok
26     const int *cpi = &ci;
27     //int *pci = &ci; //用int*指向const int* 权限放大了报错
28
29     test_r1(i);
30     test_r1(ci);
31     test_r2(i);
32     //test_r2(ci); //权限放大
33
34     test_p1(pi);
35     test_p1(cpi);
36     test_p2(pi);
37     //test_p2(cpi); //权限放大
38     return 0;
39 }

```

*const*与变量

const的核心思想是让一个对象不可被修改，一旦尝试修改就报错

```

1 const int buffSize = 512; // 规定缓冲区的大小
2 buffSize = 1024; // 对const赋值，会直接报错

```

修饰变量：置入常量区

- 局部变量：一旦写定就不能再更改了，所以const对象必须初始化。通常被用来修饰只读的数组和字符串
- 全局变量：这种写法应当尽量避免，很容易造成定义冲突
- 编译器通常不为普通const常量新开存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高

const与初始化

因为const对象一旦创建后它的值就不能再被更改，所以**const对象必须要被初始化**

利用const对象去初始化另外一个对象不会对被初始化的对象造成任何影响，这是因为const对象处于 `.rodata` 段（所谓的常量区域），而新的变量可能在任意位置

（`.rodata`、`.data`、`.text`），初始化只是将const对象的值放到寄存器里，然后再mov给新的对象。一旦拷贝完成，const对象和被初始化的新对象也就没有半毛钱关系了

const的底层

对于全局const常量，比如 `const char* ch = "abcdef";` 都会放到 `.rodata`（常量区）只读段。这个段属性（标识位）`sh_flag` 为 `SHF_ALLOC`，没有 `SHF_WRITE`，所以不允许修改它（自我修养P79）

全局const常量会将 `ch` 加入符号表中，因此在其他文件中使用它也需要extern声明，和普通变量的定义和声明是一样的

而局部const变量，则不会放入符号表中，在栈帧中放到某个页中，通过将页属性设置为只读就可以保证局部变量为只读

const与引用

常量引用 Reference to const 与权限放大问题

引用对象是const，即对常量的引用或者叫常量引用：对常量的引用不能被用作修改它所绑定的对象，注意：**常量引用本身必须是任意类型的const**，因为若引用本身不是const，就会有权限放大问题

```

1  int a = 10;
2  int& b = a; // 权限平移
3  cout << typeid(a).name() << endl;
4  cout << typeid(b).name() << endl;
5
6  const int c = 20;
7  cout << typeid(c).name() << endl;
8  // int& d = c; // 权限放大，这么写是错误
9  const int d = c;
10
11 int e = 30;
12 const int& f = e; // 权限缩小，即e可以通过其他方式改变，但f不能改变e
13
14 int ii = 1;
15 double dd = ii; // 隐式类型转换

```

- 权限不能放大，但可以缩小，也就是说可以用常量来引用非常量，但反过来不行
- const引用具有很强的接受度，因为其权限比较小
- 若使用引用传参且函数内若不改变传入参数，建议尽量用const传参

例外：常量引用的类型不必严格匹配

```

1  // double dd = (double)ii;
2  // cout << typeid(ii).name() << endl;
3  // 强制转换只是创建了一个double的临时变量，不改变原变量ii
4
5  // 和整体提升过程一样 在转换过程中产生的是具有常性的 const double 临时变量
6  // double& rdd = ii; // 因此这么写是错误的，本质原因是权限缩小
7  const double& rdd = ii;
8
9  const int& x = 10; // 甚至可以引用常量

```

初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能隐式转换成功

const与指针

常量指针与指针常量

■ const修饰指针的三种情况

- 常量指针 Pointer to const/底层const low-level const: `int const *p` 或者 `const int *p`: const修饰的是 `*p`，不能修改 `*p` 所指向的值，但可以修改p地址本身
- 指针常量 Const pointer/顶层const top-level const: `int* const p`: p一个指针常量，不能修改指针p，即p指向的地址不可修改，但 `*p` 指向的地址所存储的值可以修改
- `const int* const p`: 同时是指针常量和常量指针
- 快速的记忆方法

从右到左，遇到p就翻译成p is a，遇到 * 就翻译成 point to，比如 `int const *p` 是 p is a point(er) to int const; `int* const p` 是 p is const point to int。如何理解常量指针与指针常量？ - 李鹏的回答 - 知乎 <https://www.zhihu.com/question/19829354/answer/44950608>

一个永远不会忘记的方法，**const**默认是修饰它左边的符号的，如果左边没有，那么就修饰它右边的符号

1. `const int *p` 左边没有，看右边的一个，是int，自然就是p指针指向的值不能改变
 2. `int const *p` 此时左边有int，其实和上面一样，还是修饰的int
 3. `int* const p` 修饰的是*，指针不能改变
 4. `const int *const p` 第一个左边没有，所以修饰的是右边的int，第二个左边有，所以修饰的是*，因此指针和指针指向的值都不能改变
 5. `const int const *p` 这里两个修饰的都是int了，所以重复修饰了，有的编译器可以通过，但是会有警告，你重复修饰了，有的可能直接编译不过去
- 只允许权限缩小或平移，不允许权限放大。简单的说就是 `int*` 和 `const int*` 都可以赋值给 `const int*`，但是 `const int*` 不能赋值给 `int*`
 - 当对常变量取地址时，必须要**将指针指向的量设为const**（常量指针 `int const *p`），否则会有权限放大问题；将指向const的指针常量用const数据或非const数据的地址（权限缩小）初始化为或赋值是合法的。


```
1 const int arr[] = {0};
2 const int *ptr = arr; // 原来是const的数据，其指针指向的量也必须设置为const
```

注意这里不要混淆了，不能用const数据初始化指针常量，属于权限放大

- 但是只能把非const的指针赋给普通数据，否则会权限放大

```
1 const int arr[3] = { 0,1,2 };
2 int *ptr = arr; // 编译器不会报错，但ptr权限被放大了，可以通过*ptr修改arr
  中的数据，这种写法仍然是错误的
3 ptr[2] = 3
```

constexpr变量

constexpr 是 C++11 引入的一个关键字，用于声明一个“常量表达式函数”，也可以用于声明一个“常量表达式变量”

constexpr指针的初始值必须是nullptr或者是0，或者是存储在某个固定地址中的对象

constexpr 的出现是为了方便程序员声明常量表达式，从而提高程序的效率。使用 constexpr 声明的常量表达式可以在编译时就被求值，从而避免了在运行时进行计算。字面值类型 literal type 是指在编译时就要被计算确定的值，因此 constexpr 不能被定义在函数体内，除了地址不变的static局部静态对象，所以 constexpr 引用能绑定到静态局部对象上

除了用于声明常量表达式函数和变量之外，constexpr 还可以用于要求函数或变量在编译时必须被求值，避免栈开销，否则编译器会报错。这种约束可以帮助程序员写出更加高效和可靠的代码

```
1 constexpr int factorial(int n) {
2     return (n <= 1) ? 1 : (n * factorial(n - 1));
3 }
4
5 int main() {
6     constexpr int result = factorial(5); // 在编译时就被求值
7     static_assert(result == 120, "factorial(5) should be 120");
8 }
```

指针、常量和类型别名

*const*与函数

函数传参时等同于变量赋值，所以它的规则和前面const与变量、引用、指针的关系一模一样

const传参和返回的作用

- 修饰函数参数
 - 防止修改指针指向的内容
 - 防止修改指针指向的地址
- 修饰函数返回值

若以传指针返回加const修饰，那么函数返回的内容不能被修改，且该返回值只能以加const修饰的同类型指针接收

```
1 const char* GetString(void);
2 char* str = GetString(); // 错误
3 const char* str = GetString(); // 正确
```

Tips：尽量使用常量引用做形参

const类型的左值可以用来接受const或者非const，所以比较保险；但反过来非const就无法接受const了，因为有权限放大

const与函数重载

若参数只有const区别，不会构成函数重载

```
1 void test(int x) {};  
2 void test(const int x) {}; //redefinition  
3  
4 void test(int &x) {};  
5 void test(const int &x) {}; //正确  
6 void test(int *x) {};  
7 void test(const int *x) {}; //正确
```

但若形参是某种类型的指针或引用，则通过区分其指向的是常量对象还是非常量对象可以实现函数重载

当传递一个非常量对象或者指向非常量对象的指针时，编译器会优先选用非常量版本的函数

constexpr函数

- 函数的返回类型及所有形参的类型都得是字面值类型（即算数类型、引用、指针等编译时确定的值）
- 函数体中必须有且只有一条return语句

```
1 constexpr int new_sz() { return 42; }
```

constexpr函数和inline一样，主要是为了加快调用速度

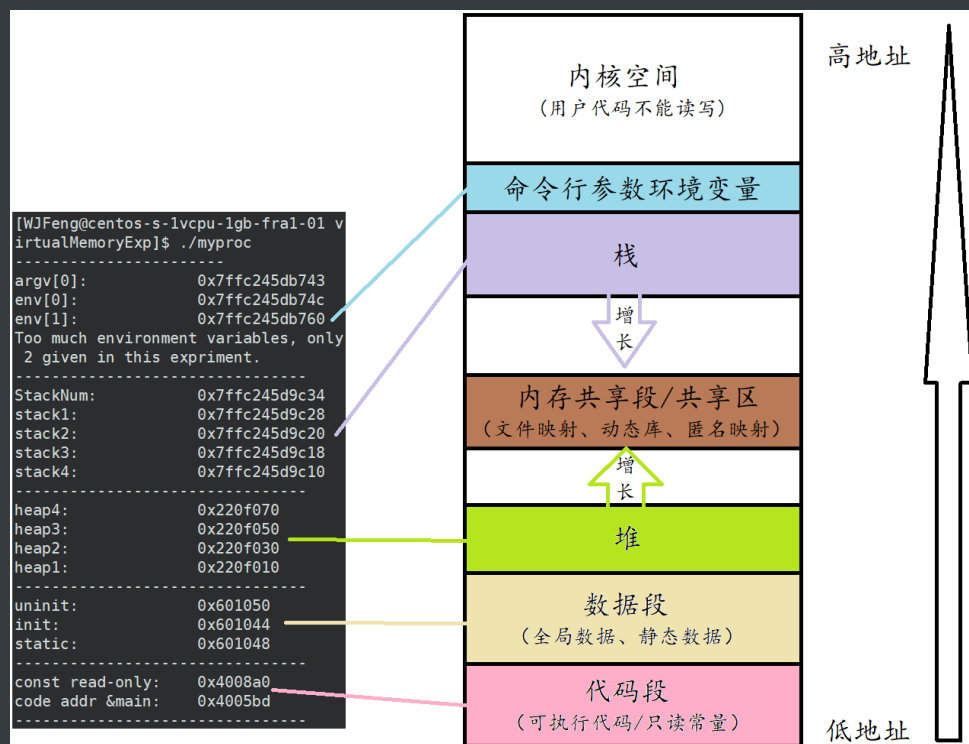
inline和constexpr的多个定义要完全一致，所以一般都直接在头文件中定义一次

const与类

在const成员函数里有说明了

C++内存管理 Memory Management

C/C++内存分布（详见操作系统）



- 堆会多申请一些空间来存放和堆自身有关的属性信息，即cookie数据
- static修饰局部变量的本质就是将该变量开辟在全局区域
- 启动可执行程序，也就是系统新建进程时会换入除堆和栈之外的代码，堆和栈只有在真正要用的时候才会开始开辟内存

一道经典例题

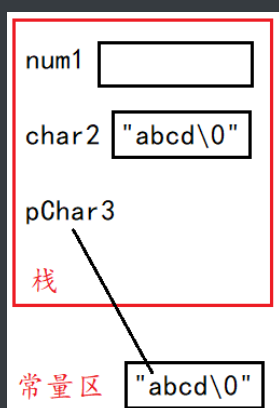
```
1  int globalVar = 1; //globalVar存在数据段（静态区）
2  static int staticGlobalVar = 1; //staticGlobalVar存在数据段（静态区）
3  void Test() {
4      static int staticVar = 1; //staticVar存在数据段（静态区）
5      int localVar = 1; //localVar存在栈上
6      int num1[10] = { 1, 2, 3, 4 }; //num1存在栈上
7      char char2[] = "abcd"; //这个表达式的意义是将常量区的"abcd\0"拷贝一份放到
   //栈上给char2数组，char2存在栈上，*char2指向首元素也在栈上
8      const char* pChar3 = "abcd"; //直接指向常量区的字符串
```

```

9      int* ptr1 = (int*)malloc(sizeof(int) * 4);
10     int* ptr2 = (int*)calloc(4, sizeof(int));
11     int* ptr3 = (int*)realloc(ptr2, sizeof(int) * 4);
12     free(ptr1);
13     free(ptr3);
14 }

```

这道题最容易出错的地方在于 char2 和 pChar3，如下图“abcd\0”是位于常量区的字符串，前者的意义是把该字符串拷贝一份给char2数组，后者的意义是 pChar3 这个指针指向常量区的字符串



C++中的动态内存管理

基本操作

```

1  int *p1 = (int*)malloc(sizeof(int));
2  int *p2 = new int;
3  int *p3 = new int[5]; //申请5个int的数组
4  int *p4 = new int(5); //申请1个int对象，初始化为5
5  //C++11支持用初始化列表初始化数组
6  int *p5 = new int[5]{1,2,3,4,5};
7
8  delete p2;
9  delete[] p3; //括号要匹配

```

- new/delete 不是函数，而是操作符，C++没有和calloc和realloc对应的操作符

- new/delete操作内置类型：new/delete跟malloc/free没有本质的区别，只有用法的区别，new/delete简化了一些和注意操作符匹配
- new/delete操作自定义类型：**new/delete 是为自定义类型准备的。不仅在堆上申请出来，还会调用构造函数初始化和析构函数清理。**若使用malloc那么即使是对自定义类型的初始化都很难，比如类成员是私有的不提供接口获取怎么办呢？
- 注意 new/delete 和 new[]/delete[] 匹配使用，否则可能就会出问题
- malloc 失败返回 NULL，每次都要检查返回值防止野指针；new 失败时抛异常 std::bad_alloc，不需要检查返回值，但是要throw或者try catch

```

1  try {
2      char *p = new char[100];
3  }
4  catch (const exception& e) {
5      // ...
6  }

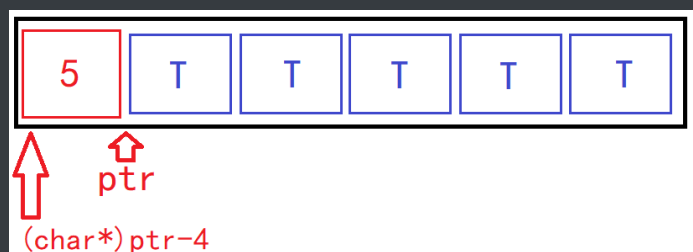
```

□ 不匹配错误的底层原因

注意：对于内存的处理与编译器类型有很大的关系，下面的情况仅针对VS系列

内置类型因为没有构造和析构函数，所以影响没有那么大，但对于自定义类型影响就大了

由于语法设计问题，new T[5] 给了要new的元素个数，这告诉编译器要对于T类元素调用1次malloc+5次构造，但是 delete[] 就没有告诉编译器到底应该要调用多少次析构+1次free



为了解决这个问题，VS系列采取的方法是在多new一个int的空间，即在new出来的空间头上记录有几个元素，这就产生了指针偏移问题，因为返回的指针实际上是真正的头指针+4

因此若是 delete[] 就会自动进行指针偏移，此时如果不匹配的使用 delete 就会取到错误的头指针，因为实际的头指针为 (char*)ptr-4，编译器就会报错

new与delete操作符的底层原理

实现原理

- new操作符 = operator new() 全局函数申请空间 + 调构造函数
- new T[N] = operator new[](N) 全局函数申请N个对象空间 + 调N次构造函数
- delete操作符 = 调析构函数清理对象中的资源 + operator delete 全局函数
- delete T[N] = 调N次析构函数清理N个对象空间中的资源 + N次 operator delete[](N) 全局函数回收空间

operator new() 与 operator delete() 全局函数

operator new() 与 operator delete() 是系统提供的全局函数（operator new() 不是new的运算符重载，这个函数的名字就叫它），new 在底层调用 operator new() 全局函数（注意不是运算符重载）来申请空间，delete 在底层通过 operator delete() 全局函数来释放空间

- operator new() 源代码

```
1 void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc) {
2     // try to allocate size bytes
3     void *p;
4     while ((p = malloc(size)) == 0) { //调malloc
5         if (_callnewh(size) == 0) {
6             // report no memory
7             // 如果申请内存失败了，这里会抛出bad_alloc 类型异常
8             static const std::bad_alloc nomem;
9             _RAISE(nomem);
10        }
11    }
12    return (p);
13 }
```

- 该函数实际通过malloc来申请空间，当malloc申请空间成功时直接返回
- 封装malloc的原因是若申请空间失败，符合C++ new的失败机制：抛异常

- 可以直接使用这个全局函数 `char *p = (char*)operator new(100);`，当然实际中是不会这么用的
- `operator delete()` 源代码

```
1  #define free(p) _free_dbg(p, _NORMAL_BLOCK) //free的实现, 封装free宏
2  void operator delete(void *pUserData) {
3      _CrtMemBlockHeader * pHead;
4      RTCCALLBACK(_RTC_Free_hook, (pUserData, 0));
5      if (pUserData == NULL)
6          return;
7      _mlock(_HEAP_LOCK); /* block other threads */
8      __TRY
9          /* get a pointer to memory block header */
10         pHead = pHdr(pUserData);
11         /* verify block type */
12         _ASSERT(_BLOCK_TYPE_IS_VALID(pHead->nBlockUse));
13         _free_dbg( pUserData, pHead->nBlockUse ); //_free_dbg就是
           free的实现
14     __FINALLY
15         _munlock(_HEAP_LOCK); /* release other threads */
16     __END_TRY_FINALLY
17     return;
18 }
```

`operator delete()`：该函数最终是通过`free`来释放空间的。实际上是可以直接`free`的，但为了和 `operator new()` 凑一对，又进行了一次封装

重载 `operator new()` 与 `operator delete()`

一般不需要重载，除非在申请和释放空间的时候有某些特殊的需求，比如

- 打印日志信息
- 重载一个类专属的 `operator new()`：利用STL容器专属的空间配置器（容器专用的内存池），当然实际STL的源码封装和下面的思路是不同的，可以看上面的封装


```

1 struct ListNode {
2     int _val;
3     ListNode* _next;
4     static allocator<ListNode> _alloc; //所有类对象共享一个空间配置器
5     void* operator new(size_t n) { //重载类专属的operator new()
6         void* obj = _alloc.allocate(1);
7         return obj;
8     }
9
10    void operator delete(void* ptr) {
11        _alloc.deallocate(ptr, 1);
12    }
13 }

```

每个类可以实现自己专属的 operator new，这时候new就不会去调库里的operator new了

定位new表达式 (placement-new)

使用格式： new(place_address)type 或 new(place_address)type(initializer-list)

定位new表达式是在**已经分配好的内存空间**中再调用构造函数初始化一个对象，因为malloc开出了空间再去通过 -> 是取不到私有类成员的

使用场景：因为new出来的已经调用构造初始化过了，所以**定位new是专门给malloc用的**，要是直接用一般的new的话直接给初始值就行了也不会有没有初始化的问题。**一般都会配合内存池进行使用**，因为内存池都是通过malloc向系统系统申请的，因此内存池分配出的内存是没有初始化过的，所以若是自定义类型对象，需要使用定位new

```

1 class A {};
2 A *p1 = (A*)malloc(sizeof(A)); //空间已经开好了!
3 if (p1 == nullptr) {
4     perror("malloc fail");
5 }
6 new(p1)A(10); //初始化为10

```

内存泄漏

什么是内存泄漏 memory leak

在main函数退出后，所有的栈、堆空间都会被OS回收，所以内存泄漏不是指分配的内存没了。而是指当进程长期运行时，比如服务器、应用软件等失去了对分配内存的控制，也就是丢失了某块内存的指针

特别是在某一个循环中不断泄漏内存后，因为OS会认为这块内容一直都在被用户使用，所以就会造成OS的可用内存越来越少，最终造成进程甚至是整台主机的宕机

内存泄漏分类

- 堆内存泄漏：程序的设计错误导致在某个局部调用的函数中通过 malloc 或 new 得到的动态开辟内存没有被释放，并且丢失了这块内容的指针入口
- 系统资源泄漏：指OS为程序分配的系统资源，如socket或者其他fd、管道等没有使用对应的函数释放掉，导致系统资源的浪费，严重可导致系统效能减少，系统执行不稳定。这种资源泄漏是无法通过关闭进程来重置的，某些时候只能通过重启系统

如何检测内存泄漏

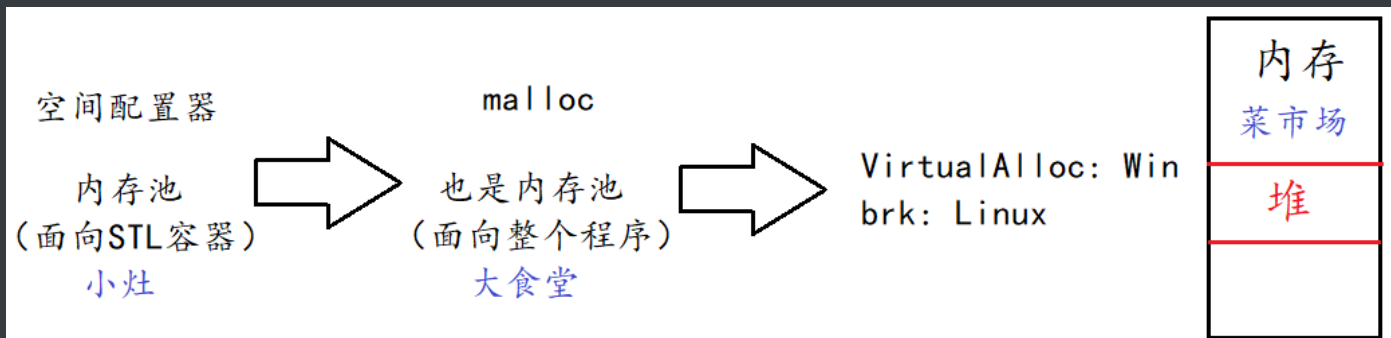
检测工具内部原理：申请内存时用一个容器记录下来，释放内存时，从容器中删除掉

valgrind

如何避免内存泄漏

SGI-STL空间配置器

intro to Allocator

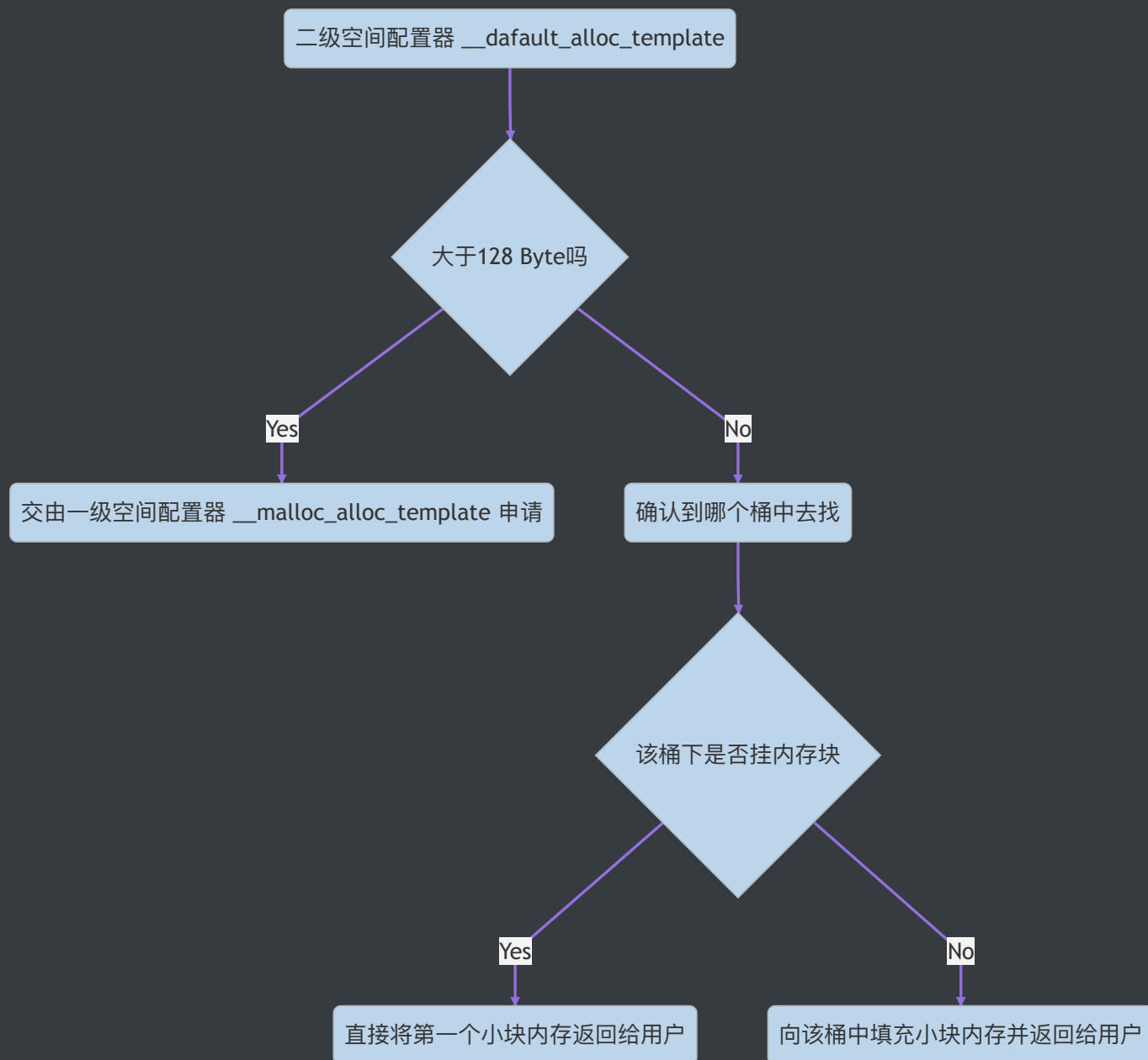


STL容器需要频繁申请释放内存，若每次都要调用malloc进行申请会有以下缺点

- 空间申请与释放需要用户自己管理，容易造成内存泄漏
- 频繁向系统申请内存块，容易造成内存碎片也影响程序运行效率
 - 外碎片问题：频繁向系统申请小块内存，有足够内存但是不连续，无法申请大块内存
 - 内碎片问题：内存块挂起来管理，由于按照一定规则对齐，就会导致内存池中的内存碎片
- 直接使用malloc与new进行申请，每块空间有额外空间浪费，因为要记录开的内存空间的相关信息，如大小等
- 代码结构不清晰
- 未考虑线程安全

因此需要设计高效的内存管理机制，空间配置并不是什么特殊的数据结构，只是对malloc的深度封装以提高它的效率

一级空间配置器



`std::allocator` 被设计为两层：一级空间配置器 `__malloc_alloc_template` 和二级空间配置器 `default_alloc_template`

一级空间配置器是二级空间配置器当申请空间大于128 Byte时的特例，而一级空间配置器就是直接用 `__malloc_alloc_template` 封装了 `malloc`和`free`，二层空间配置器的底层则是对 `malloc`和`free`的多次封装

```
1 static void *allocate(size_t n) {
2     void *result = malloc(n);
3     if (0 == result) result = oom_malloc(n);
4     return result;
5 }
6
7 static void deallocate(void *p, size_t /* n */) {
8     free(p);
9 }
```

SGL-STL默认选择使用一级还是二级空间配置器，通过 `USE_MALLOC` 宏进行控制

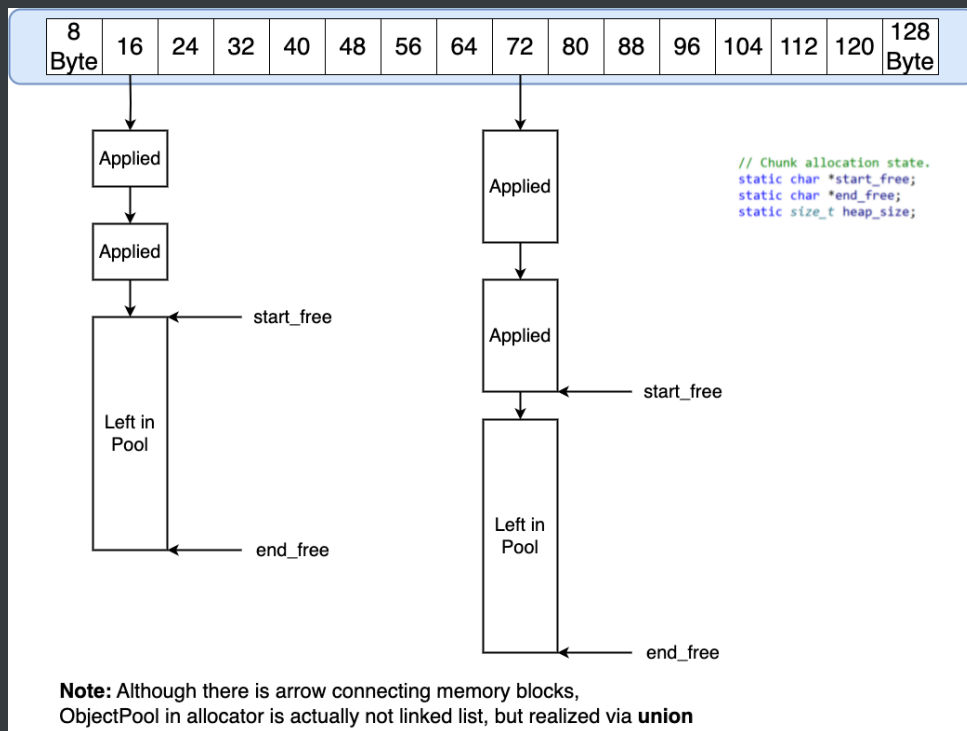
二级空间配置器设计

二级空间配置器采用哈希桶上对应的自由链表来管理内存池的方式来管理内存

因为已经是小于128字节的内存才会使用二级空间配置器，因此如果继续用1字节进行切割的话那么小内存就太碎了，而且用户申请的时候的内存基本以4字节的倍数为主，其他大小的空间几乎很少用到，因此STL中的设计是将内存对齐到8字节，也就是说128个字节内存有16个桶。选对齐到8字节也是因为每个内存块里都要存一个union，至少要有个存下指针的8字节空间

比较有特色的是自由链表中小块内存的连接方式是用**联合体 union**来实现的（`tcmalloc`中采用的都是直接存下一个内存块的地址）。这个联合体的设计还是很巧妙的，因为当把这个内存分出去的时候，前面原来写地址的部分可以被覆盖了，因为我们已经不会用这个地址去找自由链表上的下一个小内存块了（连接关系已经重新建立了）。而这块小内存用完之后，还给自由链表的时候，里面的用户数据也不需要了，又可以重新写地址了

这种哈希桶上自由链表的头插头删都是 $O(1)$ 的，效率相比于每次都要去跟系统要高太多了



内存池 Memory Pool 就是先申请一块比较大的内存块作为备用。当需要内存时，直接取内存池中去取。首位两个指针相当于是限制这个蓄水池的闸门。若池中空间不够了，就继续向内存中malloc取。若用户不使用了就可以直接交还给内存中，内存池可以位之后需要的用户重新派发这些回收的内存

那么这块内存池该采取怎么样的数据结构进行管理呢？考虑到主要问题在于归还内存的时候可能并不是按照给的时候的顺序给的，即使归还了之后也要重新将内存分派出去，那么如何知道内存块的大小呢？若将内存池设计成链表的话，需要挨个遍历效率低。因此二级空间配置器设计成了**哈希桶**的形式

注意：内存池和哈希表实现的二级空间配置器是两个结构，空间配置器是一个管理结构，它会去找内存池要，内存池可以通过malloc向内存要资源从而不断扩展，相当于 end_free 这个闸门一直在外后走。在同一个进程中所有的容器共享一个空间配置器，因此**STL空间配置器是用单例模式 Singleton Pattern设计的**（虽然源码里并不是完全的单例，但是全部属性都用了static所以页差不多），不同的容器之间只要都是用了空间配置器，那么他们之间就可以互相调用内存块

SGL-STL二级空间配置的空间申请与回收

■ 前期准备

用联合体来维护哈希桶结构：这块内存空间是两用的，申请了之后所有的都可以用，但如果只是挂在哈希桶上，那么需要用头4个或头8个字节来存下一个结点的地址

```

1 union obj {
2     union obj * free_list_link;
3     char client_data[1]; /* The client sees this. */
4 };

```

■ 申请空间

- 先去对应size的哈希桶要切好的内存块，如果没有就继续往后面的大哈希桶 x 要，如果有的话就切size大小的给用户，然后把切剩余的空间 x-size 挂到相应的哈希桶的自由链表上
- 若还是找不到空的内存块就要向内存池中索要空间了：每次直接向堆去申请都会给20个切好的小内存，返回头上那个给用户，然后将多余的19个小内存块全部挂到对应的哈希桶的自由链表上

■ 空间回收

```

1 // 函数功能：用户将空间归还给空间配置器
2 // 参数：p空间首地址 n空间总大小
3 static void deallocate(void *p, size_t n) {
4     obj *q = (obj *)p;
5     obj ** my_free_list;
6     // 如果空间不是小块内存，交给一级空间配置器回收
7     if (n > (size_t) __MAX_BYTES) {
8         malloc_alloc::deallocate(p, n);
9         return;
10    }
11    // 找到对应的哈希桶，将内存挂在哈希桶中（头插）
12    my_free_list = free_list + FREELIST_INDEX(n); //找到对应桶的序号
13    q->free_list_link = *my_free_list; //头插
14    *my_free_list = q;
15 }

```

空间配置器的二次封装

每个容器的模版参数都有一个空间配置器，默认的是 `std::allocator`，但也可以用用户自己的，只要满足空间配置器的API要求，比如`allocate`、`deallocate`等

下面是源码：以stl_list容器为例，STL中的容器都遵循类似的封装步骤。封装了一个专门针对list_node 的申请类，这样就不需要显式传 size 了

之后 create_node() 的 construct 封装的是定位new，用来给申请到的内存赋值

```
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
typedef __default_alloc_template<false, 0> single_client_alloc;

template <class T, class Alloc = alloc>
class list {
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node;
    typedef simple_alloc<list_node, Alloc> list_node_allocator;
protected:
    link_type get_node() { return list_node_allocator::allocate(); }
    void put_node(link_type p) { list_node_allocator::deallocate(p); }

    link_type create_node(const T& x) {
        link_type p = get_node();
        __STL_TRY {
            construct(&p->data, x);
        }
        __STL_UNWIND(put_node(p));
        return p;
    }
    void destroy_node(link_type p) {
        destroy(&p->data);
        put_node(p);
    }
};
```

二级空间配置器

封装一个专门针对list_node申请类

```
template<class T, class Alloc>
class simple_alloc {
public:
    static T *allocate(size_t n)
    { return 0 == n? 0 : (T*) Alloc::allocate(n * sizeof (T)); }
    static T *allocate(void)
    { return (T*) Alloc::allocate(sizeof (T)); }
    static void deallocate(T *p, size_t n)
    { if (0 != n) Alloc::deallocate(p, n * sizeof (T)); }
    static void deallocate(T *p)
    { Alloc::deallocate(p, sizeof (T)); }
};
```

申请空间

释放空间

范型编程与范型算法

泛型编程 *Generic Programming*

泛型编程：编写与类型无关的通用代码，是代码复用的一种手段，模板是泛型编程的基础

函数模板

函数模板概念

- 函数模板代表了一个函数家族，该函数模板与类型无关，在使用时被参数化，根据实参类型产生函数的特定类型版本
- 模板不支持分离编译，统一放在头文件里，但在同一个文件里可以分离编译


```
1 // 模板参数定义了模板类型，类似函数参数，但函数参数定义了参数对象
2 // typename后面类型名字T是随便取的，一般是大写字母或者单词首字母大写 T、Ty、K、V，
   代表了一个模拟类型/虚拟类型
3 template<typename T>
4 void Swap(T& left, T& right) {
5     T temp = left;
6     left = right;
7     right = temp;
8 }
9 // swap在std中有模板定义了
```

函数模板的实例化

函数模板的原理：类型推演 Type Deduction和函数模板实例化 Instantiation

函数模板是一个设计图，他本来并不是真正用来执行任务的函数，每次调用相关函数编译器都要根据函数模板进行一次类型推演产生相关函数

- 隐式实例化：让编译器根据实参**自动推演**模板参数的实际类型。模板中不会进行隐式类型转换，编译器不知道该转成哪一个会报错，需要用户自己进行强制类型转换
- 显式实例化：在函数名后的 `<>` 中由用户指定模板参数的实际类型
 - 参数需要强转时可以使用

```
1 // Add(1.1, 2) // 报错
2 Add((int)1.1, 2); // 用户强转输入
3 Add<int>(1.1, 2); // 模板的显式实例化
```

- 需要指定返回类型时一定要显式实例化

```

1  // 已经定义了一个A类
2  template<class T>
3  T* Func(int n) {
4      T* a = new T[n];
5      return a;
6  }
7  // 不能自动推演返回类型T
8  Func<A>(10); // 显式实例化

```

非类型模板参数 Nontype

```

1  template<class T, size_t N = 10>
2  class array {
3  public:
4  private:
5      T _a[N];
6  };

```

- 非类型模板参数只能用于整形、指针或引用，浮点数、类对象以及字符串是不允许作为非类型模板参数的
- 非类型的模板参数必须在编译期就能确认结果

模版编译

模版要定义在头文件里

当编译器遇到一个模版定义时，它不会生成代码，只有当实例化模版的时候编译器才会生成代码。这种特性会影响到如何组织代码以及错误何时才会被检测到

若在本模块中找不到函数和变量的地址（即定义），那么会在链接阶段去总符号表里找，所以函数和变量只要在头文件中有一个声明就可以，定义可以放到某个模块的c文件中。但是类和模版不同，它们是在编译阶段实例化数据的“蓝图”，每一个用到它们的文件都必须知道它们的定义，而不仅仅是声明。所以完整的类和模版必须要定义在头文件中

涉及到模版的报错往往非常复杂，非常难以debug。这也是使用模版的一个缺点

模板不能进行分离编译，否则会因为两次编译导致链接错误

依赖名字与可见性

非依赖名字 non-dependent name 和 **依赖名字 dependent name** 是模板编程中的两个重要概念。其中，非依赖名字是指不依赖于模板参数的名字，例如全局变量、函数名、类型名等，而依赖名字则是指依赖于模板参数的名字，例如模板参数名、模板内部定义的类型名等

在C++模板的实例化过程中，编译器需要在模板定义时和模板实例化时分别对非依赖名字和依赖名字进行名称查找

- 对于非依赖名字，编译器必须在模板定义时就确定其含义，因此这些名字必须是可见的
- 对于依赖名字，编译器只有在模板实例化时才能确定其含义，因此这些名字可以在模板定义时不可见，但必须在模板实例化时可见

声明的**可见性 Visibility**指的是在哪些作用域内可以访问该声明。一般来说，声明在其所在的作用域内是可见的，即在该作用域内可以使用该声明。若声明是在一个内层作用域中，而其外层作用域中已经存在同名的声明，那么该内层作用域中的声明会隐藏外层作用域中的同名声明，此时外层作用域中的声明不可见

```
1  template <typename T>
2  void foo(T x) {
3      int y = 10; // 可见的非依赖名字
4      x += y;     // 可见的依赖名字
5      // ...
6  }
7
8  int main() {
9      int x = 5;
10     foo(x);
11     return 0;
12 }
```

在这个例子中，`foo` 是一个函数模板，接受一个类型为 `T` 的参数 `x`。在 `foo` 函数模板内部，我们定义了一个整数变量 `y`，它是一个可见的非依赖名字。同时，`x` 是一个依赖名字，它依赖于模板参数 `T`。在 `main` 函数中，我们定义了一个整数变量 `x`，然后调用了 `foo` 函数模板，将变量 `x` 作为参数传递进去

```
1  template <typename T>
2  void foo(T x) {
3      bar(x);  // 不可见的非依赖函数名
4  }
5
6  int main() {
7      int x = 5;
8      foo(x);
9      void bar(int);  // 定义了一个非依赖函数bar，但在foo模板定义时不可见
10     return 0;
11 }
```

在这个例子中，`foo` 是一个函数模板，接受一个类型为 `T` 的参数 `x`。在 `foo` 函数模板内部，我们尝试调用一个名为 `bar` 的非依赖函数。然而，在 `main` 函数中定义的 `bar` 函数虽然与 `foo` 调用的 `bar` 同名，但是在 `foo` 模板定义时并不可见。这样的情况下，在编译时编译器无法找到可用的 `bar` 函数定义，导致编译错误

类模板

类模板的定义格式

```
1  // template<typename T>
2  template<class T> // 上下两种类模板都可以
3  class Stack {
4      //...
5  private:
6      T* _a;
7  };
```

类模板的实例化

和函数模板的自动推演不同，类模板需要显式实例化，也就是说需要显式给出 `<>` 里面的类型，这些类型被称为显式模板实参 `explicit template argument` 列表，它们会被绑定到模板参数

类模板名字不是真正的类，实例化的结果 `Stack<int>` 才是真正的类，若不显式实例化，编译器无法判断

类模板的特化 *Specialization* -- 针对某些类型进行特殊化处理

函数模板特化

```
1 // 函数模板
2 template<typename T>
3 bool Greater(T left, T right) {
4     return left > right;
5 }
6 // 若输入的参数是指针，就不会进行实例化，而是直接进入下面的特化
7 template<>
8 bool Greater<Date*>(Date* left, Date* right) {
9     return *left > *right;
10 }
```

- 必须要现有一个基础的函数模板
- 关键字 `template` 后面接一对空的尖括号 `<>`
- 函数名后跟一对 `<>`，尖括号中指定需要特化的类型
- 必须要使用模板参数关键字 `typename` 或 `class`，这两者在定义模板是等价的，不能将 `class` 换成 `struct`

类模板特化

- 正常模板

```
1  template<class T1, class T2>
2  class Data{};
```

- 全特化

```
1  template<>
2  class Data<int, char>{};
```

- 偏特化

- 部分特化：将模板参数类表中的一部分参数特化

```
1  template<class T1>
2  class Data<T1, int>{};
3
4  //调用
5  Data<int, int>; // 此时会进入特化，因为第二个参数类型匹配
```

- 参数更进一步的限制

```
1  template<class T1, class T2>
2  class Data<T1*, T2*>{};
3
4  template<class T1&, class T2&>
5  class Data<T1&, T2&>{};
```

模板参数的匹配原则

- 一个非模板函数可以和一个同名的函数模板同时存在，而且该函数模板还可以被实例化为这个非模板函数
- 匹配的非模板参数函数/类 > 最匹配、最特化 > 一般模板产生实例

例题

- 下列的模板声明中，其中几个是正确的

```
1  1)template // 错误
2  2)template<T1,T2> // 错误，缺少模板参数关键字 typename或class
3  3)template<class T1,T2> // 错误，缺少模板参数关键字 typename或class
4  4)template<class T1,class T2> // 正确
5  5)template<typename T1,T2> // 错误，缺少模板参数关键字 typename或class
6  6)template<typename T1,typename T2> // 正确
7  7)template<class T1,typename T2> // 正确
8  8)<typename T1,class T2> // 错误，没有template关键字
9  9)template<typeaname T1, typename T2, size_t N> // 正确
10 10)template<typeeaname T, size_t N=100, class _A=alloc<T>> // 正确
11 11)template<size_t N> // 正确
```

- 以下程序的输出结果为

```
1  template<typename Type>
2  Type Max(const Type &a, const Type &b) {
3      cout<<"This is Max<Type>"<<endl;
4      return a > b ? a : b;
5  }
6  template<>
7  int Max<int>(const int &a, const int &b) {
8      cout<<"This is Max<int>"<<endl;
9      return a > b ? a : b;
10 }
11 template<>
12 char Max<char>(const char &a, const char &b) {
13     cout<<"This is Max<char>"<<endl;
14     return a > b ? a : b;
15 }
16 int Max(const int &a, const int &b) {
17     cout<<"This is Max"<<endl;
18     return a > b ? a : b;
19 }
```

```

20 int main() {
21     Max(10,20); // "This is Max" // 可以直接匹配非模板参数，虽然形参是引用
                // 和权限缩小，但非模板参数仍然是最匹配的
22     Max(12.34,23.45); // "This is Max<Type>" // 没有匹配的特化模板或显式
                // 实例化的实例，因此要调用模板
23     Max('A','B'); // "This is Max<char>" // 已经给出了特化模板，直接匹配
24     Max<int>(20,30); // "This is Max<int>" 直接进行了显式实例化，但因为给
                // 出了int条件下的特化，所以使用特化模板
25     return 0;
26 }

```

- 以下程序运行结果为

```

1  template<class T1, class T2>
2  class Data {
3  public:
4      Data() { cout << "Data<T1, T2>" << endl; }
5  private:
6      T1 _d1;
7      T2 _d2;
8  };
9  template <class T1>
10 class Data<T1, int> {
11 public:
12     Data() { cout << "Data<T1, int>" << endl; }
13 private:
14     T1 _d1;
15     int _d2;
16 };
17 template <typename T1, typename T2>
18 class Data <T1*,T2*> {
19 public:
20     Data() { cout << "Data<T1*, T2*>" << endl; }
21 private:
22     T1 _d1;
23     T2 _d2;

```



```

24 };
25 template <typename T1, typename T2>
26 class Data <T1&, T2&> {
27 public:
28     Data(const T1& d1, const T2& d2)
29         : _d1(d1)
30         , _d2(d2)
31         {cout << "Data<T1&, T2&>" << endl;}
32 private:
33     const T1 & _d1;
34     const T2 & _d2;
35 };
36 int main() {
37     Data<double, int> d1; // "Data<T1, int>" 第二个参数int已经最匹配了
    class Data<T1, int>
38     Data<int, double> d2; // "Data<T1, T2>" 没有最匹配的特化模板，需要进行模板实例化
39     Data<int*, int*> d3; // "Data<T1*, T2*>"
40     Data<int&, int&> d4(1, 2); // "Data<T1&, T2&>"
41     return 0;
42 }

```

模板的优缺点

■ 优点

- 模板复用了代码，节省资源，更快的迭代开发，STL库的诞生很大程度上得益于此
- 增强了代码的灵活性

■ 缺点

- 模板只是将重复的代码交给编译器实现，因此模板也会导致代码膨胀问题，从而导致编译时间变长
- 出现模板编译错误时，错误信息非常凌乱，不易定位错误

STL

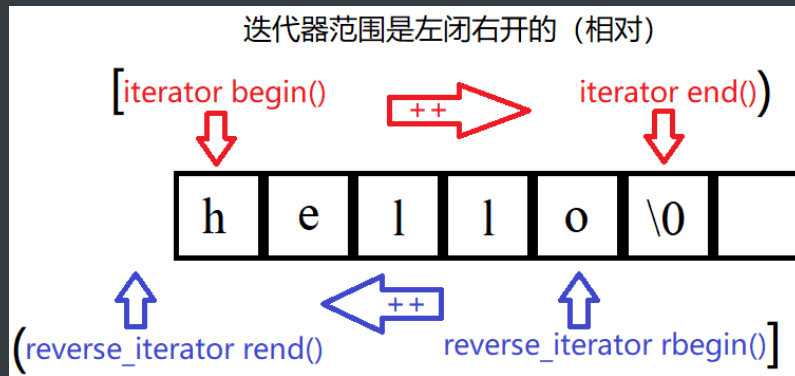
Standard Template Library 标准模板库是C++标准库 `std` 的重要组成部分，其不仅是一个可复用的组件库，而且是一个包含数据结构的软件框架

STL版本

原始HP版本 -> PJ版本 -> RW版本 -> SGI版本（主要被使用的版本）

STL六大组件

- Container 容器：容器就是各种常用的数据结构用C++实现，C++可以提供但C语言不能提供的原因主要是得益于C++提供了模板这种泛型编程方式
 - 序列式容器 Sequential container：底层为线性序列的数据结构，里面存储的是元素本身
 - `string`
 - `vector`
 - `list`
 - `deque`
 - 关联式容器 Associative container：存储的是 `<key, value>` 结构的键值对
 - `map`
 - `set`
 - `multimap`
 - `multiset`
 - c++11新增了`array`静态数组容器，和普通数组的主要区别在于对越界的检查更加严格，因为 `array[]` 的本质是函数调用 -- 运算符重载
- Iterator 迭代器：迭代器再不暴露底层实现细节的情况下，提供了统一的方式（即从上层角度看，行为和指针一样）去访问容器。屏蔽底层实现细节，体现了封装的价值和力量。迭代器被认为是`algorithm`和`container`的粘合剂，因为`algorithm`要通过迭代器来影响`container`



- iterator
- const_iterator
- reverse_iterator
- const_reverse_iterator
- Functor 仿函数
 - greater
 - less ...
- Algorithm 算法
 - find
 - swap
 - reverse
 - sort
 - merge ...
- Allocator 空间配置器
- Adapter 适配器/配接器：对容器进行复用
 - stack
 - queue
 - priority_queue

STL缺陷

- 更新维护满
- 不支持线性安全你，在并发环境下需要自己加锁，且锁的粒度比较大

- 极度追求效率，导致内部比较复杂，如类型萃取，迭代器萃取
- STL的使用会有代码膨胀的问题

常用 `<algorithm>` 库函数笔记

封装了容器通用的函数模板

- `find`

Returns an iterator to the first element in the range `[first,last)` that compares equal to `val`. If no such element is found, the function returns `last`.
- `reverse`: 注意`reverse`范围左闭右开不包括`last`

Reverses the order of the elements in the range `[first,last)`.
- `sort`
 - `sort`要使用随机迭代器，也就是`vector`那种支持随机访问的迭代器
 - `sort`的底层是不稳定快排，若要保证稳定性需要用 `stable_sort`

常用容器对比

容器	<code>vector</code>	<code>list</code>	<code>deque</code>	<code>set</code>	<code>multiset</code>	<code>map</code>	<code>multimap</code>
元素类型	值	值	值	值	值	KV	KV
可重复?	✓	✗					
可随机存取?							
迭代器类型							
元素搜索速度							
快速插入							
释放							
允许 <code>reserve</code> ?							

string

string是一个特殊的顺序表，与顺序表的不同之处是它的操作对象是字符，并且最后一个数据默认是\0

标准库中的string常用接口

string类介绍

std::string . Strings are objects that represent sequences of characters. String 是一xi个char类型的动态增长数组

常见构造

string的构造函数是添加了 \0 的，满足C语言字符串规范

std::string::string <string>

C++98 C++11 ?

来源: <https://cplusplus.com/reference/string/string/string/>

- ★ default (1) string();
- ★ copy (2) string (const string& str);
- substring (3) string (const string& str, size_t pos, size_t len = npos);
- ★ from c-string (4) string (const char* s);
- from sequence (5) string (const char* s, size_t n);
- fill (6) string (size_t n, char c);
- range (7) template <class InputIterator> string (InputIterator first, InputIterator last);

- 默认初始化空串 default
- 使用 () call构造函数通过C字符串直接初始化 direct initialization
- 使用 = 拷贝初始化 copy initializaiton

容量操作

- size 与 length 是一样的，只是因为string类产生的比较早，用的是 length ，后来出现了其他容器后统一规范为 size
- string.size() 返回的是一个无符号的 string::size_type 类型的数据，所以当求了size之后，就不要和int混用了以免出错

- `reserve(size_t n)` : 预留空间
- `clear` : 清空有效字符
- `resize(size_t n, char c)` : 将有效字符的个数改成n个, 多出的空间用字符c填充 (开空间+初始化)

访问及遍历操作

- `operator[]` 重载的意义: 可以像操作数组一样用 `[]` 去读写访问对象
- `at` 和 `operator[]` 的区别是越界以后抛异常

迭代器 iterator: 范围for的底层实现, 范围for的底层就是无脑直接替换迭代器

- iterator是属于容器的一个类, 它的用法很像指针, 可能是指针也可能不是
- string和vector不太使用迭代器, 因为 `operator[]` 更方便, 但是对于其他的容器如list、map、set来说只能用迭代器来访问。迭代器是所有容器的通用访问方式, 用法类似
- C++中的迭代区间都是左闭右开 `[)`, 右开是因为方便遍历到 `\0` 时正好结束
- `begin & end` : return iterator to beginning/end
- `rbegin & rend` : return reverse iterator
- 四种迭代器
 - `iterator/reverse_iterator`
 - `const_iterator/const_reverse_iterator`

```

1 // 迭代器示例
2 void PrintString(const string& str) {
3     string::const_iterator it = str.begin();
4     //auto it = str.begin(); // auto自动推导
5     while (it != str.end())
6         /*it = 'x';
7         cout << *it++ << " ";
8         cout << endl;
9
10    string::const_reverse_iterator rit = str.rbegin();
11    while (rit != str.rend()
```

```
12         cout << *rit++ << " ";
13     cout << endl;
14 }
```

修改操作

- `operator+=` 在字符串后追加字符串`str`
- `push_back` 和 `append`
 - 不如 `+=`，基本都是用 `+=`
 - 也可以使用迭代器，来控制插入字符的某一段

```
1 string s("hello");
2 string str(" world");
3 s.append(str.begin(), str.end());
```

- `insert` 效率很低
- `find` 和 `rfind`：返回 `size_t`，返回第一个/最后一个找到的位置，找不到则返回 `npos`，这时会解释为一个非常大的数，这是因为默认不存在这么大的字符串
- `substr` 从`pos`位置开始取`n`个字节长度的子串
- `c_str`：用来兼容C字符串，C不支持`string`类型

```
1 string filename("test.cpp");
2 FILE* fout = fopen(filename.c_str(), "r");
3 assert(fout);
4 // ...
```

非成员函数

- `getline`：`cin`和`scanf`一样，输入以空格或换行符为间隔标志，因此会自动把空格去掉。`getline`和C语言中的`fgetf`函数功能一样，帮助取到一句有空格的字符串
- `operator+`：尽量少用，传值返回导致深拷贝效率低

string 和其他数据类型之间的转换

- atoi
- stoi, stol, stof, stod, ...
- to_sring, to_wstring

string模拟实现中构造函数的问题

错误写法一

```
1  class String {
2  public:
3      String(const char* str)
4          :_str(str) // 给的是const char* 常量字符串，会报错
5      {}
6  private:
7      char* _str; // 因为_str是要提供接口被修改的，所以不能是const
8      size_t _size;
9      size_t _capacity;
10 }
```

直接给常量字符串会出很多问题，因此考虑开辟空间后复制

错误写法二：给nullptr


```

1  String() // 提供一个空的
2      :_str(nullptr)
3      , _size(0)
4      , _capacity(0) // _capacity不包括\0
5  {}
6  const char* c_str() const {
7      return _str;
8  }
9
10 // 调用c_str接口
11 String s1;
12 cout << s1.c_str() << endl;

```

若空串给的是nullptr，调用的时候会发生对空指针解引用问题

正确写法一

```

1  String() // 提供一个空的
2      :_str(new char[1]) // 多new一个\0
3      , _size(0)
4      , _capacity(0) // _capacity不包括\0
5  {
6      _str[0] = '\0';
7  }
8  String(const char* str)
9      :_str(new char[strlen(str) + 1]) // 多new一个\0
10     , _size(strlen(str))
11     , _capacity(strlen(str)) // _capacity不包括\0
12  {
13      strcpy(_str, str);
14  }

```

正确写法二：全缺省参数

```
1 // 全缺省参数同样不能给nullptr, strlen直接对nullptr解引用了
2 String(const char* str="") //隐含了一个\0
3 // String(const char* str="\0") // 其实这样给编译器还会再给一个\0, 也就是\0\0
4     :_str(new char[strlen(str) + 1]) // 多new一个\0
5     , _size(strlen(str))
6     , _capacity(strlen(str)) // _capacity不包括\0
7 {
8     strcpy(_str, str);
9 }
```

strlen()是一个O(N)的函数, 可以进一步改造

错误写法三

```
1 class String {
2     String(const char* str="")
3         :_size(strlen(str))
4         , _capacity(_size) // _capacity不包括\0
5         , _str(new char[_capacity + 1]) // 多new一个\0
6     {
7         strcpy(_str, str);
8     }
9 private:
10     char* _str;
11     size_t _size;
12     size_t _capacity;
13 }
```

这么写是错误的, 因为初始化顺序是按照声明的顺序来, 若这么写初始化列表size, capacity都是随机值。若更改声明的顺序会产生很多的维护问题

正确写法三

因为类成员都是内置类型，所以不使用初始化列表以避免初始化顺序依赖问题

```
1  class string {
2  public:
3      string(const char* str="") {
4          _size = strlen(str);
5          _capacity = _size;
6          _str = new char[_capacity + 1];
7          strcpy(_str, str);
8      }
9      ~string() {
10         delete[] _str;
11         _str = nullptr;
12         _size = _capacity = 0;
13     }
14 private:
15     char* _str;
16     size_t _size;
17     size_t _capacity;
18 }
```

*string*类的部分接口

size 和 operator[]

```
1  size_t size() const { // 普通对象和const对象都可以使用
2      return _size;
3  }
4
5  //const对象只读
6  const char &operator[](size_t pos) const
7  {
8      assert(pos < _size);
```

```

9     return _str[pos];
10 }
11
12 char &operator[](size_t pos) { // 返回引用 可读可写
13     assert(pos < _size); // string的遍历可以直接用 while(s[i++]), 因为
    operator[]使用了_size, 自动判断结束
14     return _str[pos];
15 }

```

string类中的迭代器就是原生指针

```

1  typedef char* iterator; // 在string中的迭代器就是对原生指针的封装
2  typedef const char* const_iterator;
3  // 左闭右开
4  iterator begin() {
5      return _str;
6  }
7  iterator end() {
8      return _str + _size;
9  }
10 const_iterator begin() const {
11     return _str;
12 }
13 const_iterator end() const {
14     return _str + _size;
15 }

```

string类的增删查改

- reserve 作用：避免最初的扩容开销、用来复用

```

1 char* my_strcpy(char* dest, const char* src);
2 void reserve(size_t n) {
3     if (n > _capacity) {
4         char* tmp = new char[n + 1];
5         strcpy(tmp, _str);
6         delete[] _str;
7
8         _str = tmp;
9         _capacity = n;
10    }
11 }

```

- push_back : 尾插一个字符

- 直接实现

```

1 void push_back(char ch) {
2     // 满了就扩容
3     if (_size == _capacity)
4         reserve(_capacity == 0 ? 4 : 2 * _capacity);
5     _str[_size] = ch;
6     _str[++_size] = '\0';
7 }

```

- insert 复用: insert(_size, ch);

- append : 尾插一个字符串

- 直接实现

```

1 void append(const char* str) {
2     size_t len = strlen(str);
3     // 满了就扩容
4     if (_size + len > _capacity)
5         reserve(_size + len);
6     strcpy(_str + _size, str);
7     // strcat(_str, str); // 不要用strcat, 因为要找尾, 效率比strcpy低
8     _size += len;
9 }

```

- insert 复用: insert(_size, str);

- operator+=

- push_back 复用

```

1 string& operator+=(char ch) {
2     push_back(ch);
3     return *this;
4 }

```

- append 复用

```

1 string& operator+=(const char* str) {
2     append(str);
3     return *this;
4 }

```

- insert

- 插入单字符

```

1 string& insert(size_t pos, char ch) {
2     assert(pos <= _size); // =的时候就是尾插
3     // 满了就扩容
4     if (_size == _capacity)
5         reserve(_capacity == 0 ? 4 : 2 * _capacity);
6     // 挪数据

```

```

7     size_t end = _size + 1;
8     while (end > pos) {
9         _str[end] = _str[end - 1];
10        --end;
11    }
12    // 插入字符
13    _str[pos] = ch;
14    ++_size;
15
16    return *this;
17 }

```

在这里会有两个坑

- 一开始的思路是如下的，将end定义为 `size_t`，但此时当pos=0时，因为end是一个无符号数，0-1会直接让end变成一个很大的数字，因此程序会陷入死循环。
- 然后考虑把end改为定义成int，但是当执行 `end>=pos` 这个运算时，由于**隐式类型转换**的存在，又会把end提升为一个无符号数，又出现了死循环。因此最后的设计是上面的

```

1 // 不好的设计1
2 // ...
3 size_t end = _size;
4 while (end >= pos) { //隐式类型转换
5     _str[end+1] = _str[end]
6     --end;
7 }

```

```

1 // 不好的设计2
2 // ...
3 int end = _size;
4 while (end >= (int)pos) {
5     _str[end+1] = _str[end]
6     --end;
7 }

```

这样设计不好，应该将位置下标设计为size_t，挪动设置为 `_str[end] = _str[end-1]` 防止越界

- 插入字符串

```
1  string& insert(size_t pos, const char* str) {
2      assert(pos <= _size);
3      size_t len = strlen(str);
4      // 满了就扩容
5      if (_size + len > _capacity)
6          reserve(_size + len);
7      // 挪数据
8      size_t end = _size + len;
9      while (end >= pos + len) {
10         _str[end] = _str[end - len];
11         --end;
12     }
13     // 插入字符串
14     strncpy(_str + pos, str, len);
15     _size += len;
16     return *this;
17 }
```

深拷贝 *Deep Copy*

浅拷贝 *Shallow Copy*

浅拷贝问题在之前的默认拷贝构造函数和默认赋值运算符重载中已经遇到过，默认拷贝构造是浅拷贝，对于str这种开辟了空间的变量，指针指向的是同一个内存空间。当析构的时候会释放同一个内存空间多次。为解决这一个问题需要用户需要手动写深拷贝，即创造两份内存空间

深拷贝传统写法


```

1  string(const string& s)
2      :_str(new char[s._capacity + 1])
3      , _size(s._size)
4      , _capacity(s._capacity)
5  {
6      strcpy(_str, s._str);
7  }

```

对于 `operator=`，不考虑dest空间是否足够容纳src，因为如果要考虑的话会有各种情况，比较复杂。直接将dest释放掉，然后给一个新new的src容量的空间tmp后再拷贝

```

1  string& operator=(const string& s) {
2      if (this != &s) { // 防止自己给自己赋值
3          char* tmp = new char[s._capacity + 1]; // 先new tmp再 delete[]
              _str 的原因是因为防止new失败反而破坏了_str
4          strcpy(tmp, s._str);
5          delete[] _str;
6          _str = tmp;
7          _size = s._size;
8          _capacity = s._capacity;
9      }
10     return *this;
11 }

```

现代写法：安排一个打工人

■ 拷贝构造的现代写法

```

1  void swap(string& tmp) {
2      std::swap(_str, tmp._str); // std::表明调用的是std库中的swap，否则编译器会优先调用局部域中的swap（也就是自己变成迭代了）
3      std::swap(_size, tmp._size);
4      std::swap(_capacity, tmp._capacity);
5  }
6

```

```

7  string(const string& s)
8      :_str(nullptr)
9      , _size(0)
10     , _capacity(0) // 不给初始值的话，tmp里的内置类型是随机值，而delete随机
                        值会崩溃
11 {
12     string tmp(s._str);
13     swap(tmp); // this->swap(tmp); 用的是自己写的当前域下的swap，优先访问
                        局部域
14 }

```

▪ operator= 的两种写法

▪ 利用tmp

```

1  string& operator=(const string& s) {
2      if (this != &s) { // 防止自己给自己赋值
3          string tmp(s);
4          swap(*this, tmp); // 这里的swap是string自己定义的swap
5      }
6      return *this;
7  }

```

▪ 传值传参，直接让s顶替tmp当打工人

```

1  string& operator=(string s) {
2      swap(s);
3      return *this;
4  }

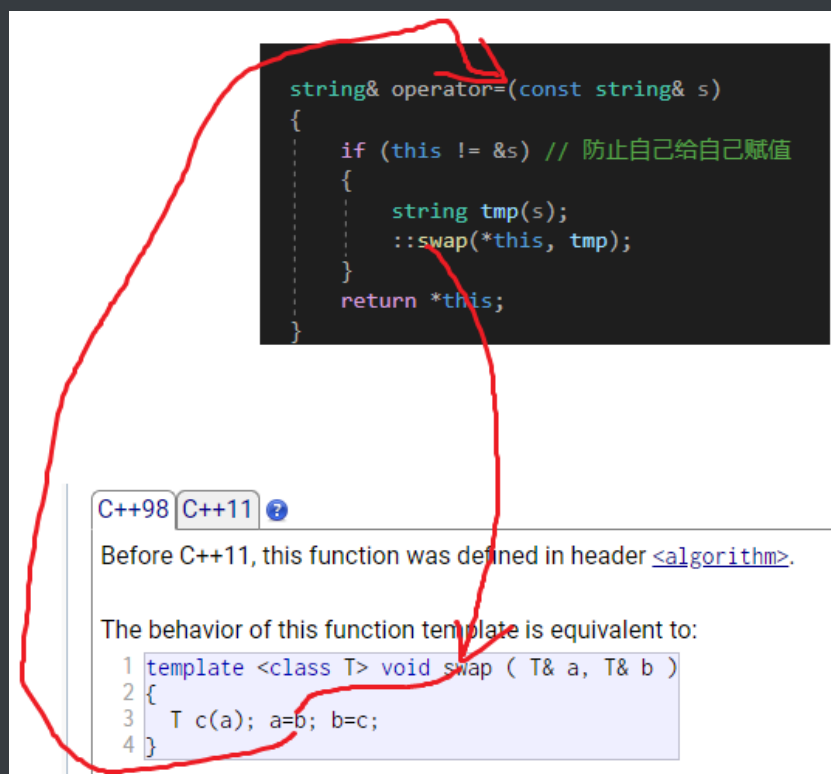
```

swap的问题

std::swap	public member function
<p>C++98 [C++11] </p> <p>// defined in <algorithm> before C++11 template <class T> void swap (T& a, T& b);</p> <p>Exchange values of two objects</p> <p>Exchanges the values of <i>a</i> and <i>b</i>.</p> <p>std标准库的swap</p> <p>C++98 [C++11] </p> <p>Before C++11, this function was defined in header <algorithm>.</p> <p>The behavior of this function template is equivalent to:</p> <pre> 1 template <class T> void swap (T& a, T& b) 2 { 3 T t(a); a=b; b=t; 4 }</pre>	<p>std::string::swap <string></p> <p>void swap (string& str);</p> <p>string类的swap</p> <p>Swap string values</p> <p>Exchanges the content of the container by the content of <i>str</i>, which is another <i>string</i> object. <i>Lengths</i> may differ.</p> <p>After the call to this member function, the value of this object is the value <i>str</i> had before the call, and the value of <i>str</i> is the value this object had before the call.</p> <p>Notice that a non-member function exists with the same name, <i>swap</i>, overloading that algorithm with an optimization that behaves like this member function.</p>

std提供的swap函数代价很高，需要进行3次拷贝（1次拷贝，2次赋值），因为需要借助中间变量。对于简单的内置变量当然无所谓，但若是一个很大的类或结构体那效率就很低了。所以可以自己写swap，通过复用库里的 `std::swap` 来交换一些开销小的内容

下面这种写法是错误的，会报栈溢出，因为 `std::swap` 会调用赋值运算符来为中间变量赋值，此时 `std::swap` 和赋值会形成重复调用，造成迭代栈溢出



写时拷贝 *Copy-on Write COW*

<https://coolshell.cn/articles/12199.html>

浅拷贝的问题及其解决方案

- 析构两次。解决方案：增加一个引用计数，每个对象析构时，--引用计数，最后一个析构的对象释放空间
- 一个对象修改影响另外一个对象。解决方案：写时拷贝，本质是一种延迟拷贝，即谁去写谁做深拷贝，若没人写就可以节省空间了

vector和list

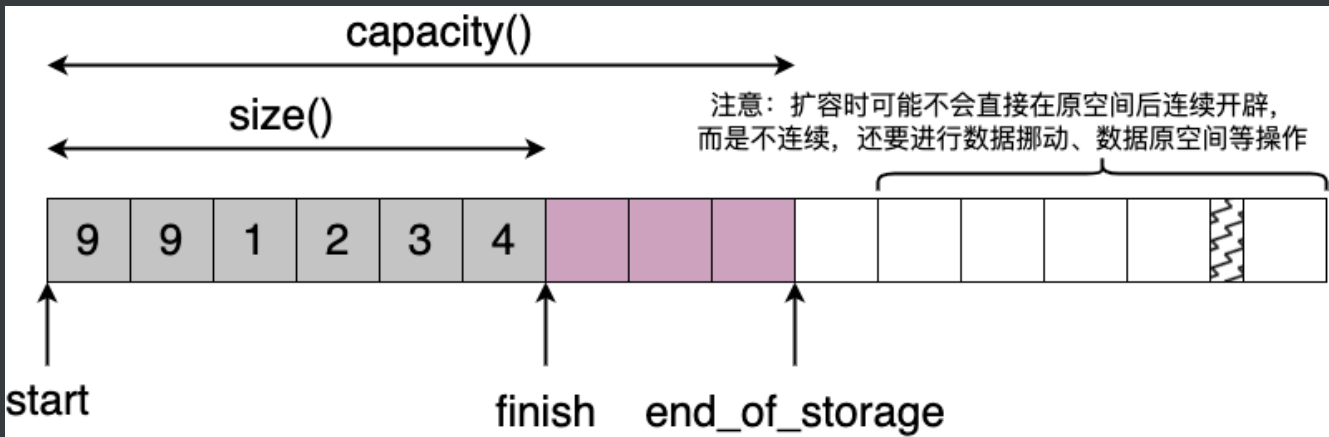
vector 顺序表

定义

```
class template
std::vector <vector>

template < class T, class Alloc = allocator<T> > class vector; // generic template
```

- vector是一个类模板，对应的是数据结构中的顺序表/数组。比如声明一个存储int数据的数组 `vector<int> v1;`
- vector成员



vector迭代器在 insert 和 erase 中的失效问题

- 问题一：当insert（在pos前插入数据）中进行扩容时因为会重新分配空间会出现迭代器失效问题
 - 旧的pos发生了越界造成了野指针问题。如下图所示，在扩容后pos指针不会有变化，已经不处于新开的空间中了，即不处于老的_start和_finish之间

```

102 void insert(iterator pos, const T& x)
103 {
104     assert(pos >= _start && pos <= _finish);
105     if (_finish == _end_of_storage) // 扩容
106         reserve(capacity() == 0 ? 4 : capacity() * 2);
107     iterator end = _finish - 1;
108     while (end >= pos) // 挪动数据
109     {
110         *(end + 1) = *end;
111         end--;
112     }
113     *pos = x;
114     _finish++;
115 }

```

扩容前		
名称	值	类型
this	0x00fafc08 {_start=0x01535f90 {1} _finish=0x0153...	wjff::vector<int> *
_start	0x01535f90 {1}	int *
_finish	0x01535fa0 {-33686019}	int *
_end_of_storage	0x01535fa0 {-33686019}	int *
pos	0x01535f98 {3}	int *
扩容后		
名称	值	类型
this	0x00fafc08 {_start=0x0153e388 {1} _finish=0x0153...	wjff::vector<int> *
_start	0x0153e388 {1}	int *
_finish	0x0153e398 {-842150451}	int *
_end_of_storage	0x0153e3a8 {-33686019}	int *
pos	0x01535f98 {-572662307}	int *

越界

- 解决失效问题：扩容前计算pos和_start的相对位置，扩容后令原pos重新指向
- 问题二：扩容/缩容引发野指针问题。在p位置修改插入数据以后不要访问p，因为p可能失效。这是因为调用insert的时候pos是传值传参，内部对pos的修改不会影响实参。STL库中的实现也没有给pos设置为传引用传参，因为这又会引起一些其他的问题，因此我们尽量跟STL保持一致；erase库中的实现有可能会出现缩容的情况，但是很少，此时也不要再在erase后解引用访问

```

1  vector<int> v1;
2  v1.push_back(1);
3  v1.push_back(2);
4  v1.push_back(3);
5  v1.push_back(4);
6  vector<int>::iterator pos = find(v1.begin(), v1.end(), 3);
7  if (pos != v1.end())
8      v1.insert(pos, 30);
9  for (auto e : v1) { cout << e << " "; } //可能会访问野指针
10 cout << endl;

```

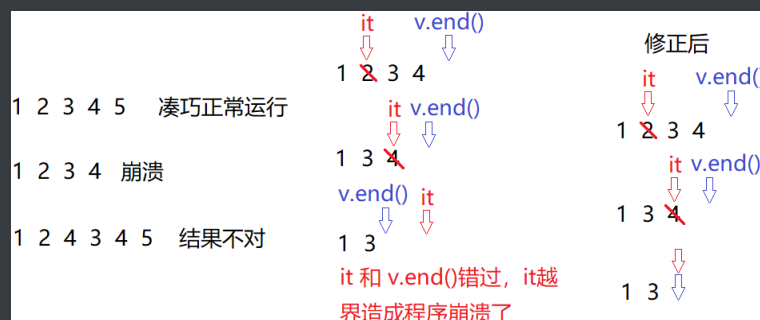
- 问题三：因为数据挪动，pos在insert/erase之后位置发生了改变

- 问题代码

```

1 void erase(iterator pos) {
2     assert(pos >= _start && pos < _finish);
3     // 从前往后挪动数据
4     iterator begin = pos + 1;
5     while (begin < _finish) {
6         *(begin - 1) = *begin;
7         begin++;
8     }
9     _finish--;
10 }
11 void test() {
12     while (it != v.end()) {
13         if (*it % 2 == 0)
14             v.erase(it);
15         it++;
16     }
17 }

```



- 修正：每次insert/erase之后要更新迭代器（++），STL规定了erase/insert要返回删除/插入位置的下一个位置迭代器

```

1 // STL规定了erase要返回删除位置的下一个位置迭代器
2 iterator erase(iterator pos) {
3     assert(pos >= _start && pos < _finish);
4     // 从前往后挪动数据
5     iterator begin = pos + 1;
6     while (begin < _finish) {

```

```

7         *(begin - 1) = *begin;
8         begin++;
9     }
10    _finish--;
11    return pos; // 返回删除数据的下一个位置 还是pos
12 }
13 void test() {
14     while (it != v.end()) {
15         if (*it % 2 == 0)
16             //v.erase(it);
17             it = v.erase(it);
18         else
19             it++; // erase之后更新迭代器
20     }
21 }

```

总结：insert/erase之后不要直接访问pos。一定要更新，直接访问可能会出现各种意料之外的结果，且各个平台和编译器都有可能不同。这就是所谓的迭代器失效

拷贝构造与高维数组的浅拷贝问题

https://www.jb51.net/article/272165.htm#_lab2_1_3

- 高维数组或容器里存的数据是另外的容器的时候，就会涉及到更深层次的深拷贝问题

```

// 浅拷贝
vector<vector<int>> generate(int numbers)
{
    vector<vector<int>> vv;
    vv.resize(numbers); // 初始化每一行
    // 每一行的首元素赋为1
    for (size_t i = 0; i < vv.size(); i++)
    {
        vv[i].resize(i + 1, 0);
        vv[i].front() = vv[i].back() + 1;
    }
    for (size_t i = 0; i < vv.size(); i++)
    {
        for (size_t j = 0; j < vv[i].size(); j++)
            if (i < j)
                vv[i][j] = vv[i - 1][j] + vv[i - 1][j - 1];
    }
    return vv;
}
vector<vector<int>> rut = vv;

```

```

// 深拷贝
vector<vector<int>> generate(int numbers)
{
    vector<vector<int>> vv;
    vv.resize(numbers); // 初始化每一行
    // 每一行的首元素赋为1
    for (size_t i = 0; i < vv.size(); i++)
    {
        vv[i].resize(i + 1, 0);
        vv[i].front() = vv[i].back() + 1;
    }
    for (size_t i = 0; i < vv.size(); i++)
    {
        for (size_t j = 0; j < vv[i].size(); j++)
            if (i < j)
                vv[i][j] = vv[i - 1][j] + vv[i - 1][j - 1];
    }
    return vv;
}
vector<vector<int>> rut = vv;

```

浅拷贝

浅拷贝存储的内容相同

```

// 浅拷贝写法1
vector<const vector<int>> v> // v2(v1)
{
    _start = new T[v.size()]; // 开v.capacity()个空间
    memcpy(_start, v._start, sizeof(T) * v.size());
    _finish = _start + v.size();
    _end_of_storage = _start + v.size();
}

```

深拷贝

深拷贝存储的内容不同

```

// 深拷贝写法1
vector<const vector<T>& v> // v2(v1)
{
    _start = new T[v.size()]; // 开v.capacity()
    // 不能调用memcpy, memcpy是浅拷贝, 应该调用
    // memcpy(_start, v._start, sizeof(T) * v.size());
    for (size_t i = 0; i < v.size(); i++)
        _start[i] = v._start[i];
    _finish = _start + v.size();
    _end_of_storage = _start + v.size();
}

```

原因是因为对于类似于在string的深拷贝实现使用的是memcpy函数，若容器中存的还是容器类的数据类型，那么它进行的仍然是浅拷贝。上图是利用杨辉三角OJ题做的试验，存储的数据类型是 `vector<vector<int>>`，可以看到左图虽然外层vector实现了深拷贝，但内容的数据仍然是浅拷贝。右边进行了修正

- 深拷贝构造的写法：本质上是调用对象T实现的深拷贝复制运算符

- 传统写法

```
1  // 传统写法1
2  vector(const vector<T>& v) { // v2(v1)
3      _start = new T[v.size()]; // 开v.capacity()个空间也可以，各有各
    的优势和劣势
4      // 不能使用memcpy，memcpy也是浅拷贝，当出现类似与
    vector<vector<int>> 这种多维数组就会有问题
5      // memcpy(_start, v._start, sizeof(T) * v.size());
6      for (size_t i = 0; i < v.size(); i++)
7          _start[i] = v._start[i]; //本质上是调用对象T实现的深拷贝
    复制运算符
8      _finish = _start + v.size();
9      _end_of_storage = _start + v.size();
10 }
11 // 传统写法2
12 vector(const vector<T>& v)
13     : _start(nullptr)
14     , _finish(nullptr)
15     , _end_of_storage(nullptr)
16 {
17     reserve(v.size());
18     for (const auto& e : v) // 若是vector则传值要深拷贝，因此用引用
19         push_back(e); // push_back 会自动处理_finish和
    _end_of_storage
20 }
```

- 现代写法

```
1  // 现代写法
2  // 提供一个迭代器区间构造
```



```

3  template <class InputIterator>
4  vector(InputIterator first, InputIterator last)
5      :_start(nullptr)
6      , _finish(nullptr)
7      , _end_of_storage(nullptr) {
8      while (first != last) {
9          push_back(*first);
10         first++;
11     }
12 }
13 void swap(vector<T>& v) {
14     std::swap(_start, v._start);
15     std::swap(_finish, v._finish);
16     std::swap(_end_of_storage, v._end_of_storage);
17 }
18 vector<T> operator=(vector<T> v) {
19     swap(v);
20     return *this;
21 }
22 vector(const vector<T>& v)
23     :_start(nullptr)
24     , _finish(nullptr)
25     , _end_of_storage(nullptr)
26 {
27     vector<T> tmp(v.begin(), v.end());
28     swap(tmp);
29 }

```

list 链表

List是一个允许常数时间插入和删除的顺序容器，支持双向迭代器

list的特殊operation

- splice：直接转移、拼接一个list到另外一个list上
- remove
- sort 排升序：属于list的sort和algorithm库中的sort的区别在于list的空间不连续，而algorithm的适用对象是连续空间，不能用于list。这是因为algorithm的底层qsort需要实现三数取中法。**list的sort底层是MergeSort**，而且list的归并不需要额外的空间了
- list排序 VS vector排序：大量数据的排序vector效率远高于list，虽然MergeSort和qsort的效率都是 $O(N\log N)$ ，但是vector的随机访问提供了巨大优势。**不要使用列表排序！**

list的insert迭代器不失效，erase迭代器仍然失效

- List的insert是在pos位置之前插入，它采取的方式新建一个newnode，然后改变指针指向，并没有挪动数据，因此迭代器的位置不会改变也不会出现野指针问题
- 但是当erase时迭代器仍然会失效：pos被erase，也就是被free了，这时候访问就是野指针问题。因此erase的返回值也是iterator，方便进行更新

list的数据结构与迭代器模拟实现

list的增删查改实现可以参考数据结构中的双向带头循环列表的实现

list的实现重点在于迭代器，因为list的迭代器不像vector是每一个元素的原生指针，而是指向一个节点node，节点中才规定了连接关系

- C++中更倾向于使用独立的类封装，而不是使用内部类。因此list的设计由三个独立的类组成：list_node 类、迭代器和 list 总体分别封装成独立的类（这里 list_node 和迭代器 直接用了 struct，因为要将类成员设置为公有，供 list 使用）
- C++用**带头双向循环链表**来实现list，所以每个node里要带prev、next和头节点

```

1  template<class T>
2  struct list_node { // 用struct和STL库保持一致
3  // 同时struct默认公有，可以让其他成员调用
4      list_node(const T& x = T()) // T() 为数据类型的默认构造函数
5          :_data(x)
6          , _next(nullptr)
7          , _prev(nullptr)
8      {}
9      T _data;
10     list_node<T> *_next;
11     list_node<T> *_prev;
12 }

```

- 迭代器类封装方便迭代器的运算符重载

```

1  template<class T, class Ref, class Ptr> // 准备多个模板参数是为了
    const_iterator复用
2  struct _list_iterator {
3      typedef list_node<T> Node;
4      typedef _list_iterator<T, Ref, Ptr> iterator;
5
6      Node *_node; // 迭代器类的唯一成员变量，实际上迭代器就是封装的Node*
7
8      _list_iterator(Node* _node)
9          :_node(node)
10     {}
11     // 对迭代器结构体的运算符重载: != == * ++ -- ->
12     bool operator!=(const iterator& it) const;
13     bool operator==(const iterator& it) const;
14     Ref operator*();
15     Ptr operator->();
16     iterator& operator++() // 前置++
17 }

```

- 不需要析构函数，默认的析构函数不会处理Node* 指针这种内置类型。这是合理的，因为不可能在用户使用迭代器操作后，把list中的节点给销毁了

- **也不需要写拷贝构造**，因为拷贝得到的迭代器必然要指向同一个地址，因此默认的浅拷贝就够了。而浅拷贝不会报错的原因是因为轮不到迭代器进行析构，迭代器只是封装，list会将所有的一块析构掉
- `_list_iterator` 不支持 `++` `--` `<` `>` 等操作符，因为空间地址不连续，这些操作没有意义
- `const_iterator` 和 `iterator` 的区别是是否能够修改数据，即是返回 `T&` 还是 `const T&`。不能使用函数重载，因为仅仅只有返回值不同的话是不构成函数重载的。因此考虑使用类模板复用的方式
- `++` 等操作的返回值为 `iterator` 的原因是因为也要支持 `const_iterator` 等其他迭代器，即要支持 `typedef _list_iterator<T, Ref, Ptr> iterator` 模板实例化后得到的所有迭代器

■ list总体

```

1  template<class T>
2  class list {
3      typedef list_node<T> Node; // Node是只给当前类使用的封装，因此设置为私有
4  public:
5      typedef _list_iterator<T, T&, T*> iterator; // 普通迭代器
6      typedef _list_iterator<T, const T&, const T*> const_iterator; // const迭代器
7      typedef __reverse_iterator<iterator, T&, T*> reverse_iterator;
8      // 反向迭代器
9      typedef __reverse_iterator<const_iterator, const T&, const T*>
10     const_reverse_iterator // const反向迭代器
11     // 迭代器+增删查改接口
12 private:
13     Node* _head; // 类成员变量只有一个哨兵位头结点
14 }

```

- 迭代器中特殊的运算符重载 `->`
 - 考虑当list中存的是一个自定义类型Pos

```

1  struct Pos {
2      int _a1;
3      int _a2;
4      Pos(int a1 = 0, int a2 = 0)
5          :_a1(a1)
6          , _a2(a2)
7      {}
8  };
9
10 list<Pos> lt;
11 lt.push_back(Pos(10, 20));
12 lt.push_back(Pos(10, 21));

```

- `T* operator->()` 返回的是`lt`中存储的一个结构体指针`*Pos`，若要取到其实中的数据应该要 `it->->_a1`，因为`it`里首先存的是`node`，`node`里才存的是数据。但编译器为了提高可读性，进行了特殊处理，即省略了一个 `->`，自动取到的就是`Pos`中的一个数据。因此当`lt`中存储的是自定义类型或者内置类型时，`->` 都可以看作是迭代器指针取数据

```

1  T& operator*()
2      return _node->_data;
3  T* operator->()
4      return &(operator*());

```

list的反向迭代器，采用适配器（复用）的方向进行设计

- 迭代器按功能分类
 - `forward_iterator`：只支持`++`，不支持`--`：比如`forward_list`、`unordered_map`、`unordered_set`
 - `bidirectional_iterator`：既支持`++`，也支持`--`：比如`list`、`map`、`set`
 - `random_access_iterator`：不仅支持`++--`，还支持`+-`，比如`vector`、`deque`
- 实现方法
 - 普通思维：拷贝一份正向迭代器，对其进行修改

- STL的设计：对iterator进行复用。反向迭代器里封装的是正向迭代器，正向迭代器里封装的是指针或节点

```

1  template<class Iterator, class Ref, class Ptr>
2  struct __reverse_iterator {
3      Iterator _curr; // 类成员，当前的正向迭代器
4      typedef __reverse_iterator<Iterator, Ref, Ptr> RIterator;
5      __reverse_iterator(Iterator it)
6          : _curr(it)
7      {}
8  }
9
10 RIterator operator++() { // 前置++, ++方向置为反向
11     --_curr;
12     return *this;
13 }
14 RIterator operator--();
15 Ref operator*(); // operator*的实现比较特殊，采用了和iterator的对称设计，见下方
16 Ptr operator->();
17 bool operator!=(const RIterator& it);

```

- Ref operator*() 的特殊设计：将 end 与 rbegin 以及 begin 与 rend 设计为对称关系

end与rbegin以及begin与rend的对称关系

正向迭代器：end() [begin()

head 1 2 3 4 5

原来设计：(rend() rbegin())

对称设计：(rbegin() rend())

```

iterator begin() { return (link_type)((*node).next); }
iterator end() { return node; }
reverse_iterator rbegin() { return reverse_iterator(end()); }
reverse_iterator rend() { return reverse_iterator(begin()); }

```

reference operator*() const {

Iterator tmp = current;
return *--tmp;

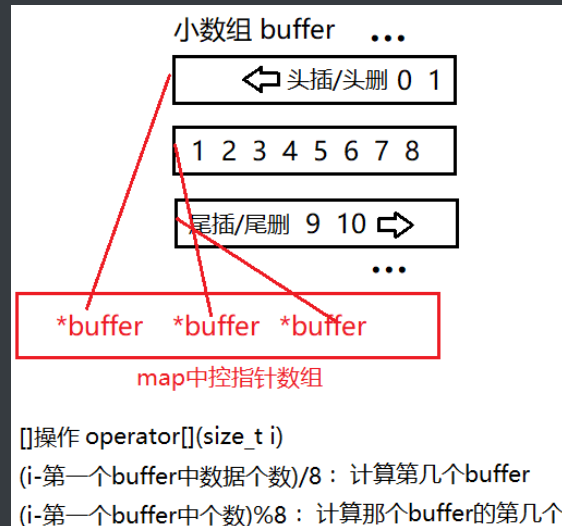
}

STL的特殊设计：对rbegin解引用得到的是它前一个data

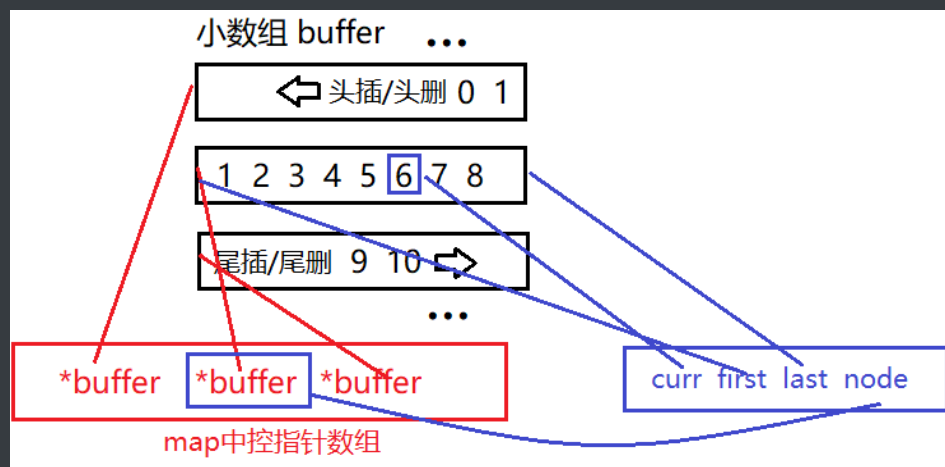
- 只要实现了正向迭代器，那么 reverse_iterator 可以复用到其他的容器上，除了 forward_list，unordered_map 和 unordered_set 不能被复用，因为这些容器的迭代器不支持 --

stack & queue 栈和队列

deque容器



- deque每次既不是开一个节点，也不是进行realloc，而是开多个可以存多个数据的小buffer
- Double ended queue 双端队列融合了vector和list的优点，既支持list的任意位置插入删除，又支持vector的下标随机访问
- 设计缺陷：
 - 相比vector，operator[] 的计算稍显复杂，大量使用会导致性能下降
 - 中间插入删除效率不高
 - 从底层角度看迭代器的设计非常复杂



- curr为当前数据
- first和last表示当前buffer的开始和结束

- node反向指向中控位置，方便遍历时找下一个buffer
- 结论：相比vector和list，deque非常适合做头尾的插入删除，很适合去做stack和queue的默认适配容器；但如果是中间插入删除多用list，而随机访问多用vector

仿函数 *Functor*

仿函数又称为函数对象 function object

在优先级队列中需要比较元素的大小以进行建堆。但在建堆的逻辑中，比较大小是写死的，若此时要将大堆切换成小堆，就需要进入源码中改比较的顺序。这在实际中是不可能的。因此在类实例化时候就需要提供一个“开关”进行控制

仿函数是一个类/结构体，它的目标是使一个类的使用看上去像一个函数。其实现就是类中实现一个 `bool operator()` (**也是运算符!**)，这个类就有了类似函数的行为，就是一个仿函数类了

模板参数只允许是类，因此仿函数就可以很好的放入模板中进行泛型编程

```
1 namespace wjf { //9.24的课
2     template<class T> //less和greater在std中都被提供了
3     struct less { //用class也行，但要设置为public
4         bool operator()(const T& l, const T& r) const {
5             return l < r;
6         }
7     };
8     template<class T>
9     class greater { //用class也行，但要设置为public
10    public:
11        bool operator()(const T& l, const T& r) const {
12            return l > r;
13        }
14    };
15 }
16
17 //调用，二者等价
18 wjf::less<int> lsFunc; //仿函数是一个类，别忘了实例化
```



```
19 cout << lsFunc(1, 2) << endl; //看起来就像是一个普通的函数调用
20 cout << lsFunc.operator()(1, 2) << endl; //本质就是调用运算符重载
```

要注意的点是类实例化时要传入的是类型，而函数重载时传入的则是对象。比如 `std::sort()` 是一个函数重载，它在调用的时候要传入一个 `Compare comp` 的对象（`std::greater<int>()` 对象有括号），而对于 `priority_queue` 类实例化则传入的是 `less<int>` 这种类型

```
1 vector<int> v;
2 sort(v.begin(), v.end(), less<int>()); // 传入一个匿名对象
3
4 priority_queue<int, vector<int>, Compare = less<int>> pq; // 传入一个类型
5 // 优先级队列的STL参数设计有写问题，因为一般不太会更改默认容器vector，但是有可能更改
   排序方式，即Compare，因此应该将Compare放在第二位比较好
```

*stack & queue*适配器

stack

class template

std::stack

<stack>

template <class T, class Container = deque<T> > class stack;

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

- 适配器 adapter 是一种设计模式，该种模式是将一个类的接口转换成客户希望的另外一个接口。其实就是复用以前的代码
- stack默认复用的容器是deque，也可以复用vector和list

queue

queue 和 stack 一样都是默认复用 deque 的适配器

priority_queue 优先级队列容器适配器

class template

std::priority_queue

<queue>

```
template <class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;
```

- priority_queue 可以实现最大值/最小值在队头。仿函数Compare默认为 std::less , 大数优先, T要支持 operator<() ; Compare为 std::greater 时为小数优先, T要支持 operator>()
- priority_queue 的底层是堆, 但用vector来控制下标
- priority_queue 的默认容器为vector, 这是因为要进行大量的下标随机访问, 用deque 或list都不好

继承 Inheritance

继承的概念及定义

继承是类层次设计的复用

继承格式

```

1  class Person {
2  public:
3      void Print();
4  protected:
5      string _name;
6      int _age;
7  };
8
9  class Student : public Person { // 指明类派生列表
10 protected:
11     int _stuID;
12 };

```

其中Person被称为父类/基类 Base class，而Student被称为子类/派生类 Derived class，Person前的访问限定符规定了派生类的继承方式

派生类通过块域 {} 符号之前的类派生列表 class derivation list 来明确指出它是从哪个基类继承而来的

继承关系和访问限定符

类成员/继承方式	public继承	protected继承	private继承
基类的public成员	派生类的public成员	派生类的protected成员	派生类的private成员
基类的protected成员	派生类的protected成员	派生类的protected成员	派生类的private成员
基类的private成员	在派生类中不可见	在派生类中不可见	在派生类中不可见

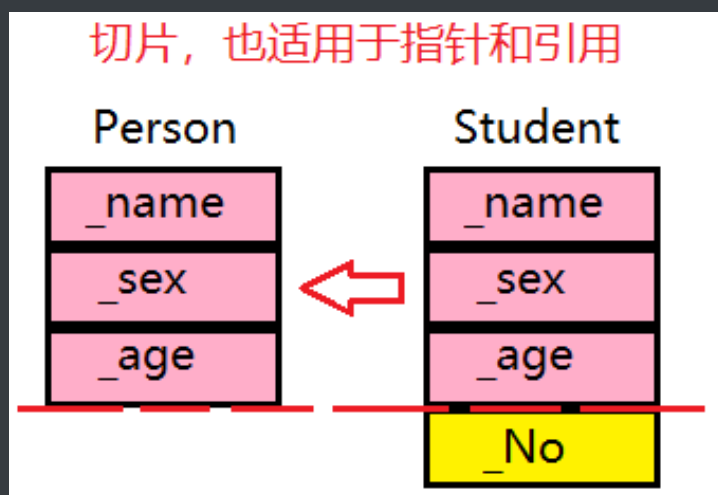
- 不可见/隐藏 hide 是指基类的私有成员虽然还是被集成到了派生类对象中，但是语法上限制派生类对象不管在类内还是类外都~~不能~~去访问它，这种继承的使用情况极少
- 当省略继承方式时，class的默认继承方式是private，而struct的默认继承方式是public
- 实际运用中一般只会使用public继承，然后通过类成员是用public还是protected修饰来区分分类内外是否可以访问

继承中的作用域

- 在继承体系中父类和子类都有独立的作用域
- 父类和子类中有同名成员时，子类成员将直接屏蔽对父类中同名成员的直接访问，这种情况叫做**隐藏 hide**，也叫做重定义。若在子类中想要访问父类中被隐藏的成员，可以通过指定父类域的方式，即 父类::父类成员 显式访问
- 对于成员函数只要函数名相同就构成隐藏。注意在不同类域中的同名函数不是函数重载
- 实际中最好不要定义相同命名的成员

派生类的默认成员函数

基类和派生类对象赋值转换：切片 slicing



- 每一个子类对象都是一个特殊的父类成员。实际上如上图所示，每一个子类对象中，都首先存放着父类成员
- 子类对象可以赋值给父类对象、父类的指针和父类的引用，这种赋值方式称为切片/切割
- 反过来父类对象不能赋值给子类对象
- 父类的指针或者引用可以通过强制类型转换赋值给子类的指针或者引用。但是必须是父类的指针是指向派生类对象时才是安全的。若父类是多态类型，可以使用RTTI的 `dynamic_cast` 来进行识别后进行安全转换

子类的默认构造函数、析构函数

- 自己的成员还是一样调用自己的默认构造函数
- 继承自父类的成员需要调用父类的构造函数进行初始化
- 若父类仅提供了不带缺省值的构造函数或者说没有可用的默认构造函数时，这个时候就需要子类为继承的父类成员提供显式构造。注意，必须调用整个父类的构造函数而不是为父类成员单独赋值，这被称为合成版本
- 析构函数同上。但析构函数有一个特殊点：子类的析构函数跟父类的析构函数构成隐藏，这是由于多态的需要，析构函数会被编译器统一处理成 `destructor()` 从而构成了隐藏，在调用父类的析构函数时需要指定类域
- 为了保证先子类析构，再父类析构的正确析构顺序，编译器会在子类析构后自动调用父类析构函数，用户不应该显式地给出父类的析构函数

```
1  class Student : public Person {
2  public:
3      Student(const char* name, int num)
4          :Person(name)
5          , _num(num)
6      {}
7  protected:
8      int _num;
9  };
10 ~Student() {
11     //Person::~~Person(); // 指定类域，避免隐藏
12     // ...
13 }
```

子类的默认拷贝构造、`operator=`

- 自己的成员还是一样调用自己的默认拷贝构造函数 -- 内置类型浅拷贝，自定义类型调用它的拷贝构造
- 继承自父类的成员需要调用父类的拷贝构造函数进行初始化
- 当子类中需要深拷贝时，需要显式提供子类拷贝构造函数：利用赋值转换切片原理为父类赋值，或者利用强制类型转换取数据 `*(Person)*this`

- 复制运算符重载同上，只是要注意显式调用父类的 `operator=` 以避免隐藏

```
1 // ...
2 Student(const Student& s) // 子类显式的拷贝构造
3     :Person(s) // 赋值转换切片
4 {}
5 Student& operator=(const Student& s) {
6     if (this != &s) {
7         Person::operator=(s); // 显式调用父类的operator=
8         _num = s._num;
9     }
10    return *this;
11 }
```

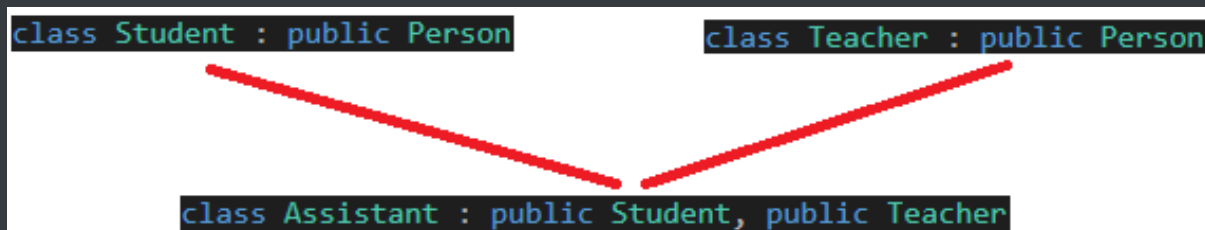
友元和静态成员的继承

- 友元关系不能被继承，也就是说父类的友元不能访问子类私有和保护成员
- 基类定义了static静态成员，则整个继承体系里面只有一个这样的成员，无论派生出多少个子类，都只有一个static成员实例
- 不能说父类或子类对象中包含了所有父类或子类的成员变量，静态变量就不属于父类或子类对象，而是属于整个类

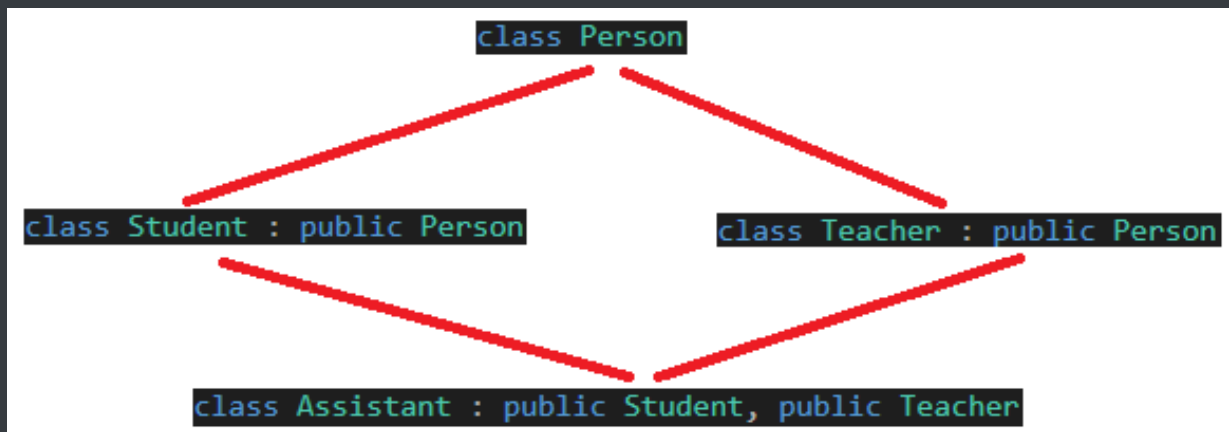
菱形继承与菱形虚拟继承

继承分类

- 单继承 Single Inheritance：一个子类只有一个直接父类时称这个继承关系为单继承
- 多继承 Multiple Inheritance：一个子类有两个或以上直接父类时称这个继承关系为多继承



- 菱形继承 Diamond Inheritance：是多继承的一种特殊情况



多继承的指针偏移问题

```

class Base1 { public: int _b1; };
class Base2 { public: int _b2; };
class Derive : public Base1, public Base2 { public: int _d; };

int main()
{
    Derive d;
    Base1* p1 = &d;
    Base2* p2 = &d;
    Derive* p3 = &d;

    return 0;
}
  
```

名称	值
▸ p1	0x0039fcd0 { _b1=-858993460 }
▸ p2	0x0039fcd4 { _b2=-858993460 }
▸ p3	0x0039fcd0 { _d=-858993460 }
▴ d	{ _d=-858993460 }
▸ Base1	{ _b1=-858993460 }
▸ Base2	{ _b2=-858993460 }
▸ _d	-858993460

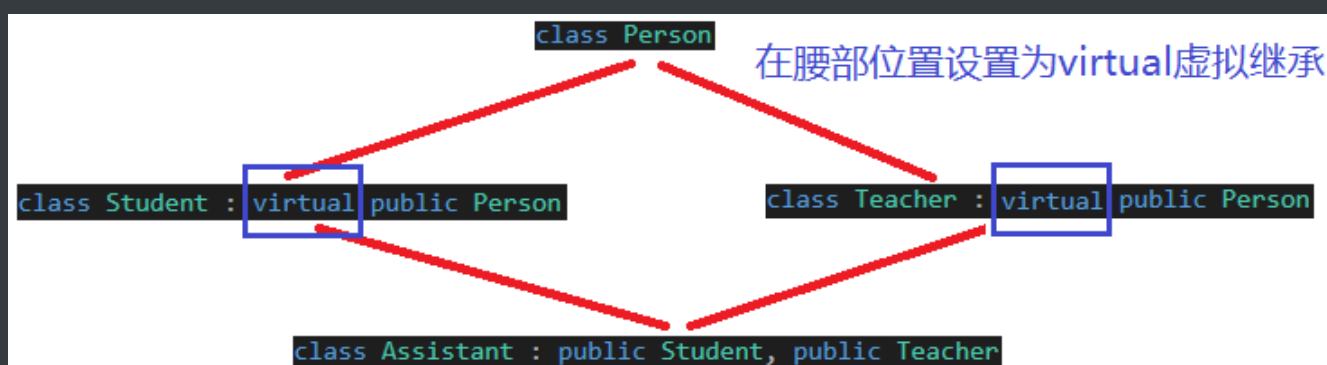
- 对于同一个多继承的子类对象，当进行切片时，父类的继承顺序会有影响。如上例，由于先继承了Base1，后继承了Base2，所以根据栈的性质Base1对象放在低地址，而Base2产生了指针偏移，由于Base1中只有一个int对象，因此Base2放在离Base1有4个字节远的高地址
- 当分别进行切片时，p1看到的的就是头四个字节，p2看到的范围为p1+4，而d则看到了全部，这就是切片的原理

菱形继承问题及虚拟继承

- 菱形继承的二义性问题：通过指定类作用域解决

```
1 Assistant at;  
2  
3  ///// 菱形继承的二义性问题  
4  //at._name = "张三";  
5  at.Student::_name = "张三";  
6  at.Teacher::_name = "李四";
```

- 数据冗余问题：同样的一个人有多个年龄、性别？Student和Teacher的类成员已经有了_age 和 _sex，但经过继承和修改后，子类可能有不同的数据，这是多继承带来的问题，单继承就没有这个问题，因为单继承的父类属性只有一份。需要通过菱形虚拟继承来解决



- 实际中都会避免定义成菱形继承，菱形继承的情况很少见，比如C++ IO流库中的iostream，底层机制会非常复杂
- 菱形虚拟继承的底层：取决于虚继类_a大不大，若很大就能节省空间，比如说当_a是一个4000字节的类对象时，我们只需要增加一个4字节虚继表指针就可以换来少存储一个冗余的_a对象
 - 若不使用虚拟继承，则编译器会根据继承顺序，算好空间后依次存放数据


```

class A
{
public:
    int _a;
};
class B : public A
//class B : virtual public A
{
public:
    int _b;
};
class C : public A
//class C : virtual public A
{
public:
    int _c;
};
class D : public B, public C
{
public:
    int _d;
};
int main()
{
    D d;
    d.B::_a = 1;
    d.C::_a = 2;
    d._b = 3;
    d._c = 4;
    d._d = 5;
    return 0;
}

```

- 1、菱形继承有冗余数据，在这里总共存储了B::_a, _b, C::_a, _c, _d 5个int，总共20个字节的空间
- 2、继承关系为先B再C，同时因为栈的性质，最先开出来的内存是在低地址的，因此从0x005AF910到0x005AF920依次存放了上述5个int数据

内存 1				
地址: 0x005AF910				
0x005AF910	01 00 00 00		
0x005AF914	03 00 00 00		
0x005AF918	02 00 00 00		
0x005AF91C	04 00 00 00		
0x005AF920	05 00 00 00		
0x005AF924	cc cc cc cc	????		
0x005AF928	00 f0 07 54	.?.T		
0x005AF92C	4c f9 5a 00	L.Z.		
0x005AF930	13 21 87 00	.!.		

- 使用了虚拟继承，此时在原来存放着继承量的地方就会改为存放指针（该指针空间称为虚继表），指针中存放的是一个偏移量，这个偏移量阐明了继承值和虚拟继承后的共同虚继类之间差了几个字节，从而帮助编译器找到

```

class A
{
public:
    int _a;
};
//class B : public A
class B : virtual public A
{
public:
    int _b;
};
//class C : public A
class C : virtual public A
{
public:
    int _c;
};
class D : public B, public C
{
public:
    int _d;
};
int main()
{
    D d;
    d.B::_a = 1;
    d.C::_a = 2;
    d._b = 3;
    d._c = 4;
    d._d = 5;
    B b = d; // 切片
    B* pb = &d;
    C c = d;
    C* pc = &d;
    return 0;
}

```

当菱形虚拟继承后只保留一个公共的_a，并将其存储在最高位的地址，非虚拟继承时存储_a的地址都被用来保存了一个指针，该指针用来指向一个保存便宜量的地址，方便切片时寻址

&d	0x012ffcfc {_d=5}
b	{_b=3}
pb	0x012ffcfc {_b=3}
c	{_c=4}
pc	0x012ffd04 {_c=4}

内存 1				
地址: 0x012FFCFC				
0x012FFCFC	dc 7b 4d 00	{m.		
0x012FFD00	03 00 00 00		
0x012FFD04	e4 7b 4d 00	{m.		
0x012FFD08	04 00 00 00		
0x012FFD0C	05 00 00 00		
0x012FFD10	02 00 00 00		
0x012FFD14	cc cc cc cc	????		
0x012FFD18	92 dd 7c be	?? ?		
0x012FFD1C	3c fd 2f 01	<?/.		
0x012FFD20	13 21 4d 00	!.M.		
0x012FFD24	01 00 00 00		
0x012FFD28	00 4a 52 01	.JR.		
0x012FFD2C	d8 a6 52 01	??R		

内存 2				
地址: 0x004D7BDC				
0x004D7BDC	00 00 00 00		
0x004D7BE0	14 00 00 00		
0x004D7BE4	00 00 00 00		
0x004D7BE8	0c 00 00 00		

004d7bdc是一个指针，指向的是一个预留空间
该空间的下一个int*保存的就是到共同的_a的偏移量

- 同时在虚拟继承后，设置为虚拟继承的类的底层存储方式也要发生变化。若不保持一致的模型，则会对其他指针的访问产生歧义，B类原来只需要开8个字节的大小，现在为了保存一个虚继表指针，变成了12字节

```

class A
{
public:
    int _a;
};

class B : virtual public A
{
public:
    int _b;
};

B bb;
bb._a = 1;
bb._b = 2;
return 0;

```

虚继表中保存了初始地址与继承来的虚继类的偏移量

内存 1			
地址: 0x00B8FA48			
0x00B8FA48	cc	7b	a8 00
0x00B8FA4C	02	00	00 00
0x00B8FA50	01	00	00 00
0x00B8FA54	cc	cc	cc cc
0x00B8FA58	cc	cc	cc cc
0x00B8FA5C	dc	7b	a8 00

内存 2			
地址: 0x00A87BCC			
0x00A87BCC	00	00	00 00
0x00A87BD0	08	00	00 00
0x00A87BD4	00	00	00 00
0x00A87BD8	08	00	00 00
0x00A87BDC	00	00	00 00

名称	值	类型
8&bb	0x00b8fa48 {_b=2}	B *

第一行是为了菱形继承的多态时，计算多态的虚表时的偏移量

- 总结：当大量数据时，虚拟继承对底层的改造是有可能影响数据的，尽量不要设计出菱形继承和虚拟继承

继承和组合

- 继承：is-a
- 组合：has-a

多态 Polymorphism

多态的定义及实现

构成多态的两个条件

多态是在不同继承关系的类对象，去调用统一函数，却产生了不同的行为。构成多态有两个条件

- 必须通过父类的指针或者引用调用虚函数
- 被调用的函数必须是虚函数，且子类必须对父类的虚函数进行重写

虚函数重写的条件 Override of virtual function

- 若想要形成某个成员函数的多态行为，就必须将成员函数声明为虚函数。虚函数是被 `virtual` 关键字修饰的类成员函数
- 对虚函数的重写/覆盖：子类中有一个跟父类完全相同的虚函数（即子类虚函数与父类虚函数的返回值类型、函数名字、参数列表完全相同，不要求参数的缺省值是否相同），称子类的虚函数重写了父类的虚函数。不符合重写就是隐藏关系

虚函数重写的三个例外

- 子类虚函数不加 `virtual`，依旧构成重写，这是因为编译器认为子类从父类那里已经继承了`virtual`，这种特例其实是为了析构函数的重写所准备的。实际中最好也加上构成同一的形式。

```
1 class Person {virtual void BuyTicket();};
2 class Student {void BuyTicket();};
```

- 析构函数的重写：若父类的析构函数为虚函数，此时无论子类析构函数是否定义或是否加`virtual`，二类都构成重写。这是因为编译器对析构函数的特殊处理，即编译后析构函数的名称同一处理为`destructor`

```
1 // 析构函数构成虚函数重写
2 class Person {virtual ~Person() {}};
3 class Student {~Student () {}};
```

- 问题：为什么建议在继承中析构函数定义成虚函数？

```
1 Person *ptr1 = new Person;
2 Person *ptr2 = new Student; // 在某些情况下使用了切片
3 delete ptr1;
4 delete ptr2; // 若此时析构不构成多态，则此时会调用Person的析构函数，而不是
               Student的析构函数
```

- 协变返回类型 Covariant returns type: 返回值可以不同，但要求必须是父子关系的指针或者引用。注意：可以不是虚函数所属类的父子关系，只要是构成父子关系的类都可以

```
1 class Person {virtual Person* BuyTicket();};
2 class Student {virtual Student* BuyTicket();};
3
4 class Person {virtual A* BuyTicket();}; // B是A的子类, A和B与Person和
    Student无关
5 class Student {virtual B* BuyTicket();};
```

C++ 11 override 和 final

- 若一个虚函数不想被重写，可以被定义为 final，但这种场景极少。因为虚函数的目标就是为了重写 `virtual void BaseFunc() final {};`
- override 检查子类虚函数是否完成重写，若没有完成重写就报错 `virtual void DerivedFunc() override {};`

抽象类/接口类

概念

- 纯虚函数 Pure virtual function: 在虚函数的后面写上 `=0`
- 包含纯虚函数的类叫做抽象类/接口类。抽象类不能实例化出对象，子类继承后也不能实例化出对象，只有重写纯虚函数后才能实例化出对象。一个大的概念就很适合定义成抽象类，比如树、花、车等，然后再定义具体的子类，比如桃树、玫瑰花、BYD等
- 纯虚函数规范了子类必须重写，相当于规定了子类必须实现哪些接口，这也体现出了接口继承的概念

```
1 class Car {
2 public:
3     virtual void Drive() = 0;
4 }
5
6 class BYD : public Car {
7 public:
8     virtual void Drive() { /*override*/ }
9 }
```

实现继承与接口继承

- 实现继承 Implementation inheritance: 普通函数的继承是一种实现继承，子类继承了父类的普通函数，继承的是函数的实现
- 接口继承 Interface inheritance: 虚函数的继承是一种接口继承，子类继承的仅仅是父类虚函数的接口，因此要求返回值类型、函数名和参数列表完全相同。目的就是为了重写以达成多态。所以如果不需要实现多态，就不要把函数定义成虚函数

实现接口

C++ 中，虽然没有像一些其他编程语言（如 Java 和 C#）那样内置的 `interface` 关键字，但是可以通过抽象基类和纯虚函数来实现接口的概念

接口在 C++ 中通常通过以下步骤实现：

1. 创建抽象基类：定义一个抽象基类 Abstract Base Class，该类中声明纯虚函数（没有实现的虚函数）来定义接口的方法。抽象基类不能被实例化，只能被用作其他类的基类

```
1 class Interface {
2 public:
3     virtual void method1() = 0; // 纯虚函数声明
4     virtual void method2() = 0;
5 };
```

2. 派生具体类：从抽象基类派生具体类，实现接口中声明的纯虚函数

```

1  class ConcreteClass : public Interface {
2  public:
3      void method1() override {
4          // 实现 method1 的具体逻辑
5      }
6
7      void method2() override {
8          // 实现 method2 的具体逻辑
9      }
10 };

```

3. 使用接口：可以通过接口类型的指针或引用来访问具体类的对象，实现对接口的使用

```

1  void someFunction(Interface* obj) {
2      obj->method1(); // 调用接口方法
3      obj->method2();
4  }
5
6  int main() {
7      ConcreteClass concreteObj;
8      someFunction(&concreteObj); // 将具体类对象传递给接口
9      return 0;
10 }

```

需要注意的是，C++ 中的接口是通过抽象基类和纯虚函数实现的，并且一个类可以实现多个接口（通过继承多个抽象基类）。在 C++20 及更高版本中，引入了 `concept` 的概念，可以用于更直观地定义和使用接口

多态原理介绍：以单继承为例

总结：满足多态条件时（函数重写+父类指针/引用调用），会产生虚表存储虚函数的指针，调用时取对象自己的虚表里找到重写的多态函数

虚函数表/虚表 virtual function table

- 虚函数在编译好后也是放在公共代码段中，虚表中放的只是虚函数指针
- 虚表的本质是一个虚函数指针数组，一般情况这个数组最后放了一个nullptr
- 虚表地址与重写的关系，这种关系的处理与编译器有关，下面是VS的情况
 - 若完成重写，同一个类的不同对象共用一张虚表
 - 若父子类都完成重写，则父子类的虚表不同，虚函数地址也不同
 - 若父类定义了虚函数，然而子类并没有重写，则父子类的虚表仍然不同，但其存储的虚函数指针是相同的
 - 若子类有父类没有的独立的虚函数，该虚函数的函数指针也会被放进虚表里，但VS中的监视窗口是看不到这个函数指针的；反过来若父类有子类没有就看不到。可以通过设计打印函数来观察

```
class Person
{
public:
    virtual void BuyTicket() { cout << "Full-price" << endl; }
};

class Student : public Person
{
public:
    virtual void BuyTicket() { cout << "Full-price" << endl; }
};

int main()
{
    // 同一个类型的对象共用一个虚表
    Person p1;
    Person p2;
    // VS下，不管是否完成重写，子类虚表跟父类虚表都不是同一个
    Student s1;
    Student s2;
    return 0;
}
```

将子类改为未完成重写

```
class Student : public Person
{
public:
    //virtual void BuyTicket() { cout << "Full-price" << endl; }
};
```

父子类完成虚函数重写

监视 1		
搜索(Ctrl+E)		
名称	值	类型
p1	{...}	Person (A)
_vfptr	0x00889b34 {Lecture2_pol...	void **
[0]	0x008815cd {Lecture2_pol...	void *
p2	{...}	Person (A)
_vfptr	0x00889b34 {Lecture2_pol...	void **
[0]	0x008815cd {Lecture2_pol...	void *
s1	{...}	Student
_vfptr	0x00889b3c {Lecture2_pol...	void **
[0]	0x008815e6 {内部 Lecture...	void *
s2	{...}	Student
_vfptr	0x00889b3c {Lecture2_pol...	void **
[0]	0x008815e6 {内部 Lecture...	void *

将子类改为未完成重写

监视 1		
搜索(Ctrl+E)		
名称	值	类型
p1	{...}	Person (A)
_vfptr	0x00db9b34 {Lecture2_pol...	void **
[0]	0x00db15cd {Lecture2_pol...	void *
p2	{...}	Person (A)
_vfptr	0x00db9b34 {Lecture2_pol...	void **
[0]	0x00db15cd {Lecture2_pol...	void *
s1	{...}	Student
_vfptr	0x00db9b3c {Lecture2_pol...	void **
[0]	0x00db15cd {Lecture2_pol...	void *
s2	{...}	Student
_vfptr	0x00db9b3c {Lecture2_pol...	void **
[0]	0x00db15cd {Lecture2_pol...	void *

- 虚表和菱形继承产生的虚继表是完全不同的东西，注意区别。虚表应对多态问题，虚继表应对菱形继承的数据冗余和二义性问题。当既是菱形继承，又发生了多态时，二者会同时产生，变得非常复杂

多态原理


```

class Base {
public:
    virtual void Func1() { cout << "Func1-Base" << endl; }
    virtual void Func2() { cout << "Func2" << endl; }
    void Func3() { cout << "Func3" << endl; }
protected:
    int _age = 1;
};
class Derived : public Base {
public:
    virtual void Func1()
    { cout << "Func1-Derived" << endl; }
protected:
    int _num = 2;
};

void Test(Base& b)
{
    b.Func1();
}

int main()
{
    Base b;
    Test(b);
    b.Func3();

    Derived d;
    Test(d);

    return 0;
}

```

父类
结构

名称	值	类型
b	{...}	Base {A}
_vfptr	0x00949b34 {Lecture2_polymorphism...	void **
[0]	0x0094159b {Lecture2_polymorphism...	void *
[1]	0x009415af {Lecture2_polymorphism...	void *
_age	1	int
d	{_num=2}	Derived
Base	{_age=1}	Base
_vfptr	0x00949c8c {Lecture2_polymorphism...	void **
[0]	0x009415b9 {内部 Lecture2_polymor...	void *
[1]	0x009415af {Lecture2_polymorphism...	void *
_age	1	int
_num	2	int

构成多态

```

b.Func1();
00F12611 mov     eax,dword ptr [b]
00F12614 mov     edx,dword ptr [eax]
00F12616 mov     esi,esp
00F12618 mov     ecx,dword ptr [b]
00F1261B mov     eax,dword ptr [edx]
00F1261D call    eax
00F1261F cmp     esi,esp
00F12621 call    __RTC_CheckEsp (0F11302h)

```

call的是虚表指针

普通调用

```

b.Func3();
00F128F3 lea     ecx,[b] 已用时间<= 1ms
00F128F6 call    Base::Func3 (0F115C8h)

```

直接call函数地址

- 上图中得到的一些虚表结论
 - 若满足虚函数重写的条件，父类和子类都会存放虚表，由上图绿框所标识，两个虚表的地址不同；若不满足虚函数重写，则不会生成虚表
 - Func1虚函数进行了重写，因此实际上成为了两个不同的函数
 - Func2函数虽然是虚函数，但子类并没有对其进行重写，因此正常地被子类所继承，因此虚表中Func2的地址是相同的
 - Func3函数没有被定义为虚函数，就没有被放入虚表中
 - 用汇编语言看也证实了，构成多态调用Func1时，实际上call的是eax寄存器，其中存放的是虚表的指针；而对于一般函数Func3则直接call函数的地址
 - 满足多态以后的函数并不是在编译时确定的，而是运行起来以后到对象的栈中找到的，所以指向谁调用谁，这也被称为运行时决议 Execution-time resolution；不满足多态的函数地址则在编译时就被确定了，这也被称为编译时决议 Compile-time resolution
- 多态的本质原理：若符合多态的两个条件，那么调用时，会到指向对象的虚表中找到对应的虚函数地址，进行调用。比如在上述代码中当传入d对象时，会调用d对象的栈
- 普通函数调用：编译连接时确定函数的地址，运行时直接调用

- 必须通过父类的指针或者引用调用虚函数原因在于利用了切片原理，由于子类对象继承了父类的结构并把父类结构放在最前面，因此传入子类进行切片后编译器看到的也是父类的结构，但里面的虚表保存的是子类中的虚函数位置，这就产生了传入什么对象就调用什么对象的虚函数。

动态绑定与静态绑定

多继承关系中的虚函数表

打印虚表

```
1  typedef void(*VFPTR)(); // 因为上面的成员函数都是空函数指针，所以对空函数指针重
   定义
2  // 即使虚函数的返回类型不是void，或者有其他参数（这种情况下VS2019也会报错），它的函
   数指针也会被强转为空函数指针
3  void PrintVFTable(VFPTR table[]) {
4      for (size_t i = 0; table[i] != nullptr; i++) { // VS中虚表以nullptr结
   尾，而Linux没有这个设计
5          // for (size_t i = 0; i < 3; i++) // VS中虚表以nullptr结尾，而Linux没
   有这个设计
6          printf("vft[%d]: %p ", i, table[i]);
7          //table[i]();
8          VFPTR pf = table[i];
9          pf();
10     }
11 }
12 int main() { PrintVFTable((VFPTR*)(*(int*)(&s1))); /*...*/ }
```

多继承中的虚函数表

- 多继承中子类中独立的虚函数会按照继承顺序放到第一个继承的父类的虚表中
- Derive的func1对两个父类都进行重写，理论上应该共用一份func1，但两个虚表中的func1地址却不一样。这是因为VS中jump指令对func1进行了多次封装，最后实际上调用的都是同一个func1
- 菱形继承、菱形虚拟继承中的虚函数表：尽量不要写出这样的继承

继承和多态的一些面试问题

- 如何定义一个不能被继承的类

- 方法1: C++98: 1、父类构造函数私有 2、子类对象实例化, 无法调用构造函数

```
1  class A {
2      private:
3          A()
4          {}
5      protected:
6          int _a;
7  };
8
9  class B : public A
10 {};
```

```
11
12 int main() {
13     B b;
14     return 0;
15 }
```

- 方法2: C++11新增了一个final关键字 (最终类) `class A final`, 此时 class A 不能用作基类

- 什么是多态?

- 什么是重载、重写/覆盖、重定义/隐藏?

- 多态的实现原理: 满足多态条件时 (函数重写+父类指针/引用调用), 会产生虚表存储虚函数的指针, 调用时取对象自己的虚表里找到重写的多态函数

- inline函数可以是虚函数吗? 内联函数没有地址不能被放进虚函数, 这和虚函数是互斥的。但是是可以实现的, 若设置为virtual, 编译器会忽略inline设置, 因为inline是一个对编译器的建议性关键字, 优先级很低

- 静态成员可以是虚函数吗? 不可以, 因为静态函数都是编译时决议, 无法访问虚表

- 构造函数可以是虚函数吗? 不可以, 因为对象中的虚表指针是在构造函数初始化列表阶段才初始化的

- 析构函数可以是虚函数吗？什么场景下析构函数是虚函数？可以，且最好把父类的虚表设置为虚函数，形成多态（见重写的例外情况），一个例子就是异常类的析构函数
- 拷贝构造和operator=可以是虚函数吗？拷贝构造不可以，拷贝构造也是构造有初始化列表；operator=可以，但是没有意义，因为赋值的参数需要是同类，这不能完成重写，但若是使用和父类拷贝构造相同的参数，又不能达到子类拷贝的目的，因此没有意义
- 对象访问普通函数快还是虚函数快？若不构成多态，就都是编译时决议，则一样快；若构成多态，则普通函数快，因为多态调用是运行时决议要通过虚表调用
- 虚函数表是在什么阶段生成的？存在哪里？虚表存在内存的常量区，在编译阶段就生成好了。注意不要和虚函数表的指针是在构造函数的初始化列表阶段生成的，此时去常量区找到虚表的地址并把它放到类对象里。

map和set

pair 键值对

```
1  template <class T1, class T2>
2  struct pair
3  {
4      typedef T1 first_type;
5      typedef T2 second_type;
6      T1 first;
7      T2 second;
8      pair(): first(T1()), second(T2()){}
9      pair(const T1& a, const T2& b): first(a), second(b){}
10 };
```

- 将键值对打包成了一个结构体，这样便于解引用操作
- 用 `make_pair` 自动推导生成键值对

set和map的重要接口

根据应用场景的不同，STL总共实现了两种不同结构的关联式容器：树形结构与哈希结构

树形结构的关联式容器主要有四种：map、set、multimap、multiset。这四种容器的共同点式使用平衡搜索树（即红黑树）作为其底层结构

set

lower_bound(val) 返回的是 \geq val的； upper_bound(val) 返回的是 $>$ val的。multiset：允许键值冗余

- find(key) 返回的是中序第一个找到的key
- erase(key) 删除的是所有的符合项
- count(key) 返回key对应项的个数，若不允许冗余，则返回0或1，可以用来判断是否存在

map

- insert
 - `pair<iterator,bool> insert (const value_type& val)`
 - The single element versions (1) return a pair, with its member `pair::first` set to an iterator pointing to either the newly inserted element or to the element with an equivalent key in the map. The `pair::second` element in the pair is set to `true` if a new element was inserted or `false` if an equivalent key already existed.
- 返回类型是first为迭代器的pair是为operator[]准备的，否则就返回一个bool就可以了，没必要返回迭代器
- operator[]
 - 给一个key，若能找到符合的key，返回val&：相当于查找+修改val的功能
 - 若没有符合的，就插入默认构造生成的pair，即 `pair(key, V())`，并返回val&：相当于插入+修改的功能
 - 底层是利用insert实现的

```

1  V& operator[](const K& key)
2  {
3      pair<iterator, bool> ret = insert(make_pair(key, V()));
4      return ret.first->second; //first是迭代器，对first解引用得到pair
   结构体，再取second
5  }

```

- at相比于operator[]就只是查找+修改，若找不到就抛_out_of_range_exce的异常，这和python字典的功能相同
- multimap允许键值冗余，没有operator[]

set和map的模拟实现

泛型编程

map和set的底层都是一棵泛型结构的RBTree，通过不同实例化参数，实现出map和set。对于map而言传的是 <Key, Pair<Key, Value>>，对于set而言传的是 <Key, Key>

```

1  template<class K, class T, class KeyOfT>
2  struct RBTree {}
3
4  template<class K>
5  class set { //...
6  private:
7      RBTree<K, K, SetKeyOfT> _t;
8  }
9
10 template<class K, class V>
11 class map { //...
12 private:
13     RBTree<K, pair<K, V>, MapKeyOfT> _t;
14 }

```

节点的比较大小问题

对于set而言，`curr->_data` 可以直接比较，因为此时的data只是一个Key；但对于map而言，此时的data是一个pair键值对，需要用个仿函数 `KeyOfT` 取出 pair 中的 Key 来实现大小比较

```
1 struct SetKeyOfT
2 {
3     const K& operator()(const K& key) {return key;}
4 };
5
6 struct MapKeyOfT
7 {
8     const K& operator()(const pair<K, V>& kv) {return kv.first;}
9 };
```

迭代器

- set和map的迭代器和list迭代器的封装非常类似，也都使用了泛型编程。set和map的迭代器主要特殊点在于其 `operator++`，`operator--` 操作
- `begin()` 返回的是最左节点，即中序的第一个节点；因为迭代器的范围是左闭右开，所有 `end()` 返回的是 `nullptr`
- `++` 返回的是中序下一个。因为是三叉链所以不需要借助辅助栈，可以直接找父亲
 - 若右子树不为空，`++` 就是找右子树中序下一个（最左节点）
 - 若右子树为空，`++` 找祖先里面当前孩子分支不是祖先的右的祖先。若找到了空的祖先，则说明走完了

```
1 Self& operator++() // 前置
2 {
3     if (_node->_right)
4     { // 右子树不为空就去找右子树的最左节点
5         Node* left = _node->_right;
6         while (left->_left)
7             left = left->_left;
8         _node = left;
```

```

9      }
10     else // 右子树为空就去找祖先里面当前孩子分支不是祖先的右的祖先
11     {
12         Node* parent = _node->_parent;
13         Node* curr = _node;
14         while (parent && curr == parent->_right) // parent为空就说明走
完了
15         {
16             curr = curr->_parent;
17             parent = curr->_parent;
18         }
19         _node = parent;
20     }
21     return *this;
22 }

```

- -- 返回的是中序前一个
 - 若左子树不为空， -- 就是找左子树中序上一个（最右节点）
 - 若左子树为空， -- 找祖先里面当前孩子分支不是祖先的左的祖先。若找到了空的祖先，则说明走完了

```

1 Self& operator--() { // 前置
2     if (_node->_left) {
3         Node* right = _node->_left;
4         while (right->_right)
5             right = right->_right;
6         _node = right;
7     }
8     else {
9         Node* parent = _node->_parent;
10        Node* curr = _node;
11        while (parent && curr == parent->_left) {
12            curr = curr->_parent;
13            parent = curr->_parent;
14        }

```

```
15         _node = parent;
16     }
17     return *this;
18 }
```

unordered_map/unordered_set

- unordered系列容器是C++11新增的，其底层是Hash。在java中叫做 TreeMap/TreeSet & HashMap/HashSet
- 大量数据的增删查改用unordered系列效率更高，特别是查找，因此提供了该系列容器
- 与map/set的区别
 - map和set遍历是有序的，unordered系列是无序的
- map和set是双向迭代器，unordered系列是单向迭代器

用底层哈希桶进行封装

和用红黑树封装map和set一样，unordered_map和unordered_set的底层都是泛型结构的哈希桶，通过不同实例化参数，实现出unordered_map和unordered_set。对于unordered_map而言传的是 <Key, Pair<Key, Value>>，对于unordered_set而言传的是 <Key, Key>

迭代器

unordered系列是单向迭代器（哈希桶是单列表）

一个类型K去做set和unordered_set的模板参数的要求

- set要求能支持小于比较，或者显式提供比较的仿函数
- unordered_set
 - K类型对象可以转换整形取模或者提供转换成整形的仿函数
 - K类型对象可以支持等于比较，或者提供等于比较的仿函数。因为要找桶中的数据

右值引用 rvalue reference (C++11)

左值引用与右值引用

左值 lvalue

对于早期C语言的，左值意味着

1. 它指定一个对象，所以引用内存中的地址
2. 它可用在赋值运算符的左侧

左值的特点是可以取地址，除了**const**不能被赋值外其他可以放到赋值符号左边。因此为了适应const的变化，C标准新增了一个术语：**可修改的左值 modifiable lvalue**，用于标识可修改的对象。也可以称为对象定位值 object locator value

左值和右值的概念是相对于赋值操作而言的，而赋值的操作就是要把值存储到内存上

- 当一个对象被用作左值的时候，用的是对象的身份（在内存中的位置）
- 当一个对象被用作右值的时候，用的是对象的值（内容）

右值 rvalue

```
1 //右值举例
2 10;
3 x + y;
4 fmin(x, y); //function
5 string("hello"); //匿名对象
```

右值不能出现在赋值符号的左边，且不能取地址。右值可以是常量、变量或其他可求值的表达式，比如函数调用

右值是指表达式或临时的、临时性的值，其值只能被读取，不能被修改。右值通常是在表达式求值过程中产生的临时结果。例如，字面值、临时对象、返回右值引用的函数调用等都是右值

个人认为一个更好的理解方式是除常量外，若一个表达式能产生临时变量就可以认为这个表达式是右值，从函数的栈帧建立过程中我们就已经发现了这一点

C++11中对右值的进一步划分

- 内置类型右值：纯右值 pure rvalue
- 自定义类型右值：将亡值 expiring value/xvalue

左值引用

左值引用只能引用左值不可以引用右值

但**const左值可以引用右值**，这样就不会因为左值改变而改变被引用的右值，因为const不能被改变

const左值可以引用右值这个特点在引用传参的时候被用到了，即引用传参时既可以接收左值也可以接收右值

```
1 //左值引用可以引用右值吗：const的左值引用可以
2 //double& r1 = x + y; //错误
3 const double& r1 = x + y;
4 template<class T>
5 void Func(const T& T) {}
```

右值引用

```
1 int&& rr1 = 10;
2 double&& rr2 = x + y;
3 double&& rr3 = fmin(x, y);
```

右值引用只能引用右值，不能引用左值

但是右值引用可以引用move以后的左值

需要注意的是右值是不能取地址的，但是**给右值取别名后却可以取地址了**，因为编译器会给右值开一块空间，此时就可以对别名取地址了

```
1 int&& rr1 = 10;
2 rr1 = 20; //合法
3 int b = 1;
4 int&& rr2 = move(b); //合法
```

右值引用使用场景和意义

引用的核心价值是减少拷贝

左值引用的短板

首先回顾一下左值引用的使用场景：做参数和做返回值左值引用

```
1 string to_string(int val); //返回一个string临时对象，不能传引用返回，只能传值拷贝返回
2 void to_string(int val, string& str);
3
4 vector<vector<int>> generate(int numRows); //返回时要拷贝一个vector<vector<int>>的开销太大了
5 void generate(int numRows, vector<vector<int>>& w);
```

在上面的情境中，当要返回一个临时对象时，是不可以使用传引用返回的，因为栈帧被消灭了

考虑解决方案：全局变量会有线程安全问题，用new的话可能会有内存泄漏问题

但用输出型参数进行改造会又不太符合使用习惯，因为一般只要用外面一个变量接收一下就行了

右值引用和移动构造补齐短板

```
//拷贝构造传统写法
string(const string& s)
:_str(nullptr)
{
    cout << "string(const string& s) -- 深拷贝" << endl;
    _str = new char[s._capacity + 1];
    strcpy(_str, s._str);
    _size = s._size;
    _capacity = s._capacity;
}
// 赋值重载
string& operator=(const string& s)
{
    cout << "string& operator=(string s) -- 赋值深拷贝" << endl;
    string tmp(s);
    swap(tmp);
    return *this;
}
```

```
// 移动构造
string(string&& s)
:_str(nullptr)
, _size(0)
, _capacity(0)
{
    cout << "string(string&& s) -- 移动构造" << endl;
    swap(s);
}
// 移动赋值
string& operator=(string&& s)
{
    cout << "string& operator=(string&& s) -- 移动语义" << endl;
    swap(s);
    return *this;
}
```

直接走一次右值构造

```
bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str;
}

int main() {
    bit::string ret = to_string(-3456);
    return 0;
}
```

左值引用返回，在编译器未优化情况下会进行两次深拷贝构造

```
bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str;
}

int main() {
    bit::string ret = to_string(-3456);
    return 0;
}
```

```
bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str;
}

int main() {
    bit::string ret;
    ret = to_string(-3456);
    return 0;
}
```

拷贝构造+
拷贝赋值

```
bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str;
}

int main() {
    bit::string ret;
    ret = to_string(-3456);
    return 0;
}
```

移动构造+
移动赋值

```
string(string&& s) -- 移动构造
string& operator=(string&& s) -- 移动赋值
```

右值引用的核心：在传值情况下通过移动构造（直接和要消亡的右值进行资源交换）减少深拷贝

右值引用不是像左值引用直接起作用的，而是通过识别右值来提供移动构造起作用的

C++11后的容器及其插入相关操作都支持了右值引用，主要就是解决了拷贝开销很大的问题，解决了传值返回这些类型对象的问题，比如push_back、insert

```
1 vector<string> v;
2 string s1("hello");
3 v.push_back(s1); //左值插入，深拷贝
4 v.push_back(string("world")); //c++11支持了右值引用插入，匿名对象是一个右值
```

模板完美转发 *Perfect forward*

万能引用

万能引用或引用折叠：模板中的 `&&` 不代表右值引用，而是万能引用。既能引用左值（传左值时 `&&` 被折叠为 `&`），也能引用右值

但是最后都是统一成了左值引用

```
1 void Fun(int& x) { cout << "左值引用" << endl; }
2 void Fun(const int& x) { cout << "const 左值引用" << endl; }
3
4 void Fun(int&& x) { cout << "右值引用" << endl; }
5 void Fun(const int&& x) { cout << "const 右值引用" << endl; }
6
7 // 万能引用/引用折叠：t既能引用左值，也能引用右值
8 template<typename T>
9 void PerfectForward(T&& t) {
10     // 完美转发：保持t引用对象属性
11     Fun(std::forward<T>(t));
12 }
13
14 int main() {
15     PerfectForward(10);           // 右值
16
17     int a;
18     PerfectForward(a);           // 左值
19     PerfectForward(std::move(a)); // 右值
20
21     const int b = 8;
22     PerfectForward(b);           // const 左值
23     PerfectForward(std::move(b)); // const 右值
24     return 0;
25 }
```

右值引用；左值引用；右值引用；左值引用；右值引用

完美转发

完美转换 `std::forward<T>(x)` 会保持引用对象的属性，比如list容器中支持了push_back，因为list是一个类模板，里面如果新增加push_back的右值引用版本，那么不论是左值还是右值都会走右值版本，因此用一个完美转发来区分

```
1  template<class T>
2  class A {
3      void push_back(const T& x) {
4          insert(end(), x);
5      }
6
7      void push_back(T&& x) {
8          insert(end(), std::forward<T>(x));
9      }
10 };
```

右值引用带来的新的类默认成员函数

C++11之后默认成员函数变成了8个，增加了移动构造和移动赋值

- 只有在要实现深拷贝的时候才有显式实现这两个成员函数的价值，比如 string、vector、list
- 若不需要深拷贝，则可以自动生成，但自动生成的条件比较苛刻
 - 没有自己实现构造函数，且没有实现析构、拷贝构造、赋值重载中的任何一个（一般要自己实现析构就说明要清理资源的深拷贝，也就要同时实现拷贝和赋值重载），此时编译器才会自动生成一个默认移动构造
 - 默认移动构造会对内置类型按字节拷贝，即浅拷贝；对于自定义类型就要看它是否实现了移动构造，若实现了就调用移动构造，没有实现就用拷贝构造
 - 移动赋值和移动构造的条件和过程一样
 - 若显式提供了移动构造或移动赋值，那么编译器就不会提供拷贝构造和拷贝赋值

C++11 特性总结

统一的列表初始化

列表初始化相当于直接调用了构造函数

```
1  //支持初始化列表的构造函数
2  #include <initializer_list>
3  vector(initializer_list<T> il)
4      :_start(nullptr)
5      , _finish(nullptr)
6      , _end_of_storage(nullptr)
7  {
8      reserve(il.size());
9      for (auto& e : il) {
10         push_back(e);
11     }
12 }
```

map和pair都支持列表初始化

```
1  map<string, string> dict = { { "sort", "排序" }, { "insert", "插入" } };
2  //auto dict{ { "sort", "排序" }, { "insert", "插入" } }; // 这样是错的, 不知道里面自动推导的是pair
```

C++11以后一切对象都可以用列表初始化。但是建议普通对象还是用以前 = 赋值来初始化，容器如果有需求可以用列表初始化

处理类型

类型别名 type alias

C语言提供了用typedef给类型起别名，从而简化一些特别长的自定义类型

C++11规定了一种新的方法，称为别名声明 alias declaration，用关键字using来定义类型别名，比如

```
1 using iterator = _list_iterator<T, Ref, Ptr>;
```

但是给指针这种复合类型和常量起类型别名要小心一点，因为可能会产生一些意想不到的后果

auto 变量类型自动推导和 decltype

```
1 int x = 10;
2 //typeid(x).name() y1= 20; //不能这么写
3 decltype(x) y1 = 20.22;
4 auto y2 = 20.22
```

- auto 可以进行自动变量推导
- decltype 可以推导一个变量的类型，再用推导结果去定义一个新的变量。过程中编译器会尝试分析表达式类型，但不会去计算表达式的值

nullptr

C语言中 NULL 被定义为常量0，因此在使用空指针时可能会出现一些错误。C++中引入了指针空值关键字 nullptr，它的类型是 (void*)0

尾置返回

一些新的关键字

范围 for

范围for的底层就是直接替换迭代器实现，这在之前的STL容器的迭代器实现中已经说明了

final 与 override

这两个关键字在继承与多态部分有使用过，链接：[C++ 11 override 和 final](#)

default 与 delete

- default 强制生成某个内联的默认成员函数，若希望default生成的是非内联的就要在类外定义使用default
- delete 禁止生成某个默认成员函数

```
1 //用delete关键字实现一个只能在堆上创建对象的类
2 class HeapOnly {
3 public:
4     //禁止析构生成，哪里都不能构造类对象
5     ~HeapOnly() = delete;
6 };
7
8 int main() {
9     //自定义类型会调析构，指针不会
10    HeapOnly* ptr = new HeapOnly;
11    return 0;
12 }
```

- 区别
 - default只能用于6个默认类成员函数，delete可以用于任何函数
 - delete必须出现在第一次函数声明

STL库的变化

新增加容器：array、forward_list 以及 unordered 系列

增加array的初衷是为了替代C语言的数组，因为委员会认为C语言的数组由于其越界判定问题所以特别不好，数组的越界写是抽查，所以不一定能查出来；越界读除了常数区和代码区外基本检查不出来

forward_list和list的区别是它是一个单向链表，所以只有正向迭代器，在只需要正向迭代的情况下它的效率可能更高

array容器进行越界判定的方式是和vector类似的 [] 运算符重载函数。所以从越界检查的严格性和安全性角度来看还是有意义的。但因为用数组还是很方便而且用array不如用vector+resize。另一个问题是array和数组都是开在栈上，而vector是在堆上。所以C++11标准新增的array容器就变成了一个鸡肋

可变参数模板

可变参数在C语言中就有了，比如printf的参数就是一个可变参数，底层是用一个数组来接收的。

C++11对可变参数进行了扩展，扩展到了模板中

模板声明

```
1  template<class ...Args>
2  void showList(Args... args) {
3      cout << sizeof...(args) << endl;
4      ////不能这么用
5      //for (int i = 0; i < sizeof...(args); i++) {
6      //    cout << args[i] << " ";
7      //}
8      //cout << endl;
9  }
```

递归函数方式展开参数包

```

1  //0个参数的时候就不能递归调用原函数了，要补充一个只有val参数的函数重载，类似于递归的
   终结条件
2  void ShowList() {
3      cout << endl;
4  }
5  //Args... args 代表N个参数包 (N >= 0)
6  template<class T, class ...Args>
7  void ShowList(const T& val, Args... args) {
8      cout << "ShowList(val: " << sizeof...(args) << " -- 参数包: ";
9      cout << val << ") " << endl;
10     ShowList(args...);
11 }

```

逗号表达式展开参数包

lambda表达式

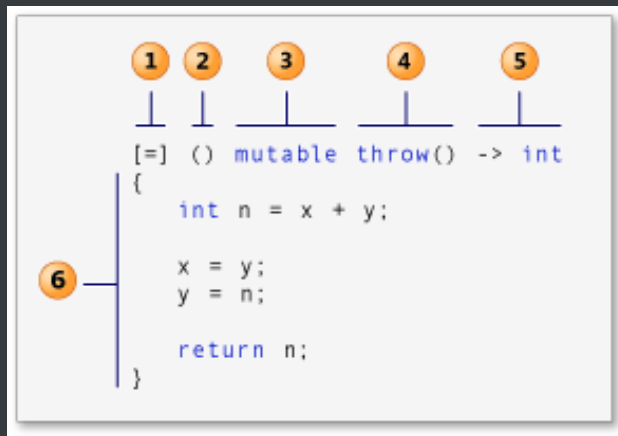
像函数一样使用的对象/类型

- 函数指针
- 仿函数/函数对象
- 表达式函数对象

当类或函数模板需要用到仿函数时，此时可以用lambda表达式替换，因为这样可以将其隐藏到类或函数中封装

可以将lambda表达式理解为一个匿名函数表达式

lambda表达式语法



```

1  //两个数相加的lambda
2  auto add1 = [](int a, int b)->int {return a + b; };
3  cout << add1(1, 2) << endl;
4
5  //省略返回值
6  auto add2 = [](int a, int b){return a + b; };
7  cout << add2(2, 3) << endl;
8
9  //交换变量的lambda
10 int x = 0, y = 1;
11 //auto swap1 = [](int& x1, int& x2)->void {int tmp = x1; x1 = x2; x2 =
    tmp; }; //这样写很难看
12 //swap1(x, y);
13 //cout << x << " " << y << endl;
14
15 auto swap1 = [](int& x1, int& x2)->void {
16     int tmp = x1;
17     x1 = x2;
18     x2 = tmp;
19 };
  
```

[capture-list](parameters)mutable -> return-type {statement} 没有函数名，记法：[](){}

- [capture-list] 捕捉列表：编译器根据 [] 来判断接下来的代码是否为lambda函数，捕捉列表能够捕捉父作用域的变量供lambda函数使用。本质还是传参

- 捕捉方式

- `[var]`：表示以值传递方式捕捉变量var，也就是说拷贝了var，对新var的改变不会改变原来的var
- `[=]`：表示值传递的方式捕捉所有父作用域中的变量（包括this）
- `&var`：表示引用传递捕捉变量var
- `[&]`：表示引用传递捕捉所有父作用域中的变量（包括this）
- `[]`：不捕捉

- 注意点

- 父作用域指的是所有包含lambda函数的语句块
- 允许混合捕捉，即语法上捕捉列表可由多个捕捉项组成，并以逗号分割，比如 `[=, &a]`，以值传递捕捉所以值，但对a采取引用捕捉
- 捕捉列表不允许变量重复传递，否则会编译错误，比如 `[=, a]` 这种捕捉方式，已经以值传递方式捕捉了所有变量了，包括a
- (parameters) 参数列表：和普通函数的参数列表一致，若无参就可以和括号一同省略
- mutable：默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。使用该修饰符时，参数列表不可省略（也就是括号不能省略）
- `->return-type` 返回值类型：没有返回值或返回值类型明确情况下都可以省略，由编译器自动推导返回类型，因此lambda表达式在大多数情况下都不会写返回值类型
- `{statement}` 函数体

lambda底层原理

和范围for底层是迭代器直接替换一样，lambda的底层就是仿函数类。在下图的汇编代码中可以看到，二者的汇编代码结构几乎完全相同

lambda函数对于用户是匿名的，但对于编译器是有名的，其名称就是lambda_uuid

```

// 函数对象
double rate = 0.49;
007D1A3F movsd    xmm0,mmword ptr [__real@3fdf5c28f5c28f5c (07D9BD0h)]
007D1A47 movsd    mmword ptr [rate],xmm0
    Rate r1(rate);
007D1A4C sub      esp,8
007D1A4F movsd    xmm0,mmword ptr [rate]
007D1A54 movsd    mmword ptr [esp],xmm0
007D1A59 lea      ecx,[r1]
007D1A5C call     std::endl<char,std::char_traits<char> > (07D1442h) call构造函数
    r1(10000, 2);
007D1A61 push     2
007D1A63 sub      esp,8
007D1A66 movsd    xmm0,mmword ptr [__real@40c3880000000000 (07D9BD8h)]
007D1A6E movsd    mmword ptr [esp],xmm0
007D1A73 lea      ecx,[r1]
007D1A76 call     std::basic_ostream<char,std::char_traits<char> >::_Sentry_base::_Sentry_base (07D143Dh) operator ()
007D1A7B fstp     st(0)

```

```

// lambda
auto r2 = [=](double monty, int year)->double {return monty * rate * year;
};
007D1A7D lea      eax,[rate]
007D1A80 push     eax
007D1A81 lea      ecx,[r2] lambda_uuid call构造函数
007D1A84 call     <lambda_94df1d8bfa5de2dff5406eebc46e4cf6>::<lambda_94df1d8bfa5de2dff5406eebc46e4cf6> (07D1870h)
    r2(10000, 2);
007D1A89 push     2
007D1A8B sub      esp,8
007D1A8E movsd    xmm0,mmword ptr [__real@40c3880000000000 (07D9BD8h)]
007D1A96 movsd    mmword ptr [esp],xmm0
007D1A9B lea      ecx,[r2] operator ()
007D1A9E call     <lambda_94df1d8bfa5de2dff5406eebc46e4cf6>::operator() (07D18F0h)
007D1AA3 fstp     st(0)

```

包装器 Wrapper

包装器解决的2个问题

可调用对象的种类过多引起的效率问题

C++中可调用对象很多，ret=func(x) 中的 func() 既可以是函数指针，也可以是仿函数对象，还可以是lambda表达式

```

1  template<class F, class T>
2  T useF(F f, T x) {};
3
4  // 函数名
5  cout << useF(f, 11.11) << endl;
6  cout << useF(f, 22.22) << endl;
7  // 函数对象
8  cout << useF(Functor(), 11.11) << endl;
9  // lambda表达式
10 cout << useF([](double d)->double{ return d/4; }, 11.11) << endl;

```

当有如上的用可调用对象作为模板参数的时候，如果直接这么写，即使传入的本质参数和返回值完全相同的不同可调用对象，函数模板也会实例化多份。试验结果如下，当传入不同的可调用对象时，会生成不同的函数模板，前两个传入的都是函数地址，因此使用的是同一份模板。这回导致代码膨胀+效率降低

```

count:1
count: 0x60219c
5.555

count:2
count: 0x60219c
11.11

count:1
count: 0x6021a0
3.70333

count:1
count: 0x602198
2.7775

```

为可调用对象提供了一个统一的参数

上面的 `useF(F f, T x) {};` 是一个函数模板，里面的参数F是一个为可调用对象设计的统一接口，可以往里面放任何的可调用对象。若没有包装器机制，就只能是调固定类型的函数指针、仿函数或者lambda函数了，最尴尬的是当模板参数的类型是lambda可调用对象的时候，甚至连类型都写不了

function包装器

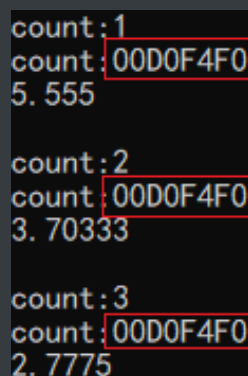
这个问题可以用function包装器来解决。包装器是一个函数专用的类模板，其特点是统一类型，减轻编译器自动推导的压力

```
1 //包装器的头文件和使用方法
2 #include <functional>
3 template <class Ret, class... Args>
4 class function<Ret(Args...)>; //Ret 为被调用函数的返回类型, Args 为调用函数的参数包
```

具体到上面的情况为如下

```
1 // def for f, Functor()
2 std::function<double(double)> func1 = f;
3 cout << useF(func1, 11.11) << endl;
4 // 函数对象
5 std::function<double(double)> func2 = Functor();
6 cout << useF(func2, 11.11) << endl;
7 // lambda表达式
8 std::function<double(double)> func3 = [](double d)->double { return d/4; };
9 cout << useF(func3, 11.11) << endl;
```

此时就可以统一使用一个函数模板了



```
count:1
count:00D0F4F0
5.555

count:2
count:00D0F4F0
3.70333

count:3
count:00D0F4F0
2.7775
```

介绍一个应用：[150. 逆波兰表达式求值 - 力扣 \(LeetCode\)](#)

类成员函数的包装问题

静态成员函数可以直接包装，因为它的参数没有多this指针。但是对于普通函数参数中有一个多出来的this指针需要特殊处理

- 在包装的时候，类非静态成员函数的包装器要多一个不可省略的参数 Plus（C++11规定了传类名，而不是 this），并且还要非静态成员函数的地址，也是C++的规定
- 并且若采用函数名调用，需要在参数中添加一个匿名对象，比如 func5(Plus(), 11.11, 11.11);

```
1  class Plus {
2  public:
3      static int plusi(int a, int b) {
4          return a + b;
5      }
6      double plusd(double a, double b) {
7          return a + b;
8      }
9  };
10 //调用包装器
11 std::function<int(int, int)> func4 = Plus::plusi;
12 cout << useF(func4, 11.11) << endl;
13 std::function<double(Plus, double, double)> func5 = &Plus::plusd; //非静态成员函数要取地址，C++规定，Plus相当于是this指针
14 cout << useF(func5, 11.11) << endl;
15 func5(Plus(), 11.11, 11.11); //直接调用要传入匿名对象
```

std::bind() 解决参数数量、顺序不匹配的问题

在下面这种情况时，因为map数据结构的类型需要的输入包装器已经写死了是要以int为返回值，以两个int为输入，此时无法匹配map带有三个参数的非静态类成员函数的function包装器

```

1 map<string, std::function<int(int, int)>> opFuncMap = { //第二个模板参数必
    须要传入两个参数的
2     {"普通函数指针", f},
3     {"函数对象", Functor() },
4     {"成员函数指针", &Plus::plusi} //报错
5 }

```

此时可以用 `std::bind()` 通用函数适配器来调整可调用对象的参数个数和顺序。`std::bind` 函数是一个函数模板，它会生成一个新的可调用对象

```

1 class Plus {
2 public:
3     Plus(int x = 2)
4         :_x(x)
5     {}
6     int plusi(int a, int b) {
7         return (a + b)*_x;
8     }
9 private:
10    int _x;
11 };
12
13 int main() {
14     std::function<int(Plus, int, int)> func3 = &Plus::plusi;
15     cout << func3(Plus(), 100, 200) << endl;
16
17     std::function<int(int, int)> func4 = std::bind(&Plus::plusi,
18         Plus(10), \
19         std::placeholders::_1, std::placeholders::_2); //绑定参数,
20     //std::placeholder是占位符
21     //调整顺序
22     cout << func4(100, 200) << endl;
23     return 0;
24 }

```

`std::placeholder::_1` 是一个占位符，还剩下几个绑定后的参数就用几个占位符

- 调整参数个数，一般是实际要传入的参数个数比接受的传入参数个数要多

```
1 //绑定两个参数
2 std::function<int(int, int)> func4 = std::bind(&Plus::plusi,
        Plus(10), \
3         100, std::placeholders::_1); //绑定参数, std::placeholder是占位符
```

- 调整参数顺序

```
1         std::function<int(int, int)> func4 = std::bind(&Plus::plusi,
        Plus(10), \
2         std::placeholders::_2, std::placeholders::_1); //绑定参数,
        std::placeholder是占位符
```

绑定参数的原理是相当于先把某些参数传进去了，然后返回一个已经传入部分参数的函数继续接收参数

包装器在TCP server 派发任务中的应用

看计算机网络套接字编程 TCP server 部分

线程库

异常 Exception

C++异常概念

C语言中的错误码

C语言采用的是传统的错误处理机制

- 终止程序，如 `assert`，缺陷是比较粗暴，难以排查，因为只有很严重的错误才应该直接终止程序
- 返回错误码，如 `perror` 等，缺陷是需要程序员自己根据错误码来排查对应的错误

C++异常概念

异常是一种处理错误的方式，当一个函数发现自己无法处理的错误时就可以抛异常，让函数的直接或间接地调用者来处理这个错误

异常体系的三个关键字

- `throw`：当问题出现时，程序会抛出一个异常，则是通过使用 `throw` 关键字来完成的
- `catch`：在用户想要处理问题的地方，通过 `catch` 来捕获异常，可以有多个 `catch`
- `try`：`try` 块中的代码标识符将被激活的特定异常，它后面通常跟着一个或多个 `catch` 块。`try` 块中的代码被称为保护代码

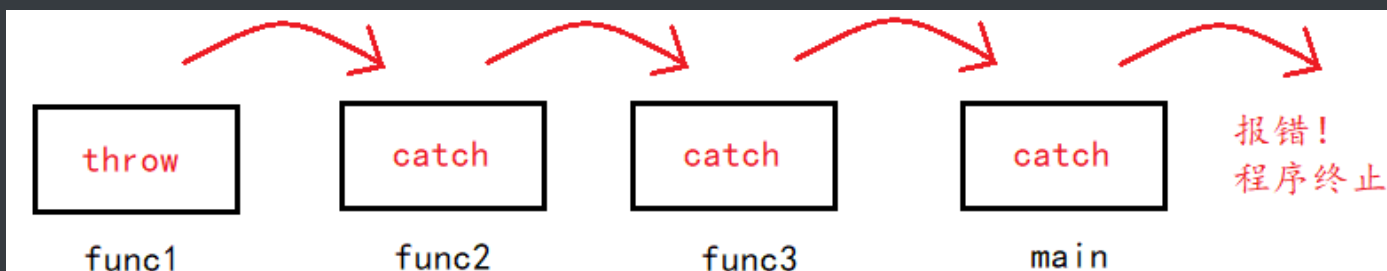
```
1 while (1) {
2     try {
3         Func(); //保护代码里写throw来处理错误
4     }
5     catch (const char* errmsg) {
6         //...
7     }
8     catch (int msgNO) {
9         //...
10    }
11    //...
12    catch (...) { //捕获任意类型的异常，防止出现未捕获异常时，程序直接终止
13        cout << "Unknown exception" << endl;
14    }
15 }
```

异常的使用

异常的抛出和匹配原则

- 异常是通过抛出对象而引发的，该对象的类型决定了应该激活哪个catch的处理代码
- 被选中的处理代码是调用链 **call chain**中与该对象类型匹配且离抛出异常位置最近的那个
- 抛出异常对象后，会生成一个异常对象的拷贝，因为抛出的异常对象可能是一个局部临时对象，类似于传值返回
- `catch (...)` 可以捕获任意类型的异常，问题是不知道异常错误是什么。这是C++中**保证程序健壮性的最后一道底线**，必须要写
- 实际中抛出和捕获的匹配原则有个例外，并不都是类型完全匹配，可以抛出子类对象，使用父类捕获（切片）

在函数调用链中异常栈展开匹配原则



- 首先检查 `throw` 本身是否在 `try` 块内部，如果是的话就再查找匹配的 `catch` 语句，若有匹配的就跳到catch的地方进行处理
- 没有匹配的 `catch` 则退出当前函数栈，继续在调用函数的栈中进行查找匹配的 `catch`
- 若达到main函数栈依旧没有匹配的 `catch` 就报错
- 找到匹配的 `catch` 语句并处理以后，会继续沿着 `catch` 语句后面继续处理

异常的重新抛出

有可能单个的catch不能完全处理一个异常，在进行一些校正处理以后，希望再交给更外层的调用链函数来处理，`catch`则可以通过重新抛出将异常传递给更上层的函数进行处理

```
1 void Func() {
```

```

2      // 这里可以看到如果发生除0错误抛出异常，另外下面的array没有得到释放
3      // 所以这里捕获异常后并不处理异常，异常还是交给外面处理，这里捕获了再重新抛出去
4      int* array = new int[10];
5      int len, time;
6      cin >> len >> time;
7
8      try {
9          cout << Division(len, time) << endl;
10     }
11     catch (...)
12     {
13         cout << "delete []" << array << endl;
14         delete[] array;
15
16         throw; // 捕获什么抛出什么
17     }
18     cout << "delete []" << array2 << endl;
19     delete[] array2;
20 }

```

异常安全

- 构造函数完成对象的构造和初始化，最好不要在构造函数中抛出异常，否则可能导致对象不完整或没有完全初始化
- 析构函数主要完成资源的清理，最好不要在析构函数中抛出异常，否则可能导致资源泄露
- C++中异常经常会导致资源泄漏的问题，比如在 new 和 delete 中抛出了异常，导致内存泄漏（如下面的例子），且处理这种情况很麻烦。或者在 lock 和 unlock 之间抛出了异常导致死锁。C++经常使用RAII来解决以上问题

```

1 void Func() {
2     // 1、如果p1这里new 抛异常会如何?
3     // 2、如果p2这里new 抛异常会如何?
4     // 3、如果div调用这里又会抛异常会如何?
5     int* p1 = new int;
6     int* p2 = new int;
7
8     cout << div() << endl;
9
10    delete p1;
11    delete p2;
12    cout << "释放资源" << endl;
13 }

```

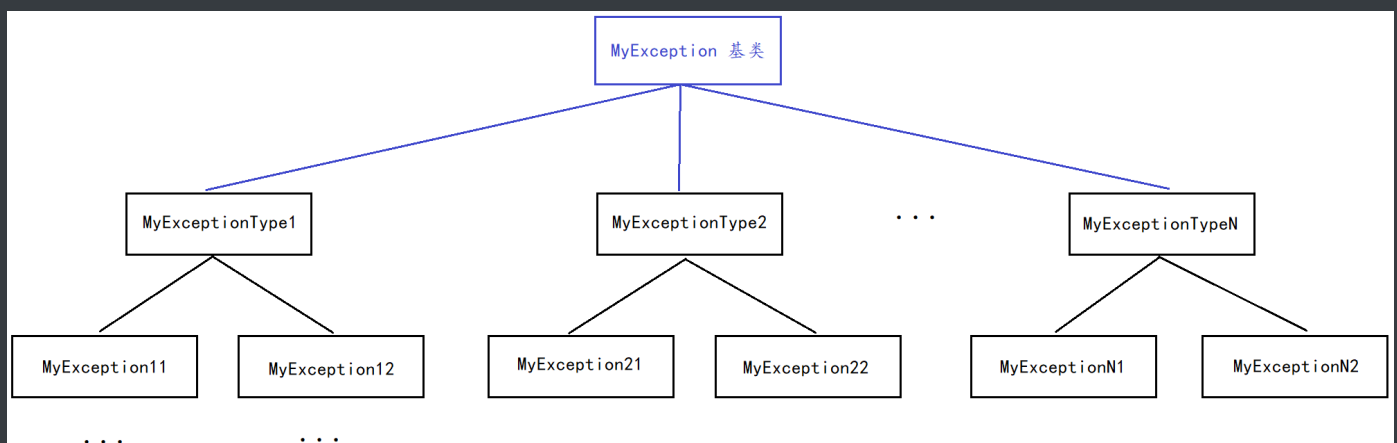
异常规范

异常规范是一种最好遵守的建议，但它不能做到强制程序员遵守，因为C++需要兼容C语言，而C语言中并没有异常体系

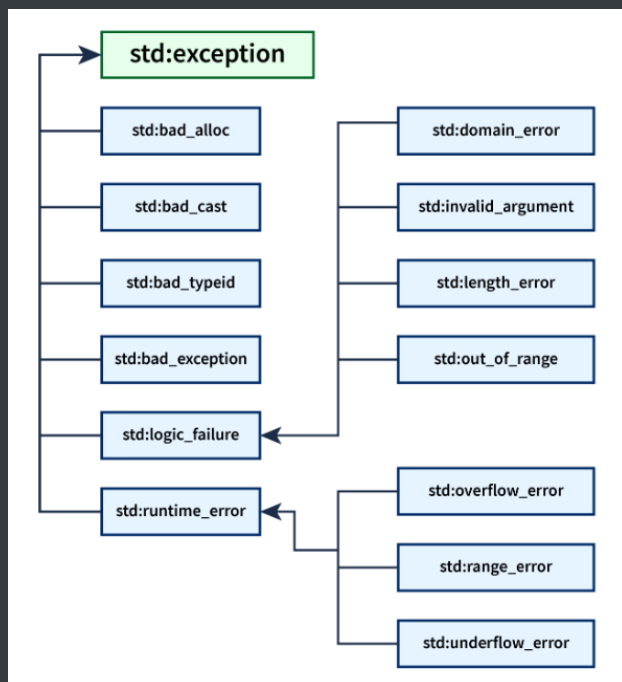
C++11中新增关键字 `noexcept`，表示不会抛异常

异常体系

自定义异常体系



C++异常体系



注意一个点，除零错误不是C++的标准异常，因此如果不throw来try除零的代码，不会抛异常

异常的优缺点

优点

- 比起错误码而言可以展示更丰富的信息，甚至可以包含堆栈调用的信息，帮助用户更好地定位程序bug
- 调用链很深的情况下，可以直接抛异常给外层接受处，不需要层层返回
- 很多的第三方库都包含异常，比如 `boost`，`gtest`，`gmock` 等等常用的库，使用它们的时候也需要使用异常
- 部分函数使用异常更好处理，比如构造函数没有返回值，不方便使用错误码方式处理，比如越界使用异常或者直接 `assert` 终止程序

缺点

- 导致程序地执行流乱跳，非常混乱，有点像 `goto`。程序的运行有时候往往超乎用户想象，此时用比如打断点的方式可能就不能很好的调试程序。这个缺点是最严重的，其他缺点都或多或少有解决方法
- 异常要拷贝对象，有一些多余的性能开销，但这个问题随着硬件发展已经几乎可以忽略

- C++没有垃圾回收机制 Garbage Collection GC，需要用户自己管理资源。有了异常就经常容易造成内存泄漏、死锁等异常安全问题。这个需要使用RAII来处理资源的管理问题
- C++标准库的异常体系定义的不好，导致不同公司、不同项目之间会自定义各自的异常体系，非常混乱
- 虽然C++有异常规范，但由于各种历史原因，规范不是强制的。异常要尽量规范使用，否则会造成严重后果

和Java对比

在C++中，异常处理机制是作为一种可选的语言特性提供的，而不是强制性的。这意味着开发人员可以选择是否使用异常处理机制，并且可以在代码中自由地控制如何处理异常

此外，C++的异常处理机制也有一些局限性和不足之处。例如

1. 异常处理的成本较高：与Java相比，C++的异常处理机制需要更多的时间和空间成本，因为它涉及到堆栈展开和对象销毁等操作，而这些操作可能会导致性能下降
2. 异常的类型不够严格：在C++中，异常可以是任何类型的对象，包括内置类型和自定义类型。这意味着异常的类型不够严格，可能会导致不必要的异常处理或者异常被忽略
3. 异常的语义不够明确：C++的异常处理机制并没有明确定义异常的语义，例如何时应该抛出异常、何时应该捕获异常等问题。这可能会导致开发人员在使用异常处理时出现混淆或错误

因此，尽管C++提供了异常处理机制，但它并没有像Java那样将异常处理视为必不可少的语言特性。在实际开发中，开发人员需要权衡使用异常处理的成本和好处，并根据实际情况选择是否使用异常处理

异常总体而言，利大于弊，所以在大型工程中还是要鼓励使用异常，而且基本所有的面向对象语言都用异常来处理错误

智能指针 Smart Pointer (C++11)

后来发展的面向对象语言因为借鉴了C++缺乏有效资源管理的机制，都发展出了垃圾回收机制。智能指针是C++为了补不设置垃圾回收机制的坑，且垃圾回收对于主程序而言是一个独立的进程，会有一定的性能消耗，C++考虑到性能也就没有采取垃圾回收的方法

但智能指针主要是为了**保证异常安全**，因为异常实际上和goto一样打乱了正常的程序执行流，以前依靠正常的程序执行流来手动delete回收资源的方法现在就很难行得通了

智能指针的使用及原理

RAII思想

RAII Resource Acquisition Is Initialization 资源获取即初始化 是一种**利用对象生命周期来控制程序资源**（如内存、文件句柄、网络接连、互斥量等等）的技术。Java中也会利用这种思想，虽然Java有垃圾回收机制，但同样会面对加锁和解锁时内存资源没有正常释放的问题

在对象构造时获取资源，最后在对象析构的时候析构资源，**不论在任何情况下当对象退出所在的内存空间**，也就是说其生命周期结束后，**一定会调用析构进行清理**，这是由语法定义决定的。**相当于把管理一份资源的责任托管给了一个对象**。这样做有两大好处

- 不需要显式地释放资源
- 采用这种方式，对象所需的资源在其生命周期内始终保持有效

原理与模拟实现

实现主要有3个方面

- RAII行为
- 支持指针操作
- 核心问题是浅拷贝多次析构的问题，解决方法是引用计数

```
1 //利用RAII设计delete资源的类
2 template<class T>
3 class SmartPtr {
```

```

4  public:
5      SmartPtr(T* ptr)
6          :_ptr(ptr)
7      {}
8
9      ~SmartPtr() {
10         cout << "delete: " << _ptr << endl;
11         delete _ptr;
12     }
13     //要支持指针操作行为
14     T& operator*() {
15         return *_ptr;
16     }
17
18     T* operator->() {
19         return _ptr;
20     }
21
22 private:
23     T* _ptr; // 把指针封装进类
24 };
25
26 void Func() {
27     SmartPtr<int> sp1(new int); //若这里new出问题抛异常，那么退出后由类对象析
    构进行处理
28     SmartPtr<int> sp2(new int); //若这里new出问题抛异常，那么sp1和sp2也会调用
    析构处理
29     cout << div() << endl; //若这里出问题，也是一样的
30 }

```

C++标准库提供的智能指针

C++98: `std::auto_ptr`

上面实现的“智能指针”有浅拷贝问题：和迭代器的行为非常类似，都是**故意要浅拷贝**，因为我们**想要用拷贝的指针去管理同一份资源**，但是对上面实现的智能指针就会出现浅拷贝析构问题。而迭代器浅拷贝析构不会报错的原因是因为轮不到迭代器进行析构，迭代器只是封装，容器会将所有的内容一块析构掉

对此 `std::auto_ptr` 的解决方法是**管理权转移**，就是把管理的指针直接交给拷贝对象，然后自己置空。这是一种极为糟糕的处理方式，类似对左值进行了右值处理，直接交换了原指针的资源。会导致被拷贝对象悬空，再次进行解引用就会出现对空指针解引用问题。因此绝大部分公司都明确**禁止使用这个指针类来进行资源管理**

```
1  //管理权转移的实现
2  auto_ptr(auto_ptr<T>& sp)
3  :_ptr(sp._ptr) {
4      sp._ptr = nullptr;
5  }
6
7  auto_ptr<T>& operator=(auto_ptr<T>& ap) {
8      // 检测是否为自己给自己赋值
9      if (this != &ap) {
10         // 释放当前对象中资源
11         if (_ptr)
12             delete _ptr;
13         // 转移ap中资源到当前对象中
14         _ptr = ap._ptr;
15         ap._ptr = NULL;
16     }
17     return *this;
18 }
```

C++11: `std::unique_ptr`

C++11 的 `std::unique_ptr` 是从先行者boost库中吸收过来的，原型是 `scoped_ptr`

```

1 //C++98只能通过声明而不实现+声明为私有的方式来做，但C++11可以用delete关键字
2 unique_ptr(unique_ptr<T>& ap) = delete; //禁止生成默认拷贝构造
3 unique_ptr<T>& operator=(unique_ptr<T>& ap) = delete; //禁止生成默认赋值重载

```

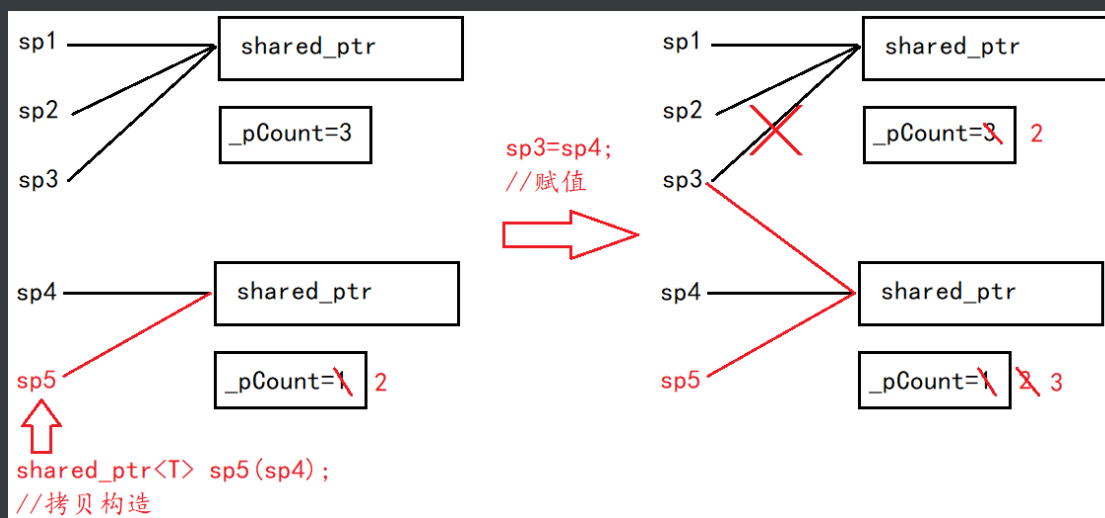
非常的简单粗暴，直接禁止了拷贝构造，但并没有从根本上解决问题。只适用于一些不需要拷贝的场景

C++11: `std::shared_ptr` 的引用计数机制

`std::shared_ptr` 是智能指针和面试中的重点

采取和进程PCB块中的程序计数一样的思想，即引用计数：每个对象释放时，--计数，最后一个析构对象时，释放资源

利用静态成员变量实现是不对的，因为静态变量是属于类的所有对象的，因此在有多个类时会共享一个计数器，这就起不到计数的作用了



```

1 template<class T, class D = Delete<T>>
2 class shared_ptr
3 public:
4     shared_ptr(T* ptr = nullptr)
5         : _ptr(ptr)
6           , _pCount(new int(1)) //给一个计数器
7     {}
8

```

```
9  ~shared_ptr() {
10      Release();
11  }
12
13  void Release() {
14      if (--(*_pCount) == 0) { //给对象赋值是建立在*this目标已经定义的情况下的
15          // 此时计数器至少为1，若没有这步，直接更改指向对象会造成内存泄漏
16          cout << "Delete: " << _ptr << endl;
17          //delete _ptr;
18          D()(_ptr);
19          delete _pCount;
20      }
21  }
22
23  shared_ptr(shared_ptr<T>& sp)
24      :_ptr(sp._ptr)
25      , _pCount(sp._pCount)
26      {
27          (*_pCount)++;
28      }
29
30  shared_ptr<T>& operator=(const shared_ptr<T>& sp) {
31      //防止自己给自己赋值
32      if (_ptr == sp._ptr) {
33          return *this;
34      }
35
36      Release();
37
38      _ptr = sp._ptr;
39      _pCount = sp._pCount;
40
41      (*_pCount)++;
42      return *this;
43  }
44
```

```
45 T& operator*() {
46     return *_ptr;
47 }
48
49 T* operator->() {
50     return _ptr;
51 }
52
53 T* get() { //给weak_ptr使用
54     return _ptr;
55 }
56
57 private:
58 T* _ptr;
59 int* _pCount; //计数器
60 D _del;
61 };
```

std::shared_ptr 的线程安全

std::shared_ptr 的循环引用问题

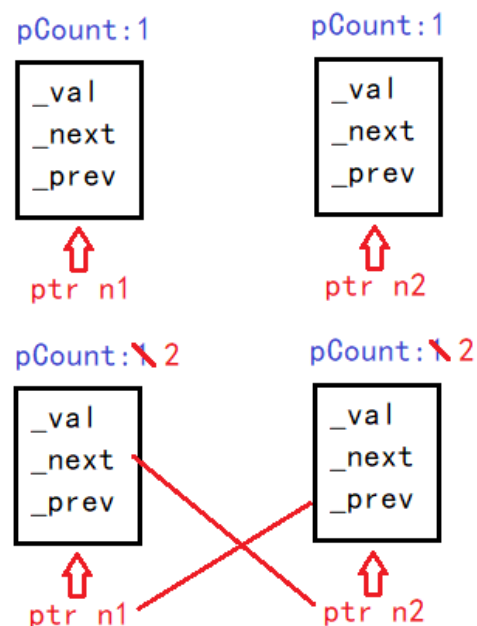
```

struct Node {
    int _val;
    std::shared_ptr<Node> _next;
    std::shared_ptr<Node> _prev;

    ~Node() {
        cout << "~Node()" << endl;
    }
};

//循环引用，没有报错是因为main退出后会自动清理资源
//但很多程序是需要长时间运行的，在这种情况下内存泄漏是很可怕的
void test_shared_ptr2() {
    std::shared_ptr<Node> n1(new Node);
    std::shared_ptr<Node> n2(new Node);
    n1->_next = n2;
    n2->_prev = n1;
}

```



左边空间销毁 → 左边计数器=0 → 管理左边的指针为0 → 右边的 `_prev` 销毁
 左边的 `_next` 销毁 ← 管理右边的指针为0 ← 右边计数器=0 ← 右边的空间销毁

如上图所示，当退出 `test_shared_ptr2()` 时，`n1`和`n2`指针虽然销毁了，但new出来的空间还在，分别被右边的 `_prev` 和左边的 `_next` 管理，此时两个计数器都回到1。然后就产生了一个逻辑矛盾的销毁路径。这个问题被称为循环引用 `circular reference`

该问题用 `std::weak_ptr` 来解决，`std::weak_ptr` 不是常规智能指针，没有RAII，也不支持直接管理资源

`std::weak_ptr` 主要用 `std::shared_ptr` 来构造，因此不会增加计数，**本质就是不参与资源管理**，但是可以访问和修改资源

```

1  template<class T>
2  class weak_ptr { //自己实现，库里的比这个复杂得多
3  public:
4      weak_ptr()
5          :_ptr(nullptr)
6      {}
7
8      weak_ptr(const shared_ptr<T>& sp) //支持对shared_ptr的拷贝构造

```



```

9         :_ptr(sp.get())
10     {}
11
12     weak_ptr(const weak_ptr<T>& wp)
13         :_ptr(wp._ptr)
14     {}
15 }

```

以下是利用 `std::weak_ptr` 解决循环引用问题

```

1  struct Node {
2      int _val;
3      std::weak_ptr<Node> _next; //解决循环引用，不会增加计数
4      std::weak_ptr<Node> _prev;
5
6      ~Node() {
7          cout << "~Node()" << endl;
8      }
9  };
10
11 //循环引用，没有报错是因为main退出后会自动清理资源
12 //但很多程序是需要长时间运行的，在这种情况下内存泄漏是很可怕的
13 void test_shared_ptr2() {
14     std::shared_ptr<Node> n1(new Node);
15     std::shared_ptr<Node> n2(new Node);
16     n1->_next = n2;
17     n2->_prev = n1;
18 }

```

定制删除器

如上面自己实现的 `shared_ptr` 所示，析构的时候其实不知道到底该用 `delete` 或 `delete[]`，甚至有可能数据是用 `malloc` 出来的，为了规范，此时应该要用 `free`。特别是 `[]` 问题，不匹配的结果是很可怕的

因此就要给一个模板，显式传入要用哪种 `delete` 方式

`std::shared_ptr` 不是实现的类模板，而是实现了构造函数的函数模板，这样比较直观也符合逻辑，但这种实现方式是很复杂的，和上面我们自己实现的用类模板不一样

下面给出两个仿函数的例子

```
1  template<class T>
2  struct DeleteArray {
3      void operator()(T* ptr) {
4          cout << "delete" << ptr << endl;
5          delete[] ptr;
6      }
7  };
8
9  template<class T>
10 struct Free {
11     void operator()(T* ptr) {
12         cout << "free" << ptr << endl;
13         free(ptr);
14     }
15 };
16
17 //调用仿函数对象
18 std::shared_ptr<Node> n1(new Node[5], DeleteArray<Node>());
19 std::shared_ptr<Node> n2(new Node);
20 std::shared_ptr<int> n3(new int[5], DeleteArray<int>());
21 std::shared_ptr<int> n4((int*)malloc(sizeof(12)), Free<int>());
```

但大部分情况下，都会直接使用lambda来传

```
1  //lambda
2  std::shared_ptr<Node> n1(new Node[5], [](Node* ptr) {delete[] ptr; });
3  std::shared_ptr<Node> n2(new Node);
4  std::shared_ptr<int> n3(new int[5], [](int* ptr) {delete[] ptr; });
5  std::shared_ptr<int> n4((int*)malloc(sizeof(12)), [](int* ptr)
    {free(ptr); });
```