- C++
  - c/c++源文件如何从代码变成可执行程序的（程序的编译链接）
  - 编译器的编译和解释器的解释的区别
  - 静态链接和动态链接
    - 静态链接
      - 优点
      - 缺点
    - 动态链接
      - 优点
      - 缺点
  - C++ 初始化列表
    - 定义
    - 构造函数的执行阶段
      - 初始化阶段
      - 计算阶段
    - 为什么使用初始化列表
    - 哪些东西必须放在初始化列表中?
  - 内存分配原理
    - 内存分配方式
      - 简介
      - 明确区分堆和栈
      - 堆和栈究竟有什么区别
    - new/delete
      - new/delete 和 new[]/delete[]
      - new, operator new 和 placement new
        - operator new
        - placement new
      - oeprator new 重载
  - C/C++ new malloc
    - 相同点
    - 不同点
      - 类型不同
      - 执行操作不同
      - 返回类型不同

        - 分配失败处理过程
        - 指定内存大小
        - 对数组的处理
        - new 与 malloc 是否可以相互调用
        - 是否可以被重载
        - 是否可以重新分配内存
    - C++ 关键字
        - static
            - 静态全局变量
            - 静态局部变量
            - static 修饰普通函数
            - static 修饰类成员变量（静态成员变量）
            - 静态成员函数
        - explicit
            - 使用注意
        - const
            - const 对象
            - const 引用
            - const 对象的动态数组
            - 指针和const
                - 指向 `const` 的指针
                - `const` 指针
                - 指向 `const` 的 `const` 指针
            - 函数和 const
                - 类中的 const 成员函数（常量成员函数）
                - const 修饰函数参数
            - const 和 static
        - volatile
            - volatile 对象
            - volatile 指针
                - 注意
            - 使用情况
        - mutable
            - 为什么要使用 `mutable` 关键字突破 `const` 限制
        - extern

        - 基本解释
        - extern 出现的问题
        - extern 和 static
        - extern 和const
    - typedef
- C++ 内联函数
    - 什么是内联函数
    - 如何使用内联
    - 内联函数的优缺点
        - 优点
        - 缺点
    - 内联函数与宏定义
- 指针
    - 指针的算术运算
    - 指针、指向指针的指针
    - 指针、数组
    - 指针 函数
    - 指针和 const
- C++ 指针和引用区别
    - 引用
    - 指针
    - 相同点
    - 区别
    - 函数参数传递
        - 指针传递
        - 引用传递
- C++构造函数和析构函数可以抛出异常吗?
    - 构造函数
    - 析构函数
- C++ 类型转换
    - 隐式类型转换
    - 显示类型转换
        - static_cast
        - dynamic_cast
        - reinterpret_cast

        - const_cast
    - 多态性都有那些？静态和动态
        - 静态多态
        - 动态多态
    - 虚函数 纯虚函数 抽象类
        - 前提
        - 虚函数
        - 纯虚函数
        - 抽象类
        - 总结
    - 虚函数表
        - 一般继承（无虚函数覆盖）
        - 一般继承（有虚函数覆盖）
        - 多重继承（无虚函数覆盖）
        - 多重继承（有虚函数覆盖）
    - 操作符重载(`operator`关键字)
        - 为什么使用操作符重载
        - 如何声明一个重载的操作符
            - 操作符重载实现为类成员函数
            - 操作符重载实现为非类成员函数（即全局函数）
        - 如何确定一个操作符重载为类成员函数还是全局函数呢?
        - 运算符重载限制
    - C++ 内存对齐
        - 基本概念
        - 为什么需要内存对齐
        - 对齐规则
        - 示例
    - C++ 模板
        - 模板参数类型
            - 泛化
            - 全特化
            - 偏特化
    - 智能指针
        - 为什么使用智能指针
        - 智能指针分类

        - auto_ptr
        - shared_ptr
        - unique_ptr
        - weak_ptr
    - 简单智能指针的设计和实现
    - C++构造函数
    - C++ 成员初始化顺序
    - C++ 多态性
        - C++ 构造函数和析构函数
        - C++ 拷贝构造函数 参数引用传递还是值传递
- 参考，强烈推荐

C++

c/c++源文件如何从代码变成可执行程序的（程序的编译链接）

1. 编译器的工作过程 http://www.ruanyifeng.com/blog/2014/11/compiler.html
2. C/C++程序编译过程详解 http://www.cnblogs.com/mickole/articles/3659112.html
3. 从源代码到可执行文件——编译全过程解析 http://lxwei.github.io/posts/262.html
4. 【C 语言】编译过程 分析（预处理 | 编译 | 汇编 | 链接 | 宏定义 | 条件编译 | 编译器指示字）http://blog.csdn.net/shulianghan/article/details/78524438

编译器的编译和解释器的解释的区别

1. 你知道「编译」与「解释」的区别吗？http://huang-jerryc.com/2016/11/20/do-you-konw-the-different-between-compiler-and-interpreter/
2.

静态链接和动态链接

静态链接

在链接阶段，将汇编生成的目标文件`.o`与引用到的库一起链接打包到可执行文件中，此种链接方式为静态链接，静态链接使用到的库为静态链接库

静态链接库能够与汇编生成的目标文件一起链接成可执行文件，故相比静态链接库和`.o`文件格式相似，其实静态链接库可看作是一组目标文件`.o`的集合，即很多目标文件压缩打包后形成的文件

优点

1. 方便程序移植，因为可执行文件与静态链接库再无关系，故放在任何环境都可执行
2. 运行效率高，因为使用到的函数都在可执行文件中，无需重定位查找

缺点

1. 空间浪费，因为静态链接要把静态库中的函数或过程链接到可执行文件中，即可执行那个文件包含了运行时所需的全部代码。当多个程序使用相同静态库时，内存中就会存在该库的多个拷贝。
2. 若静态库更新后，则使用它的应用必须全部重新编译。因为静态库中函数要一起打包到可执行文件中，若静态库更新，则需重新编译链接

动态链接

动态链接指在链接阶段，可执行文件不会拷贝所使用到的库函数，而是仅加入所调用函数的信息，比如重定位信息，当可执行文件运行时，才会与相应的动态链接库建立连接关系，根据链接产生的重定位信息调用动态库中的函数，执行其代码。

优点

1. 避免空间浪费， 不同的应用程序如果调用相同的库，则在内存里只有一份该共享库的实例，规避了空间浪费问题
2. 若动态库更新后，只需保持接口不变，则使用库的应用程序也不用改变，解决了静态库更新导致其应用程序页必须更新的问题。

缺点

1. 需要依赖系统环境，动态链接的应用需要系统中存在使用的动态库，若不存在，则会运行失败
2. 运行效率较低，动态链接的应用程序需要在运行过程中动态调用动态库，故执行速度较慢

详细请看 C静态库与动态库 Linux 中的动态链接库和静态链接库是干什么的？ 动态链接和静态链接的区别 静态链接库与动态链接库－－－－C/C **推荐==>** 动态连接和静态连接的区别

C++ 初始化列表

定义

构造函数除了有名字、参数列表和函数体之外，还可以有初始化列表, 初始化列表以冒号开头，后跟一系列以逗号分隔的初始化字段。

构造函数的执行阶段

构造函数的执行可以分为两个阶段，初始化阶段和计算阶段，初始化阶段先于计算阶段

初始化阶段

所有类类型的成员都会在初始化阶段初始化，即使该成员没有出现在构造函数的初始化列表中

计算阶段

一般用于执行构造函数体内的赋值操作

```
struct A
{
    A() { cout << "A constructor" << endl; }
    A(const A& a) { cout << "A Copy constructor" << endl; i = a.i; }
    A& operator=(const A& a) { cout << "A Assignment" << endl; i =
a.i; return *this; }
    ~A() { cout << "A destructor" << endl; }

    int i;
};

struct B
{
    A a;
    B(A& a2) { a = a2; }
};

int main()
{
    A a;  // 调用默认构造函数,  输出 A constructor
    B b(a);  // 初始化阶段：调用默认构造函数初始化对象 a；计算阶段：钓鱼哦该赋
值运算符，赋值操作
}
// output
// A constructor
// A constructor
// A Assignment
```

为什么使用初始化列表

初始化类的成员有两种方式

- 使用初始化列表：使用初始化列表主要是基于性能问题，对于内置类型，如`int`，`float`等，使用初始化类表和在构造函数体内初始化差别不是很大，但对于类类型来说，最好使用初始化列表。因为从上面的测试克制，**使用初始化列表减少了一次调用默认构造函数的过程**，对于数据密集型的类来说，是非常高效的。

```
struct C
{
    A a;
    C(A& a3): a(a3) {}  // 初始化列表：直接调用拷贝构造函数，减少一次默认构造函数调用过程
};
// output: A Copy constructor
```

- 构造函数体内进行赋值操作

哪些东西必须放在初始化列表中?

1. 常量成员：因为常量只能初始化不能赋值，所以必须放在初始化列表中
2. 引用类型：引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
3. 没有默认构造函数的类类型：因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

注意：类成员是按照在类中出现的顺序进行初始化的，而不是按照他们在初始化列表中出现的顺序初始化的

详细请看 C++ 初始化列表

内存分配原理

内存分配方式

简介

C++中，内存分成`5`个区，分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

- **栈**：在执行函数时，函数内局部变量的存储单元都是在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限
- **堆**：就是指由`new`分配的内存块，申请和释放都由编程人员控制，一般一个`new`对应一个`delete`，若申请的内存没释放，则会在程序结束后，操作系统自动回收
- **自由存储区**：就是指由`malloc`分配的内存块，与堆类似，用`free`来释放
- **全局/静态存储区**：全局变量和静态变量被分配到同一块内存，在以前的C语言中，全局变量又分为初始化的和未初始化的，但在C++中没有这种区分，他们共同占用同一块内存
- **常量存储区**：一块比较特殊的存储区，存放的是常量，不允许修改

明确区分堆和栈

```
void f() { int *p = new int[5]; }
```

上述代码同时包含了堆和栈，看到`new`，则应该想到分配了一块堆内存，而指针`p`，则是分配的一块栈内存。上述代码意思是：在栈内存分配了一块内存`p`指向了一块堆内存。而释放内存应该使用`delete []p`， 因为删除的是一个数组，编译器会根据相应的`Cookie`信息进行释放内存的工作

堆和栈究竟有什么区别

1. 管理方式不同： 对于栈来讲，由编译器自动管理，无需我们手工控制；对于堆来说，申请释放都由我们控制，容易产生内存泄漏
2. 空间大小不同： 一般来讲， `32` 位系统下，堆内存能达到 `4G` ，即堆内存几乎没限制。而栈一般都是有一定的空间大小。
3. 能否产生碎片不同：对于堆来说，频繁的 `new/delete` 势必会造成内存空间的不连续，从而造成大量的碎片，程序效率降低。对于栈来说，则不会存在这个问题，因为栈是先进后出的数据结构，进出一一对应，不可能有一个内存块从栈中间弹出，故不会有碎片产生。
4. 生长方向不同：对于堆来说，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来说，生长方向是乡下的，是向着内存地址减小的方向增长。
5. 分配方式不同：堆都是动态分配的，没有静态分配的堆。静态分配都是在编译阶段就确定大小的，而动态分配是在执行过程中分配的。而栈有 `2` 种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 `alloca` 函数进行分配，但是栈的动态分配和堆不同，栈的动态分配由编译器进行释放，无需我们手工控制。
6. 分配效率不同：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 `C/C++` 函数库提供的，机制很复杂，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会获得足够大小的内存，然后返回。显然，堆的效率比栈要低得多。

虽然栈好处众多，但和堆相比不那么灵活，故若分配大量内存空间时，还是用堆比较好。无论是堆还是栈，都要防止越界现象。

new/delete

C++中的 `new operator/delete operator` 就是 `new` 和 `delete` 操作符，
而 `operator new/operator delete` 是函数。

new/delete 和 new[]/delete[]

函数原型

```
void *operator new(size_t);     //allocate an object
void *operator delete(void *);    //free an object

void *operator new[](size_t);     //allocate an array
void *operator delete[](void *);    //free an array
```

```
class A *pa = new A();
```

new operator 原理是

1. 调用 operator new 标准库函数，分配足够大的原始的未初始化的内存空间
2. 运行该类型的相应的构造函数，在上一步分配的内存上初始化对象
3. 返回指向新分配的并构造初始化的对象的指针

```
delete pa;
```

delete operator 原理是

1. 对 `pa` 指向的对象运行析构函数
2. 调用 operator delete 标准库函数释放该对象所用的内存

```
class A *pAa = new A[3];
```

new[] operator 原理是

1. 调用 operator new[] 标准库函数，分配足够大的原始的未初始化的内存，注意要多出 `4` 个字节出来存放数组大小
2. 在刚分配的内存上运行构造函数，对数组中的每个新建的对象进行初始化操作。
3. 返回指向新分配并构造好的对象数组的指针，是对象数组的指针，不包括存放数组带下的内存地址

```
delete []pAa;
```

1. 对对数组中各个对象运行析构函数，数组的维数保存在`pAa`指向的内存空间的前面`4`个字节的内存空间中
2. 调用 operator delete[] 标准库函数释放申请的空间，注意，不仅释放对象数组所占空间，还有上面保存数组大小的`4`个字节的内存空间。

new, operator new 和 placement new

C++里的运算符， 比如`A *a new A;`， 对于`new`来说，是不可以被重载的， 执行过程有

- 调用 operator new 分配内存
- 调用对应构造函数生成对象
- 返回相应指针

operator new

即实现不同的内存分配行为的函数，operator new有多种重载形式，如下所示。当然`operator new`可以被重载，当然`operator delete`、`operator new[]`和`operator delete[]`也可以被重载

```
// exception throwing (1)
void* operator new (std::size_t size) throw (std::bad_alloc);
// exception nothrow (2)
void* operator new (std::size_t size, const std::nothrow_t&
nothrow_value) throw();
// placement (3)
void* operator new (std::size_t size, void* ptr) throw();
```

`(1)(2)`的区别仅在于是否抛出异常，当分配失败时，前者会抛出`bad_alloc`异常，后者返回`null`，不会抛出异常，他们都分配一个固定大小的连续内存。

placement new

`(3)`就是`placement new`，也是对`operator new`的一个重载版本，并不分配内存， 如果你想在已经分配好的内存中创建一个对象，使用`new`是行不通的，也就是说`placement new`允许你在一个已经分配好的内存中构造一个新对象(通过调用对象的构造函数)，原型中`void *ptr`实际上就是指向一个已经分配好的内存缓冲区的首地址。这在内存池上有广泛应用

```
class A
{
public:
    A() { cout << "call A constructor" << endl; }
    ~A() { cout << "call A destructor" << endl; }
};

int main()
{
    A *a = new A;  // new 运算符 分配内存，构件对象，返回该类型指针
    delete a;

    // 以下两行等于 new 运算符
    A *p = (A*)::operator new(sizeof(A));   // 调用 operator new 分配内存，  返回指针
    new(p) A();  // 调用 placement new 构件对象
    // 以下等于 delete 运算符
    p->~A();  // 调用析构函数销毁对象
    ::operator delete(p);  // 释放内存
    return 0;
}
```

oeprator new 重载

前面提到`A *a = new A;`的执行过程：调用 operator new， 分配内存。类可以重载`operator new`， 全局作用域也可以重载`operator new`函数，故此时的重载遵循作用域覆盖原则，即在里向外寻找operator new的重载时，只要找到operator new()函数就不再向外查找，如果参数符合则通过，如果参数不符合则报错，而不管全局是否还有相匹配的函数原型。

相信内容请看

1. C/C++内存管理详解
2. C++中的new、operator new与placement new
3. 浅谈 C++ 中的 new/delete 和 new[]/delete[]
4. C++ 内存分配(new，operator new)详解

C/C++ new malloc

相同点

都可用于申请动态内存和释放内存

不同点

类型不同

`new/delete`是操作符，而`malloc/free`是标准库函数。

malloc 和 free 的函数原型

```
void* malloc( size_t size );
void  free( void* ptr );
```

执行操作不同

new 操作符分配对象内存是会执行三个操作

1. 调用 operator new 函数（对于数组是 operator new[]),分配一块足够大的、原始的未初始化的内存空间
2. 编译器运行相应的构造函数以构造对象，并初始化
3. 对象构造完成后，返回一个指向该对象的指针

delete 操作符释放对象内存时执行两个操作

1. 调用对象的析构函数
2. 编译器调用 operator delete(或 operator delete[])函数释放内存空间

malloc 和 free 只是单纯的申请空间和释放空间，并不会执行相应的对象构造函数和析构函数

返回类型不同

new 操作符分配内存成功后，返回的是对象类型的指针，类型严格与对象夜歌匹配，无需进行类型转换，故 new 是符合类型安全性的操作符

`malloc`内存分配成功则是返回`void *`， 需要通过强制类型转换将`void *`指针转换成需要的类型。

分配失败处理过程

`new` 操作符申请内存分配失败时，将返回 `0` 或引发异常 `bad_alloc` .在 `operator new` 抛出异常之前，会调用客户指定的错误处理函数， 即 `new-handler`

```
//set_new_handler函数
namespace std {
    typedef void (*new_handler)();
    new_handler set_new_handler(new_handler p) throw();  //异常声明,
表示该函数不抛出任何异常
}
main()
{
    std::set_new_handler(outOfMem); //安装内存处理函数
}
```

`set_new_handler` 函数参数是指针，指向 `operator new` 无法分配足够内存时该被调用的函数。当无法满足内存申请时，会不断调用 `new-handler` 函数，直到找到足够内存。更多可参考 **STL源码剖析 分配器**一章

`malloc` 分配失败时返回 `null` 指针，故需要进行判空操作。

指定内存大小

使用 new 操作符申请内存分配时无需指定内存块大小，编译器会根据类型信息自行计算。

malloc 则需要显示指出所需内存的大小

```
class A{...}
A * ptr = new A;
A * ptr = (A *)malloc(sizeof(A)); //需要显式指定所需内存大小sizeof(A);
```

对数组的处理

C++ 提供了 new[] 和 delete[] 专门处理数组类型，使用 new[] 分配的内存必须使用 delete[] 释放

new[] 所执行的操作

1. 调用 operator new[] 标准库函数分配足够大的原始未类型化的内存，注意，要多出`4`字节存放数组大小
2. 在刚分配的内存上运行构造函数对新建的对象进行初始化构造
3. 返回指向新分配并构造好的对象数组的指针

delete[] 所执行的操作

1. 对数组中各个对象运行析构函数，数组的位数保存在数组对象起始地址前`4`个字节中
2. 调用 operator delete[] 标准库函数释放申请空间，注意，不仅仅释放对象数组所占空间，还有上面`4`个字节

```
A * ptr = new A[10];//分配10个A对象
delete [] ptr;
int * ptr = (int *) malloc( sizeof(int)* 10 );//分配一个10个int元素的数组
```

new 与 malloc 是否可以相互调用

operator new 和 operator delete 的实现可以基于 malloc， 而malloc的实现不可以调用 new

是否可以被重载

operator new / operator delete 可以被重载，标准库定义了 operator new函数和 operator delete 函数的 8 个重载版本。

malloc / free 不允许重载

是否可以重新分配内存

`maloc`分配内存后，若使用中发现内存不足，可使用`realloc`函数进行内存重新分配实现内存扩充，`realloc`先判断当前的指针所指内存后是否有足够的连续空间，如果有，原地扩大可分配内存，并返回原来的地址指针。若空间不够先按照指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存空间，而后释放原来的内存空间。

`new`没有类似`realloc`函数的功能

1. 细说new与malloc的10点区别
2. Effective C++ 条款49：了解new-handler的行为

C++ 关键字

static

`static`关键字可用于声明变量、函数、类数据成员和类成员函数。 默认情况下，在所有块外部定义的对象或变量具有静态存储期和外部链接。静态存储期意味着程序开始时分配对象或变量，程序结束时释放对象或变量。外部链接意味着，对象或变量在其他编译单元可见；内部链接意味着，只能在本编译单元可见

静态全局变量

静态全局变量，作用域仅限于当前编译单元中，其他编译单元即使用`extern`声明也无法使用。准确的说作用域从定义处开始，文件结尾处结束。

```
static int j;
int main() {}
```

静态局部变量

函数内部定义的变量，在程序执行到其定义处时，编译器为它在栈上分配空间，函数在栈上分配的空间在此函数结束时会释放。若想将此变量的值保存至下一次调用时，如何实现？最简单的方法是定义其为全局变量，但全局变量有很多缺点，最明显的就是破坏了此变量的访问范围，即不只是受此函数控制。

而在函数内部定义的静态局部变量，作用域被限定在此函数内，而生命周期却是静态存储期，且静态局部变量的内存分配在静态存储区， 故即使该函数运行结束，此变量的值仍然保存

```
void fun(void)
{
    static int i = 0;
    i++;
}
```

static 修饰普通函数

普通函数前加 `static` 使得函数成为静态函数，此时的 `static` 不是指存储方式，而是指函数的作用域仅限于本编译单元，故又称内部函数。好处是，不用担心函数名与其他编译单元中的函数名冲突

```
static void fun(void)
{
    static int i = 0;
    i++;
}
```

static 修饰类成员变量（静态成员变量）

静态成员变量属于类，不属于对象，故即使类生成多个对象，静态数据成员在程序中只分配一次内存，多个对象共享访问，静态数据成员存储在全局数据区，在定义处分配空间。虽然全局变量也可以让多个对象共享访问，但全变量也可以让其他函数或类访问或更改，故安全性较低。

```
class A
{
public:
    static int static_var;
    int var;
};
//int A::static_var;  // 初始化，编译器会给 static 变量默认初始化为 0
//int A::static_var = 4;  // 初始化同时赋值
// 不初始化会链接错误
int main()
{
    A a;
    cout << A::static_var << endl;
    cout << a.static_var << endl;
}
```

初始化的格式为 `数据类型 类名::静态数据成员名 = 值`

1. 静态数据成员是静态存储的，所以必须进行初始化，默认值为 0
2. 静态数据成员初始化在类体外，且签名不加`static`，以区分一般静态变量或对象
3. 初始化时不加该成员的访问权限控制符`private`、`public`等
4. 初始化时使用作用域运算符表明它所属的类
5. 静态数据成员的内存空间既不是在声明类是分配，而不是在创建对象时分配，而是在初始化时分配
6. 静态成员变量可通过类访问，也可通过对象访问

静态成员函数

在类中，`static`还可修饰成员函数，修饰后变为静态成员函数。普通成员函数可以访问所有成员变量，而静态成员函数只能访问静态成员变量。静态成员函数没有`this`指针，故无法对对象中的非静态成员访问

```
class A
{
public:
    static int static_var;
    int var;

    A(int a) : var(a) {}
    static void staic_print() { cout << static_var << endl; }
    void not_static_print() { cout << static_var << " " << var <<
endl; }
};
int A::static_var;  // 初始化，编译器会给 static 变量默认初始化为 0
int main()
{
    A a(10);
    A::staic_print();
//    A::not_static_print();  //  报错
    a.staic_print();
    a.not_static_print();
}
```

1. 静态成员函数中不能调动非静态成员
2. 非静态成员函数中可以调用静态成员
3. 静态成员变量使用前比如初始化
4. 静态成员函数不能声明为虚函数
5. 不能与具有相同参数类型的非静态成员函数同名

详细内容请看 C/C++ 中的static关键字

c++ static的作用，以及static对象在类和函数中区别

笔试知识点：面试常考点static

explicit

在 C++ 中，`explicit`关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以显示的方式进行类型转换

```
class String
{
public:
    String(int n) { cout << "String(int n)" << endl; }
    String(const char*p) { cout << "String(const char *p)" << endl;
}
};
String s1(10);
String s2 = String(10);
String s3 = "hello";
String s4 = 5;  // String(int n),  编译通过，分配 5 个字节的空字符串
String s5 = 'a';  // String(int n),  编译通过，分配int（'a'）个字节的空字
符串
```

s4 和 s5分别把 int 型 和 char 型，隐式转换成了分配若干字节的空字符串，容易令人误解，为了避免这种错误，可使用`explicit`关键字

```
explicit String(int n) { cout << "String(int n)" << endl; }
String s4 = 5;  // 编译不通过，不允许隐式的转换
String s5 = 'a';  // 编译不通过，不允许隐式的转换
```

使用注意

1. `explicit`关键字只能用于类内部的构造函数声明上
2. `explicit`关键字作用域单个参数的构造函数, 或多个参数，其余有默认参数，只有一个无默认参数的构造函数
3. C++中`explicit`关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换

C++关键字explict的详解和使用

const

const 是C++中常用的类型修饰符,常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。

const 对象

`const`修饰符将对象转为常量对象，表示经过`const`修饰的变量值在程序的任意位置都不能被修改

- 常量对象不可被修改，否则会发生编译错误.
- 常量对象必须初始化。
- 类中的`const`对象需要通过初始化列表初始化
- `const`对象默认为文件的局部变量，即在全局作用域声明的`const`对象只能被编译单元使用，其他编译单元不可使用，除非使用`extern`修饰,就可在整个程序中访问`extern`对象

```
const int i = 5, j;  // Error, j 未初始化
i = 6; // Error
class A
{
public:
    A(int i) : a(i) {}
private:
    const int a;
};
// file1.h
extern const int bufSize;
// file1.cpp
extern const int bufSize = 5;
// file2.cpp
extern const int bufSize;  // 使用声明在file1.h中的 bufSize
```

const 引用

`const`引用是指向`const`对象的引用

- `const`引用不可修改所指向的`const`对象
- `const`对象不能被非`const`引用所绑定
- `const`引用可以初始化为不同但相关类型的对象或者右值，比如字面值常量

```
const int ival = 1024;
const int& refVal = ival;  // const 引用绑定 const 对象
int& ref2 = ival;  // 非 const 引用不能绑定 const 对象
const int &r = 42;  // const 引用初始化为 字面值常量

double dval = 3.14;
const int &ri = dval; // 编译器会将上述代码转换为下列代码

int temp = dval;
const int &ri = temp;  // 即编译器创建一个临时变量存储 dval，将 ri 绑定到该临时变量
```

const 对象的动态数组

1. 如果我们在自由存储区中创建的数组存储了内置类型的const对象，则必须为这个数组提供初始化： 因为数组元素都是const对象，无法赋值。实现这个要求的唯一方法是对数组做值初始化。
2. C++ 允许定义类类型的`const`数组，但类必须提供默认构造函数

```
const int *pci_bad = new const int[100];  // Error
const int *pci_ok = new const int[100]();  // OK
const string* pcs = new string[100];  // 此时会调用 string 类的默认构造函数初始化数组元素
```

指针和const

指向`const`的指针

指针指向`const`对象，即指针指向的对象不可改变

- 指向`const`的指针可以不初始化， 但初始化后不能修改所指对象的值
- 不允许将一个`const`对象的地址赋给一个普通的指向非`const`对象的指针
- 允许将非`const`对象的地址赋给指向`const`对象的指针，该独享可修改，只不过不同能过该指针修改

```
const int i = 5;
const int *p = &i;  // 等价于 int const *p = &i;
int *p2 = &i;  // Error, 把 const 对象赋值给非 const 指针会编译错误
int j = 5;
const int *pj = &j;  // 允许将非`const`对象的地址赋给指向`const`对象的指针
```

`const`指针

指针为`const`对象，即指针不可改变，而指针指向的对象不做限制.`const`指针必须被初始化

```
int a = 0;
int* const p = &a;
(*p)++;
```

指向`const`的`const`指针

顾名思义，就是指针的地址和地址所被保存的内容都是不可变的

```
const int* const p = &a;
int const* const p = &a;
```

函数和 const

类中的 const 成员函数（常量成员函数）

- 类中的常量成员函数不能修改数据成员，且不能调用其他非`const`成员函数
- 只有常量成员函数才能操作常量数据成员
- const 可实现函数重载

```
class S
{
private:
    int size;
public:
    S(int s) : size(s) {}
    int GetSize() { return size; }  // 函数重载
    int GetSize() const { return size; }
};
```

const 修饰函数参数

通过实参初始化的形参，用const修饰后，表示函数内不可改变该形参

const 和 static

1. const 定义的常量在超出其作用域后会释放其空间，而static 定义的静态常量在函数执行后不会释放
2. static表示的是静态的，类的静态成员函数、静态数据成员是和类相关的，不是和具体的对象相关
3. C++ 中 static静态成员变量不能在类内部初始化，类内知识声明，定义需在类外。而const成员变量只能在初始化列表中初始化
4. static 成员函数没有this 指针，所以不能直接存取类的非静态成员变量，调用非静态成员函数，且不能被声明为 virtual

详细内容请看 C中const关键字的使用方法，烦透了一遍一遍的搜，总结一下，加深印象！！！， 前面那个比较乱，可以看这个关于C const 的全面总结

volatile

volatile 对象

`volatile`关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器位置的因素更改，比如操作系统、硬件或者其他线程等。遇到该关键字修饰的变量，编译器对访问该变量的代码不进行优化，从而可以提供对特殊地址的稳定访问。当要求使用`volatile`声明的变量的值的时候，系统总是重新从其所在内存读取数据，即使前面的指令刚从该出读取过数据，而且读取的数据立刻被保存。

```
volatile int i = 10;
int a = i;
// 其他代码，并未明确告诉编译器，对 i 进行过操作
int b = i;
```

`volatile`表示 `i`随时可变，每次使用都必须从其内存地址中读取。而一般的优化做法是，编译器发现如果两次读取`i`的代码中间没有对`i`进行操作，会自动把上次读的数据(可能放到了寄存器中)放进`b`中，而不是从其内存地址中重新读。若`i`因为外在因素如多线程、终端等改变，就会导致上次读的数据和内存中的不同，从而出错。

volatile 指针

对于指针， `volatile`和`const`关键字类似，const 有常量指针和指针常量的说法，volatile 也有相应的概念

- 修饰由指针指向的对象、数据是`volatile` 的：`volatile char* vpch;`
- 指针自身的值——一个代表地址的整数变量，是`volatile` 的：`char* volatile pchv;`

注意

1. 可以把一个非volatile int赋给volatile int，但是不能把非volatile对象赋给一个volatile对象。
2. 除了基本类型外，对用户定义类型也可以用volatile类型进行修饰。

使用情况

volatile用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加volatile；
2. 多任务环境下各任务间共享的标志应该加volatile；
3. 存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；

C/C++中volatile关键字详解

mutable

在C++中， `mutable` 是为了突破 `const` 的限制而设置的，被 `mutable` 修饰的变量，将永远处于可变的状态，即使在 `const` 函数中，甚至结构体变量或者类对象为 `const` ，其 `mutable` 成员也可被修改

```
struct A
{
    int a;
    mutable int b;
};
const A a = {1, 2};
a.a = 11;  // Error 编译错误
a.b = 22;  // OK
```

为什么要使用 `mutable` 关键字突破 `const` 限制

通过 `const` 关键字可以避免在函数中错误的修改了类的状态，而 `mutable` 则是为了突破 `const` 的限制，使得类的一些次要的或者辅助性的成员变量可被修改，
即 `mutable` 修饰的变量可能不算对象内部状态，修改它并不影响 `const` 语义

C/C++要点全掌握（五）——mutable、volatile

extern

基本解释

1. `extern` 关键字可以置于变量或者函数前，表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时要去其他模块中寻找其定义。比如 `extern int i;` ，声明了在变量可在全局范围内使用，但这只是声明，而不是定义。
2. `extern C` 表示进行链接指定，即与 `"C"` 连用时，如 `extern "C" void fun(int a, int b);` 告诉编译器在编译 `fun` 这个函数时要按照 `C` 的规则去翻译相应的函数名，而不是 `C++` 的，因为C++为了支持重载，会把函数名以一定的规则进行转变，而C语言不支持重载，故不会转变。

extern 出现的问题

1. **`extern` 变量**：比如在一个源文件中定义了`char a[6]`，而若想在另一个文件中引用，则要声明`extern char a[]`，如果写成`extern char *a`就错了，**注意使用`extern`要严格对应声明时的格式**
2. **`extern` 变量**：如果在头文件中把全局变量的声明和定义放一块，就会出错。比如

```
// test.h
extern char g_str[] = "123456";
// test1.cpp
#include "test.h"
// test2.cpp
#include "test.h"
```

就会报错， 因为`test1.cpp`和`test2.cpp`都包含了`test.h`，表示变量`g_str`被定义了两次。**注意，只在头文件中做声明，不定义**

3. **`extern` 函数原型**：当函数提供方单方面修改了函数原型时，若使用方继续使用，编译时不会出错，但运行时，就会报错。解决办法是不用`extern`，而是在头文件中提供声明，使用方`include`头文件

extern 和 static

1. `extern`表示该变量在其他模块已经被定义，本模块知识使用
2. `static`表示静态变量，分配内存，存储在静态区，不存储在栈上。表示该变量只允许本模块使用，即作用域只是本编译单元。
3. `static`和`extern`不能同时修饰一个变量

extern 和const

C++中const修饰的全局常量据有跟static相同的特性，即它们只能作用于本编译模块中，但是const可以与extern连用来声明该常量可以作用于其他编译模块中.所以当const单独使用时它就与static相同，而当与extern一起合作的时候，它的特性就跟extern的一样了

详细内容请看 C/C++中extern关键字详解

typedef

1. 定义一种类型的别名，而不只是简单的宏替换，可用作同时声明指针型的多个对象。

```
char *pa, pb;  // 若想声明 pa, pb都为 char* 指针，则此语句错误

typedef char* pchar;
pchar pa, pb;  // pa, pb 都是 char* 指针
```

2. 旧代码中用来帮助 `struct`， 旧代码中声明`struct`对象时，需要加上`struct`

```
typedef struct A{};
A a;  // 有了 typedef，不需要写 struct A a;
```

3. 可用来定义与平台无关的类型， 因为typedef是定义了一种类型的新别名，不是简单的字符串替换，所以它比宏来得稳健（虽然用宏有时也可以完成以上的用途）。

```
typedef long double REAL;  // 比如标准库的 size_t
```

4. 为复杂的声明定义一个新的简单别名

```
int *(*a[5])(int, char*);  // a 右边为[]运算符，表示 a 是一个具有 5 个元素的数组； a 左边的 *，表示 a 的元素是 指针；跳出括号后，右面是括号，表示 a 数组元素是函数指针，指向 int* (int, char*) 函数，返回类型是 int*，参数为 int, char*

typedef int *(*pFun)(int, char*);
pFun a[5];  // 愿声明的简化版
```

相信内容请看 C/C++ typedef用法详解（真的很详细）

C++ 内联函数

什么是内联函数

内联函数是`C++`的增强特性之一，用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联，编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段。当然，内联函数仅仅是对编译器的内联建议，编译器可根据情况选择是否内联

如何使用内联

1. 首先要确定`inline`是实现修饰符，故关键字`inline`必须与函数定义体放在一起才能使函数成为内联，仅将`inline`放在函数声明前面不起任何作用
2. 类中定义的函数默认都是`inline`函数
3. 内联函数应该在头文件中定义,这一点不同于其他函数。编译器在调用点内联展开函数的代码时，必须能够找到 inline 函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}

cout << max(a, b) << endl;  // 编译阶段会展开成 cout << (a > b ? a :
b) << endl, 从而消除函数的额外开销

class A
{
public:
    void Foo(int x, int y) {} // 自动称为内联函数
};
```

内联函数的优缺点

优点

1. 避免函数调用所带来的额外开销，比如变量弹栈、压栈、函数执行结束返回原现场等的开销

缺点

1. 因为代码扩展，导致可执行程序的体积增大
2. 内联函数是在编译阶段展开，故若发生变化，需重新编译
3. 某些系统比如嵌入式不允许体积很大的可执行程序，故内联函数不适用

内联函数与宏定义

1. 宏定义可能会导致意想不到的错误

```
// 优先级引出的错误
#define MAX(a, b) (a) > (b) ? (a) : (b)
result = MAX(2, 1) + 2;  // 预处理器会扩展为 result = (2) > (1) ? (2) :(1) + 2;, 故结果为 2

result = MAX(i++, j);  // 预处理器扩展为 result = (i++) > (j) ? (i++) : (j); 同一个表达式中, i 求值两次
```

2. 宏定义不可调试，但内联函数可被调试。内联函数虽然也是像宏定义一样进行代码展开，但内联函数的可调式指的是在程序的`Debug`模式下，没有真正内联，编译器会像普通函数那样为其生成含有调试信息的可执行代码。而在程序的`Release`版本中，编译器才会实施真正的内联。
3. 在C++中，宏定义无法操作类的私有数据成员，而类的内联函数作为成员函数，可以
4. 编译器调用内联函数时，会进行类型安全检查，或者自动类型转换，正确后才会替换函数调用语句，而预处理器并不能进行类型安全检查和自动类型转换。故内联函数比宏定义安全

详细内容请看 强推 c++ 内联函数（一看就懂）

指针

程序中通过变量名直接操作内存单元的方式称为**直接存取**，而通过指针操作其指向的内存单元的方式为**间接存取**。指针就是用来存放对象地址的对象，指针的值为另一对象的地址。指针一般分为指针、指向指针的指向、数组指针、函数指针等

```
int p; //这是一个普通的整型变量
int *p; //首先从P 处开始,先与*结合,所以说明P 是一个指针,然后再与int 结合,说明指针所指向的内容的类型为int 型.所以P是一个返回整型数据的指针
int p[3]; //首先从P 处开始,先与[]结合,说明P 是一个数组,然后与int 结合,说明数组里的元素是整型的,所以P 是一个由整型数据组成的数组
int *p[3]; //首先从P 处开始,先与[]结合,因为其优先级比*高,所以P 是一个数组,然后再与*结合,说明数组里的元素是指针类型,然后再与int 结合,说明指针所指向的内容的类型是整型的,所以P 是一个由返回整型数据的指针所组成的数组
int (*p)[3]; //首先从P 处开始,先与*结合,说明P 是一个指针然后再与[]结合(与"()"这步可以忽略,只是为了改变优先级),说明指针所指向的内容是一个数组,然后再与int 结合,说明数组里的元素是整型的.所以P 是一个指向由整型数据组成的数组的指针
int **p; //首先从P 开始,先与*结合,说是P 是一个指针,然后再与*结合,说明指针所指向的元素是指针,然后再与int 结合,说明该指针所指向的元素是整型数据.由于二级指针以及更高级的指针极少用在复杂的类型中,所以后面更复杂的类型我们就不考虑多级指针了,最多只考虑一级指针.
int p(int); //从P 处起,先与()结合,说明P 是一个函数,然后进入()里分析,说明该函数有一个整型变量的参数,然后再与外面的int 结合,说明函数的返回值是一个整型数据
Int (*p)(int); //从P 处开始,先与指针结合,说明P 是一个指针,然后与()结合,说明指针指向的是一个函数,然后再与()里的int 结合,说明函数有一个int 型的参数,再与最外层的int 结合,说明函数的返回类型是整型,所以P 是一个指向有一个整型参数且返回类型为整型的函数的指针
int *(*p(int))[3]; //可以先跳过,不看这个类型,过于复杂从P 开始,先与()结合,说明P 是一个函数,然后进入()里面,与int 结合,说明函数有一个整型变量参数,然后再与外面的*结合,说明函数返回的是一个指针,,然后到最外面一层,先与[]结合,说明返回的指针指向的是一个数组,然后再与*结合,说明数组里的元素是指针,然后再与int 结合,说明指针指向的内容是整型数据.所以P 是一个参数为一个整数据且返回一个指向由整型指针变量组成的数组的指针变量的函数.
```

指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的，以单元为单位

```
int a[20];
int *ptr = a;
ptr++;
```

在上例中，指针ptr 的类型是int*,它指向的类型是int，它被初始化为指向整型变量a。接下来的第3句中，指针ptr被加了1，编译器是这样处理的：它把指针ptr 的值加上了sizeof(int)，在32 位程序中，是被加上了4，因为在32 位程序中，int 占4 个字节。由于地址是用字节做单位的，故ptr 所指向的地址由原来的变量a 的地址向高地址方向增加了4 个字节

指针、指向指针的指针

指针存放着其他对象的地址，而指向指针的指针存放着指针的地址。通过指针指向的地址，可以间接操作其指向对象。

```
    int a = 3;  // int 对象
    cout << "a's value is " << a << "! " << "a's address is " << &a
<< endl;

    int *pa = &a;  // int* 指针，指向 int 类型对象
    cout << "pa's value is " << pa << "! " << "pa's address is " <<
&pa << endl;

    int **ppa = &pa;  // int **指针，指向 int* 类型指针
    cout << "ppa's value is " << ppa << "! " << "ppa's address is "
<< &ppa << endl;
    // 输出
    a's value is 3! a's address is 0x7ffc04e701f4
    pa's value is 0x7ffc04e701f4! pa's address is 0x7ffc04e701f8
    ppa's value is 0x7ffc04e701f8! ppa's address is 0x7ffc04e70200
```

指针、数组

数组名表示数组本身， 如果把数组名看成指针的话，它指向数组的首地址。

```
    int arr[3] = {1, 2, 3}; // 一维数组;
    cout << arr << sizeof(arr);  // 输出 数组首地址，以及数组整体占用空间，12字节

    int *parr = arr;
    cout << parr << *parr;  // 输出 数组首地址，以及数组中第一个元素 1

    cout << parr + 1 << *(parr + 1); // 输出数组元素 2 的地址，以及 元素 2

    int arr2[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int (*dp)[3] = arr2;  // 指针 dp 类型为 int(*)[3]
    cout << dp << *dp; // 输出数组首地址，以及数组第一行首地址，不过两地址相同，1
    cou << dp + 1 << *(dp + 1) << **(dp + 1); // 输出数组第二行首地址，以及数组第二行第一个元素地址，不过两地址相同，4
```

由上可知,针对指针的算术运算，增加的都是`sizeof(指针所指对象)`， 比如`parr`，所指对象类型为`int`， 故`+1`，增加`4`个字节.指针`dp`， `+1`操作，增加了`12`个字节，因为指针指向的对象类型为`int (*)[3]`表示含有`3`个`int`元素的数组

数组名退化成指针的情况

1. 数组名作为变量赋值时`int *p = a`,`a`为数组名
2. 数组名作为参数传递时，会降级为指针
3. 数组名作为函数返回类型时

指针 函数

可以把指针声明成为一个指向函数的指针，函数指针指向的是函数而非对象，和其他指针一样，函数指针指向某种特定类型。函数的类型由它的返回类型和形参类型共同决定，与函数名无关。例如

```
int fun1(char *,int);
int (*pfun1)(char *,int);
pfun1=fun1;
int a=(*pfun1)("abcdefg",7); //通过函数指针调用函数。
```

指针和 const

请看上面的 `const` 知识点

详细内容请看 Pointer

C++ 指针和引用区别

引用

引用就是某一对象（变量）的别名，它在逻辑上不独立，它的存在具有依附性。所以引用必须一开始就被初始化，而且在整个生命周期中不能改变引用对象

指针

从概念上将，指针从本质上就是存放变量地址的一个变量，逻辑上是独立的，可以被改变，包括其所指向地址的改变和其指向地址中存放数据的改变

相同点

1. 都是有关地址的概念，指针指向一块内存，存放着一块内存的地址。而引用则是某块内存的别名
2. 都可以用于函数参数和返回值的传递

区别

1. 指针是一个对象(`C++ Primer`中说对象是一块能存储数据并具有某种类型的内存空间)， 而引用则只是一个别名
2. 引用只能在定义时被初始化一次，之后不可改变引用对象；指针可变，可更改指向对象；引用从一而终，指针可以见异思迁
3. 引用没有`const`， 指针有`const`， `const`指针不可更改指向对象。(具体指没有`int& const a`这种形式，而`const int& a`是有的。前者指引用本身不可改变，这是引用的属性，故不需要这种形式；后者指引用所指的值不可改变)
4. 引用不能为空，引用为对象别名，对象不存在，怎么会有别名。 而指针可以为空，所以使用指针之前要进行判空操作
5. `sizeof`引用， 得到的是所指向变量(对象)的大小，而`sizeof`指针得到的是指针本身的大小
6. 指针和引用的自增(`++`)运算意义不一样，指针自增是改变了内存地址`+1`， 而引用是对所指对象的值`+1`
7. 引用比指针安全，而指针的指向很灵活，容易产生野指针，比如多个指针指向一块内存，free掉一个，别的指针就成了野指针。

函数参数传递

指针传递

指针传递参数本质上是值传递的方式，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。

引用传递

在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址（指针传递参数时，指针中存放的也是实参的地址，但是在被调函数内部指针存放的内容可以被改变，即可能改变指向的实参，所以并不安全，而引用则不同，它引用的对象的地址一旦赋予，则不能改变）。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

详细内容请看 C++中引用，指针，指针的引用，指针的指针

C++构造函数和析构函数可以抛出异常吗?

构造函数

1. **构造函数中抛出异常将会导致析构函数无法执行，但对象已申请到的内存资源将会被系统释放，按申请顺序逆序释放**：C++仅能delete掉完全构造的对象，只有对象的构造函数完全运行完，才能完整的构造该对象。所以如果构造函数抛出异常，则该对象只会构造一部分，析构函数将不会被运行
2. **构造函数抛出异常可能会导致内存泄漏**：也是由于析构函数无法被调用，可能导致内存泄漏或资源未被释放。

析构函数

1. 不要在析构函数中抛出异常，虽然C++并未禁止析构函数抛出异常。但这样会导致程序过早结束或出现未知行为。
2. 如果析构函数非要抛出异常，则可使用`try catch`吞下。

详细请看 是否能在构造函数，析构函数中抛出异常?

C++ 类型转换

隐式类型转换

1. 标准转换支持数值类型，bool以及某些指针之间相互转换。注意：某些转换可能会导致精度丢失，比如从 long 转换到 int。
2. 可被单参调用（只有一个参数或多个参数但至少从第二个参数起均带有缺省值）的构造函数或隐式类型转换操作符也会引起隐式类型转换。隐式类型转换会导致问题，故C++提供了关键字`explicit`规避隐式类型转换。

```
class B
{
public: B() {}
public: B (A a) {}
public: B (int c, int d = 0){}
public: operator double() const{ return 5.0; }  // 实现隐式类型转换操作符
};

int main()
{
    A a;
    B b1 = a;  // 调用B(A a)构造函数
    B b2 = 10;  // 调用 B(int c, int d = 0) 构造函数
    B b3;
    double d;
    d = 10 + b3;  //
    cout << d << endl;  // 输出 11
}
```

显示类型转换

static_cast

用法 `static<type-id>(expression)`， 该运算符将 `expression` 转换成 `type-id` 类型，但没有运行时类型检查来保证转换的安全性, 且不能转换掉 `const`, `volitale` 属性， 主要有如下几种用法

1. 用于类层次结构中基类和派生类之间指针或引用的转换
    - 进行上行转换（派生类指针或引用转换为基类表示）是安全的
    - 进行下行转换（基类指针或引用转换为派生类表示），由于没有动态类型检查，故不安全
2. 用于基本数据类型之间的转换，比如把 `int` 转换成 `char`，安全性也要开发人员保证
3. 把空指针转换成目标类型的空指针
4. 把任何类型的表达式转换成 `void` 类型

dynamic_cast

用法：`dynamic_cast<type-id>(expression)`，该运算符把`expression`转换成`type-id`类型的对象，`type-id`必须是类的指针、引用或者`void*`如果`type-id`是类指针类型，那么`expression`也必须是指针，引用同理。被用于安全地沿着类的继承关系向下进行类型转换，能用dynamic_cast把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用，而且你能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时）。其使用运行时信息（RTTI）来进行类型安全检查。

1. 用于类层次见的上行转换和下行转换，上行转换类似于`static_cast`，下行转换时，具有类型检查功能，比`static_cast`更安全，但效率要差一些，注意下行转换时，基类必须要有虚函数才能进行下行转换。
2. 用于类之间的交叉转换，即将类指针或引用转换成其兄弟类的指针或引用

reinterpret_cast

用法：`reinterpret_cast<type-id>(expression)`， 一般用于底层代码，平台移植性较差， 用来处理无关类型之间的转换；它会产生一个新的值，这个值会有与原始参数（expressoin）有完全相同的比特位。按照reinterpret的字面意思“重新解释”，即对数据的比特位重新解释。

1. 用来执行低级转型，比如将`int*`转换为`int`。转换结果与平台息息相关，不具有可移植性
2. 转换函数指针类型，比如将一种类型的函数指针转换为另一种类型的函数指针，但可能导致不正确的结果。

const_cast

用法`const_cast<type-id>(expression)`，用来修改类型的`const`或`volitale`属性。除了`const`和`volitale`之外，`type-id`和`expression`的类型应该是一样的。

1. 常量指针被转换成非常量指针，并且仍指向原来的对象，引用同理
2. 常量对象转换成非常量对象

```
class B
{
public:
    int m_iNum;
}

void foo()
{
    const B b1;
    b1.m_iNum = 100; //comile error
    B b2 = const_cast<B>(b1);  // 使用 const_cast 把常量对象转换成非常对象
    b2. m_iNum = 200; // fine, 其内成员可被改变
}
```

详细内容请看 C++ 中四种强制类型转换的区别 [深入理解C++(一)]类型转换（Type Casting)

多态性都有那些？静态和动态

C++ 的多态性分为编译器多态和运行期多态，也被称为静态多态和动态多态。

静态多态

> 编译期多态， 即在编译时期确定的一种多态性，主要体现在 C++ 中的函数模板，注意函数重载和多态无关，那么函数模板如何体现编译期多态的呢？请看例子

```
template <typename T>
T add(T a, T b)
{
    return a + b;
}

main()
{
    int i1 = 1, i2 = 2;
    cout << add(i1, i2) << endl;  // 输出 3

    double d1 = 1.1, d2 = 2.2;
    cour << add(d1, d2) << endl;  // 输出 3.3
}
```

> 即函数模板 add 在使用的时候才会根据参数的数据类型推断出调用哪种数据类型的 add 函数， 也就是在编译期，编译器确定函数模板的参数类型

动态多态

动态多态就是指在程序运行的时候动态绑定所调用的函数，动态的找到调用函数的入口地址，从而确定调用函数。C++中一般使用虚函数来实现动态多态，请看例子

```
class Parent
{
public:
    virtual void eat(){
        cout << "Parent eat" << endl
    }

    void drink(){
        cout << "Parent drink" << endl;
    }
};

class Child : public Parent
{
public:
    void eat(){
        cout << "Child eat" << endl
    }

    void drink(){
        cout << "Child drink" << endl;
    }

    void play(){
        cout << "Child play" << endl;
    }
};

main(){
    Parent *pa = new Child();  // 父类指针指向子类
    pa->eat();  // 输出 Child eat
    pa->drink();  // 输出 Parent drink, 因为父类中 drink 不是虚函数
    pa->play();  // 编译失败, 无法调用父类中没有的方法
}
```

> C++ 中只能使用 指针 或 引用 来实现多态，不能通过普通对象来实现。由于父类中 eat 函数为虚函数，故在运行期间，会找到动态绑定到父类指针上的子类对象，即运行期间找到了子类对象中 eat 函数的入口地址，然后调用子类的 eat 函数，这就是运行时多态。 纯虚函数也可以实现多态，只是纯虚函数不需要实现。

注意

> 对于C++语言，带变量的宏和函数重载（function overload）机制也允许将不同的特殊行为和单个泛化记号相关联。然而，习惯上并不将这种函数多态（function polymorphism）、宏多态（macro polymorphism）展现出来的行为称为多态（或静态多态），否则就连C语言也具有宏多态了。谈及多态时，默认就是指动态多态，而静态多态则是指基于模板的多态。

详细资料请看 多态性之编译期多态和运行期多态(C++版) 维基百科 多型

虚函数 纯虚函数 抽象类

前提

> 虚函数可以给出目标函数的定义，但该目标的具体指向在运行时才能确定。但虚函数并不代表该函数不被实现，而是为了允许用基类的指针或引用调用派生类的该函数。纯虚函数才代表函数不被实现，定义纯虚函数是为了实现一个接口，起到规范的作用，即继承基类的派生类中必须实现该纯虚函数

虚函数

通过实例讲解虚函数

```
class Base
{
public:
    virtual void hello() { cout << "I am Base::hello" << endl; }
    virtual void bye() { cout << "I am Base::bye" << endl; }
};

class Derived : public Base
{
public:
    Derived() { hello(); bye(); }
    void hello() { cout << "I am Derived::hello" << endl; }
};

main()
{
    Base *pb = new Derived();
    delete pb;  // 调用基类析构函数，仅释放基类所占内存

    Derived d = new Derived();  // 输出 I am Derived::hello
                                            I am Base::bye
}
```

1. 派生类中虚函数可不加 `virtual` 关键字
2. C++ 中的多态依靠 指针 和 引用 来实现
3. **访问级别**： 若虚函数在基类中为 `private`，即使在派生类中重写虚函数为 `public`，则也不能通过父类指针调用
4. **虚析构函数**： 若基类中析构函数不是虚函数，则上述代码 `delete pb` 会导致内存泄露，因为析构函数只会回收基类所占用的内存，执行基类的析构函数，无法执行派生类的析构函数。
5. **构造函数中调用虚函数**： 构造函数中调用虚函数类似于调用普通函数，不会发生动态多态，被调用的而函数来自于本类中重写的或者继承于基类的。

纯虚函数

> 纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法，定义纯虚函数的方法是在函数原型后加 `=0`，即 `virtual void function() = 0`

引入原因

1. 为了方便使用多态特性，常常需要在基类中定义虚函数
2. 在很多情况下，基类本身生成对象是不合理的，比如动物作为基类，可以派生出老虎、孔雀等子类，而动物本身生成对象不合理

为了解决上述问题, 引入了纯虚函数，将函数定义为纯虚函数(`virtual ReturnType function() = 0`), 则编译器要求在派生类中必须予以重写以实现多态性。同时拥有纯虚函数的类称为抽象类，不能生成对象，就很好的解决了上述两个问题。

作用

1. 声明了纯虚函数的类为一个抽象类，抽象类无法创建实例
2. 派生类必须重写该纯虚函数，否则该派生类也是抽象类
3. 纯虚函数的目的在于使派生类仅仅只是继承函数接口

抽象类

抽象类是一种特殊的类，是为了抽象和设计的目的建立的，处于继承层次的角上层

1. 定义： 拥有纯虚函数的类为抽象类，抽象类无法定义对象
2. 作用： 将有关操作作为接口组织在一个继承层次中，作为根，派生类将具体实现在抽象类中作为接口的操作
3. 注意： 抽象类只能做基类，其纯虚函数的实现由派生类给出，若派生类中没有重写纯虚函数，只是继承的话，则该派生类仍然是一个抽象类。

总结

1. 虚函数是 C++ 中用于实现多态的机制，核心思想就是通过基类指针或引用访问派生类定义的函数
2. 虚函数声明如下 `virtual ReturnType FunctionName(args); 虚函数在基类中必须实现
3. 纯虚函数声明如下 `virtual RetruanType FunctionName()=0`， 纯虚函数在基类中必须没有定义，只是作为接口用来规范派生类行为
4. 拥有纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向该抽象类的指针或引用。
5. 对于虚函数，基类和派生类都有各自的版本，由多态方式调用的时候动态绑定
6. 实现了纯虚函数的派生类，该纯虚函数在派生类中就变成了虚函数，派生类的派生类可以覆盖该虚函数，由多态方式调用的时候动态绑定
7. 在有动态分配堆上内存的时候，析构函数必须为虚函数，但没有必要为纯虚函数。虚析构函数会根据指针实际指向对象来调用析构函数。
8. 构造函数中调用虚函数类似于调用普通函数
9. 友元不是成员函数，只有成员函数才可以为虚函数，但可以通过让友元函数调用虚函数来解决友元函数的虚拟问题

详细内容请看 C++手稿：虚函数与多态 虚函数和纯虚函数的区别

虚函数表

C++ 的虚函数(`Virtual Function`)是通过一张虚函数表(`Virtual Table`)来实现的，简称`V-Table`。该表是存放着类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其真实反映的函数。这样，在有虚函数的类实例中，这个表被分配在这个实例的内存中，所以当我们使用父类的指针来操作一个子类的时候，该虚函数表就指明了实际所应该调用的函数

虚函数表主要分为基类

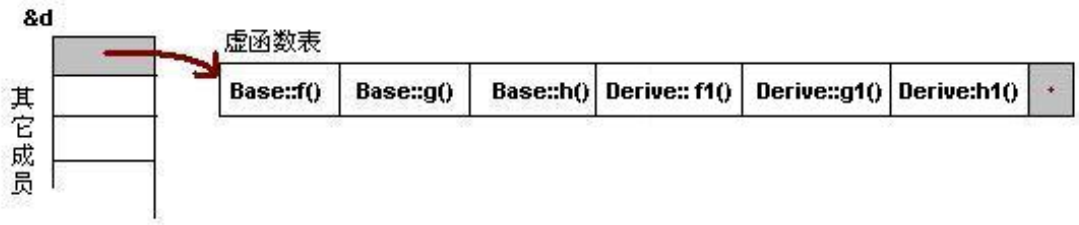
1. **一般继承**： 又分为一般继承无虚函数覆盖，一般继承有虚函数覆盖
2. **多重继承**： 又分为多重继承无虚函数覆盖，多重继承有虚函数覆盖

注意

1. C++ 的编译器要求 虚函数表的指针存在于对象实例中最前面的位置，这是为了保证在多层继承或多重继承的情况下，取虚函数表有最高的性能。
2. 同一个类的不同实例公用同一个虚函数表， 即在内存中，一个类的虚表只存在一份
3.

一般继承（无虚函数覆盖）

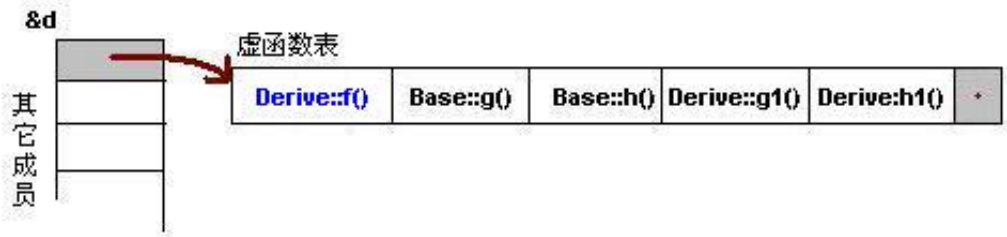
假如子类未重写父类中的虚函数，子类实例中，虚函数表如下



1. 虚函数按照其声明顺序放于表中
2. 父类的虚函数在子类的虚函数前面

一般继承（有虚函数覆盖）

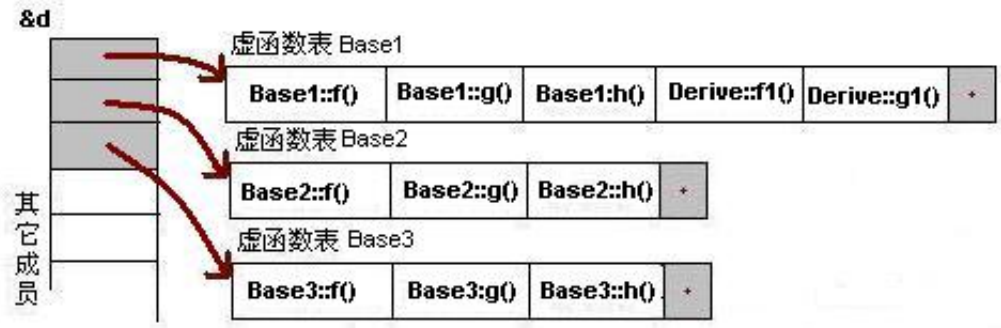
假如子类重写了父类中的虚函数，则子类实例中，虚函数表如下



1. 覆盖的 `f()` 函数被放到了虚表中原来父类虚函数的位置
2. 没有被覆盖的函数依旧

多重继承（无虚函数覆盖）

若子类继承了多个父类，且子类并未覆盖父类中的函数，则虚函数表如下



1. 每个父类都有自己的虚表
2. 子类的成员函数被放到了第一个父类的表中（所谓的第一个父类是按照声明顺序来判断的）

这样做的目的是为了解决不同的父类型的指针指向同一个子类实例，而能够调用实际的函数， 比如下例

```
Derived d;
Base1 *pb1 = &d;
pb1->f();  // 输出 Base1::f

Base2 *pb2 = &d;
pb2->f();  // 输出 Base2::f

Base3 *pb3 = &d;
pb3->f();  // 输出 Base3::f
```

多重继承（有虚函数覆盖）

加入子类继承了多个父类，且覆盖了父类中的`f()`函数， 则子类的虚表如下

1. 三个父类虚函数表中的f()的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的f()了。

详细内容请看 **强烈推荐** C++ 虚函数表解析 C++中的虚函数(表)实现机制以及用C语言对其进行的模拟实现

操作符重载(`operator` 关键字)

`operator` 是 `C++` 关键字，和运算符一起使用时，表示一个运算符函数，应将 `operator=` 视为整体

为什么使用操作符重载

对于系统内置的数据类型和标准库提供的类，C++提供了运算符重载。而对于用户自定义的类型， 比如 `Person` 类，如果想支持基本操作，比如大小、相等等，就需要用户自己定义操作符重载。

如何声明一个重载的操作符

操作符重载实现为类成员函数

重载的操作符在类体中被声明，声明方式如同普通成员函数一样，只不过函数名包括关键字 `operator` 以及紧跟其后的一个 `C++` 预定义操作符，比如下例

```
class Person
{
private:
    int age;
public:
    Person(int a): age(a) {}
    inline bool operator==(const Person& rhs) const;
};

inline bool Person::operator==(const Person &rhs) const
{
    return this->age == rhs.age;
}
int main()
{
    Person s1(0), s2(50);
    cout << (s1 == s2) << endl;
}
```

操作符重载实现为非类成员函数（即全局函数）

对于全局重载操作符，代表左操作数的参数必须被显式指定。例如：

```
class Person
{
public:
    int age;
    Person(int a): age(a) {}
};

bool operator==(const Person& lhs, const Person &rhs)
{
    return lhs.age == rhs.age;
}

int main()
{
    Person s1(0), s2(50);
    cout << (s1 == s2) << endl;
}
```

如何确定一个操作符重载为类成员函数还是全局函数呢?

1. 如果重载操作符为类成员函数，则其左操作数为调用该操作符的对象。若操作符的左操作数必须为其他类型时，则操作符重载为全局命名空间的成员
2. C++ 要求 `赋值=`、`下标[]`、`调用()`和`成员指向->`操作符必须定义为类成员操作符，否则编译错误
3. 具有对称性的操作符可以定义为全局命名空间成员

运算符重载限制

1. 只有C++预定义的操作符才可以被重载
2. 对于内置类型的操作符，预定义不能改变，即不能改变操作符原来的功能
3. 重载操作符不能改变操作符优先级
4. 重载操作符不能改变操作数的个数
5. 除了重载的函数调用运算符`operator()`外，其他重载运算符不能含有默认实参

详细内容请看C++ operator关键字（重载操作符） C++ 学习笔记之(14) - 重载运算与类型转换

C++ 内存对齐

基本概念

1. 数据类型自身的对齐值：对于char型数据，其自身对齐值为1，对于short型为2，对于int,float,double类型，其自身对齐值为4，单位字节。
2. 结构体或者类的自身对齐值：其成员中自身对齐值最大的那个值。
3. 指定对齐值：#pragma pack(n)，n=1,2,4,8,16改变系统的对齐系数
4. 数据成员、结构体和类的有效对齐值：自身对齐值和指定对齐值中小的那个值。

为什么需要内存对齐

1. 平台原因：不是所有的平台都支持访问任意地址上的任意数据，某些硬件平台若访问未对齐地址，会报错
2. 性能原因：数据结构（尤其是栈）应尽可能对其，因为访问未对齐的内存，处理器需要进行两次内存访问，而对齐的内存访问仅需要一次

对齐规则

1. 如果设置了内存对齐为 `i` 字节，类中最大成员对齐字节数为`j`，那么整体对齐字节`n = min(i, j)`（某个成员的对齐字节数定义：如果该成员是c++自带类型如int、char、double等，那么其对齐字节数=该类型在内存中所占的字节数；如果该成员是自定义类型如某个class或者struct，那个它的对齐字节数 = 该类型内最大的成员对齐字节数《详见实例4》）
2. 每个成员对齐规则：类中第一个数据成员放在offset为0的位置；对于其他的数据成员（假设该数据成员对齐字节数为`k`， `n`为整体对其字节），他们放置的起始位置offset应该是`min(k, n)`的整数倍
3. 整体对齐规则：最后整个类的大小应该是n的整数倍
4. 当设置的对齐字节数大于类中最大成员对齐字节数时，这个设置实际上不产生任何效果（实例2）；当设置对齐字节数为1时，类的大小就是简单的把所有成员大小相加

示例

```
class node1
{
    /*
     * 没有指定对齐字节, 则 n = max(char, int, short) = 4
     * 成员共占用[0-9] 10个字节, 还需整体对齐, 大小应该是 4 的倍数, 即 12
     */
    char c;  // 放在位置 0, 位置区间[0]
    int i;   // 4 = n, 那么放置起始位置应该是 4 的倍数, 即 4, 位置区间为[4-7]
    short s;  // 2 < n, 那么放置起始位置应该是 2 的倍数, 即 8, 位置区间为[8-9]
};

#pragma pack(8)
class node2
{
    /*
     * 指定对齐字节 8, 则 n = min(8, max(char, int, short)) = 4
```

```
     * 成员共占用 [0-7] 8个字节,刚好是 4 的倍数, 故大小是 8
     */
    int i;  // 放在位置 0, 位置区间为[0-3]
    char c;  // 1 < n, 那么放置起始位置应该是 1 的倍数, 即4, 区间为[4]
    short s;  // 2 < n, 那么放置起始位置应该是 2 的倍数, 即 6, 区间为[6-7]
};

#pragma pack(2)
class node3
{
    /*
     * 指定对齐字节 2, 则 n = min(2, max(char, int, short)) = 2
     * 成员共占用 [0-7] 8个字节,刚好是 4 的倍数, 故大小是 8
     */
    char c;  // 放在位置 0, 位置区间为 [0]
    int i;   // 4 > n, 那么放置起始位置应该是 2 的倍数, 即 2, 区间为 [2-5]
    short s;  // 2 = n, 那么放置起始位置应该是 2 的倍数, 即 6, 区间为 [6-7]
};

class node4
{
    /*
     * 未指定对齐字节, 故 node3 的对齐字节为 4, 整体大小为 12
     * n = min(4, max(char, int, short)) = 4
     * 成员共占用 [0-17] 18个字节,还要整体对齐, 大小应该是是 4 的倍数, 故大小是 20
     */
    char c;  // 放在位置 0, 位置区间为 [0]
    node3 n3;   // node3的对齐字节数 = 4 = n, 那么放置起始位置应该是 4 的倍数, 即 4, 区间为 [4-15]
    short s;  // 2 < n, 那么放置起始位置应该是 2 的倍数, 即 16, 区间为 [16-17]
};

class node5: public node4
{
    /*
     * 未指定对齐字节, 故 node4 的对齐字节为 4, 整体大小为 20
     * gcc 中是先继承后对齐, 故 node5 的整体大小为 24, 刚好为 4 的倍数, 无需调整
     */
    int i;  // 4 = n, 那么放置起始位置应该是 4 的倍数, 即 20, 区间为 [20-23]
};
```

详细内容请看 C++ 内存对齐

C++ 模板

模板机制为C++提供了泛型编程的方式，在减少代码冗余的时候仍可以提供类型安全。模板分为函数模板和类模板、模板的声明和定义只能在全局、命名空间和类范围内进行。

模板参数类型

模板参数类型

1. 类型形参，下例中的 `T1` 为类型形参
2. 非类型形参，`Maxsize` 为非类型形参， **非类型参数的模板实参必须是常量表达式**
3. 默认类型形参，`T2=int` ， 为默认类型形参，默认为 `int` ，注意，若后面还有形参，则都需要默认值。 **且默认实参只需提供一次**，否则会报错default argument for template parameter for class enclosing

```
template <typename T1, int Maxsize, typename T2=int>
class A {
public:
    T2 equal1(T1 a, T2 b);
};

template <typename T1, int Maxsize, typename T2>
T2 A<T1, Maxsize, T2>::equal1(T1 a, T2 b) { return a == b ? Maxsize
: 0;}

template <typename T1, int Maxsize, typename T2=int>
T2 equal1(T1 a, T2 b)
{
    return a == b ? Maxsize : 0;
};

int main()
{
    A<int, 5> a;
    cout << a.equal1(1, 1) << endl;
    cout << equal1<int, 5, int>(1, 1) << endl;
}
```

注意 示例无实际意义，仅是用作例子加深理解

泛化

```
template <typename 形参名, typename 形参名>
返回类型 函数名(参数列表) { 函数体 }
```

其中`template`和`typename`是关键字，`typename`可以被`class`代替。在此情景下`typename`和`class`无区别。<>括号中的参数叫做模板形参，模板形参不能为空，由模板实参来初始化。注意， 模板形参类型只能通过**实参推演**来进行，不能指定模板形参类型。以下为示例

```
// 类模板泛化
template <typename T1, typename T2>
struct B {
    B() { cout << "泛化 B<T1, T2>" << endl; }
};
// 函数模板泛化
template <typename T1, typename T2>
T2 equal2(T1 a, T2 b)
{
    cout << "泛化 equal2<T1, T2>" << endl;
};
```

全特化

全特化一个模板，可以对一个特定参数集合自定义当前模板，类模板和函数模板可以全特化。全特化的模板参数列表为空。并应该给出模板实参列表

```
// 类模板全特化
template <>
struct B<int, int> {
    B() { cout << "全特化 B<int, int>" << endl; }
};
// 函数模板全特化
template <>
int equal2(int a, int b)
{
    cout << "全特化 equal2<T1, T2>" << endl;
};
```

偏特化

类似于全特化，偏特化也是为了给自定义一个参数集合的模板，但偏特化后的模板需要进一步的实例化才能形成确定的签名。 值得注意的是函数模板不允许偏特化，这一点在Effective C++: Item 25中有更详细的讨论。 偏特化也是以template来声明的，需要给出剩余的"模板形参"和必要的"模板实参"。例如：

```
// 类模板偏特化
template <typename T2>
struct B<int, T2> {
    B() { cout << "偏特化 B<int, T2>" << endl; }
};
```

详细内容请看 C++模板的偏特化与全特化

智能指针

为什么使用智能指针

C++中会频繁的进行`new/delete`操作来申请和释放内存，而编程人员手动管理内存容易发生内存泄漏（忘记释放），或二次释放（导致异常）等，而使用智能指针可以更好的管理内存

智能指针分类

auto_ptr

`auto_ptr`为C98出现的智能指针，但因为某些问题被废弃

1. auto_ptr 赋值后会转移所有权
2. 不能指向数组，因为使用 delete 销毁对象
3. 不能作为容器成员，赋值后，会变为null

shared_ptr

多个智能指针可以共享同一个对象，`shared_ptr`智能指针使用引用计数，当最后一个`shared_ptr`销毁时，它有责任销毁指向的对象，并清理与该指向对象的所有资源

- **拷贝和赋值**：拷贝使得对象的引用计数加`1`， 而赋值使得左值对象引用计数减`1`，当计数为`0`时，自动释放内存，右值对象引用加`1`.
- **问题**：会出现循环引用。

unique_ptr

`unique_ptr`唯一拥有其所指对象，同一时刻只能有一个`unique_ptr`指向给定对象（通过禁止拷贝语义，使用移动语义实现）。

- 生命周期：从创建开始，直到离开作用域。离开作用域时，若其指向对象，则将其素质对象销毁。
- `unique_ptr`与所指对象关系: 在智能指针生命周期内，可更改智能指针所指对象，如通过`release`方法释放所有权，通过移动语义转移所有权

weak_ptr

`weak_ptr`允许共享但不拥有某对象，一旦最后一个拥有该对象的智能指针失去所有权，任何`weak_ptr`都会自动变成空。且`weak_ptr`不具有普通指针行为，没有重载`operator*`和`->`， `weak_ptr`可通过`shared_ptr`和`weak_ptr`构造。

使用`weak_ptr`主要是为了打破循环引用

简单智能指针的设计和实现

智能指针使用引用计数来跟踪指向同一指针的对象。

1. 创建对象时，初始化指针并将引用计数置为`1`
2. 通过拷贝构造函数创建对象时，增加与之对应的引用计数
3. 拷贝赋值运算符操作，减少左操作数所指对象的引用计数，若减少为`0`，则删除对象。增加右操作数的引用计数
4. 析构函数，减少引用计数，若减少为`0`，则删除对象

```
template <typename T>
class smart_pointer
{
public:
    smart_pointer(T* ptr=nullptr): _ptr(ptr)
    {
        if(_ptr)
            _count = new size_t(1);
        else
            _count = new size_t(0);
    }

    smart_pointer(const smart_pointer& sp)
    {
        if(this != &sp)
        {
            this->_ptr = sp._ptr;
            this->_count = sp._count;
            ++(*_count);
        }
    }

    smart_pointer&operator=(const smart_pointer& sp)
    {
        if(this->_ptr == sp._ptr)
            return *this;
        if(this->_ptr)
        {
            (*this->_count)--;
            if(*this->_count == 0)
            {
                delete(_ptr);
                delete(_count);
            }
        }

        this->_ptr = sp._ptr;
        this->_count = sp._count;
        (*this->_count)++;
```

```
        return *this;
    }

    T&operator*()
    {
        assert(this->_ptr == nullptr);
        return *(this->_ptr);
    }

    T*operator->()
    {
        assert(this->_ptr == nullptr);
        return this->_ptr;
    }
    ~smart_pointer()
    {
        (*this->_count)--;
        if(*this->_count == 0)
        {
            delete(_ptr);
            delete(_count);
        }
    }

    size_t use_count()
    {
        return *this->_count;
    }

    T* get()
    {
        return _ptr;
    }
private:
    T* _ptr;
    size_t* _count;
};
```

详细请看 C11中智能指针的原理、使用、实现 C/C面试知识总结

C++构造函数

C++ 成员初始化顺序

C++成员变量初始化列表和变量初始化顺序

C++成员变量、构造函数的初始化顺序

C++ 多态性

C++ 三大特性之多态

浅谈C++多态性

C++ 多态深度剖析

C++学习:虚函数,纯虚函数(virtual),虚继承,虚析构函数

C++ 虚函数&纯虚函数&抽象类&接口&虚基类

C++ 构造函数和析构函数

C++ 拷贝构造函数 参数引用传递还是值传递

参考，强烈推荐

1. C++常见面试问题总结
2. C++后台开发校招面试常见问题
3. C++ 后台开发面试时一般考察什么?