# Main

## 在main执行之前和之后执行的代码可能是什么？

main函数执行之前，主要就是初始化系统相关资源：

* 设置栈指针
* 初始化静态static变量和global全局变量，即.data段的内容
* 将未初始化部分的全局变量赋初值：数值型short，int，long等为0，bool为FALSE，指针为NULL等等，即.bss段的内容
* 全局对象初始化，在main之前调用构造函数，这是可能会执行前的一些代码
* 将main函数的参数argc，argv等传递给main函数，然后才真正运行main函数

## main函数执行之后：

* 全局对象的析构函数会在main函数之后执行；
* 可以用 atexit 注册一个遗言函数，它会在main 之后执行;

# 堆和栈的区别

## 申请方式不同

栈由系统自动分配；堆是自己申请和释放的。

## 申请大小限制不同

栈顶和栈底是之前预设好的，栈是向栈底扩展，大小固定，可以通过ulimit -a查看，由ulimit -s修改；堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。

## 申请效率不同

栈由系统分配，速度快，不会有碎片；堆由程序员分配，速度慢，且会有碎片。

## 形象的比喻

栈就像我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

《C++中堆（heap）和栈(stack)的区别》：

https://blog.csdn.net/qq\_34175893/article/details/83502412

# 指针

## 区别以下指针类型？

### int\* p[10]指针数组

拆分：int\* (p[10])

int \*p[10]是一个长度为10的指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。

const char\* (str[]) = { "aa","bbb","cccc" } 同理。

### int (\*p)[10]数组指针

拆分：int ((\*p)[10])

int (\*p)[10]指向int[10]数组的指针，强调是指针，只有一个变量，是指针类型。指向的是一个int类型的数组，这个数组大小是10。

### int\* p(int)函数声明

拆分int\* p(int)。

int \*p(int)是函数声明，函数名是p，参数是int类型的，返回值是int \*类型的。

### int (\*p)(int)函数指针

拆分int ((\*p)(int))

int (\*p)(int)是函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。

## a和&a

有什么区别？

假设数组int a[10];  
 int (\*p)[10] = &a; //p是指向数组的指针

* a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。
* &a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。
* 若(int \*)p ，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。

## 数组名和指针

（这里为指向数组首元素的指针）区别？

* 二者均可通过增减偏移量来访问数组中的元素。
* 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
* 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

## 野指针和悬空指针

都是是指向无效内存区域(这里的无效指的是"不安全不可控")的指针，访问行为将会导致未定义行为。

### 野指针

指的是没有被初始化过的指针

int main(void) {   
  int \* p;  
  std::cout<<\*p<<std::endl; return 0;  
}

因此，为了防止出错，对于指针初始化时都是赋值为 nullptr，这样在使用时编译器就会直接报错，产生非法内存访问。

### 悬空指针

指针最初指向的内存已经被释放了的一种指针。

int main(void) {   
  int \* p = nullptr;  
  int\* p2 = new int;   
  p = p2;  
  delete p2;  
}

此时 p和p2就是悬空指针，指向的内存已经被释放。继续使用这两个指针，行为不可预料。需要设置为p=p2=nullptr。此时再使用，编译器会直接保错。

避免野指针比较简单，但悬空指针比较麻烦。c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生。

# 一些关键字

## 宏定义#define

宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对 “参数” 进行的是一对一的替换。

### 宏定义和函数有何区别？

* 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
* 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
* 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
* 宏定义不要在最后加分号

### 宏定义和typedef区别？

* 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
* 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
* 宏不检查类型；typedef会检查数据类型。
* 宏不是语句，不在在最后加分号；typedef是语句，要加分号标识结束。
* 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

### 变量声明和定义区别？

* 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
* 相同变量可以在多处声明（外部变量extern），但只能在一处定义。

## define宏定义和const的区别

### 编译阶段

define是在编译的预处理阶段起作用，而const是在编译、运行的时候起作用。

### 安全性

* define只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包含住全部的内容，要不然很容易出错
* const常量有数据类型，编译器可以对其进行类型安全检查

### 内存占用

* define只是将宏名称进行替换，在内存中会产生多分相同的备份。const在程序运行中只有一份备份，且可以执行常量折叠，能将复杂的的表达式计算出结果放入常量表
* 宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。
* 宏不检查类型；const会检查数据类型。
* 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

## static

### 作用

* 修饰普通变量

隐藏。修改变量的存储区域和生命周期，使变量存储在静态区。

在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。

* 修饰普通函数

隐藏。表明函数的作用范围，仅在定义该函数的文件内才能使用。

在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。

* 修饰成员变量

修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。

只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始化，必须在类定义体外部初始化，初始化时不需要标示为static；可以被非static成员函数任意访问。

* 修饰成员函数

修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。

不具有this指针，无法访问类对象的非static成员变量和非static成员函数；不能被声明为const、虚函数和volatile；可以被非static成员函数任意访问。

### 用法和作用

1.先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可）

当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。

2.static的第二个作用是保持变量内容的持久。（static变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。

3.static的第三个作用是默认初始化为0（static变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节默认值都是0x00，某些时候这一特点可以减少程序员的工作量。

4.static的第四个作用：C++中的类成员声明static

1)  函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；

2)  在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

3)  在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；

4)  在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

5)  在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员变量。

### 类内

6)  static类对象必须要在类外进行初始化，static修饰的变量先于对象存在，所以static修饰的变量要在类外初始化；

7)  由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问 static修饰的类成员；

8)  static成员函数不能被virtual修饰，static成员不属于任何对象或实例，所以加上virtual没有任何实际意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual function

### 静态变量什么时候初始化

1)  初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。

2)  静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，但在C和C++中静态局部变量的初始化节点又有点不太一样。

在C中，初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在C语言中无法使用变量对静态局部变量进行初始化，在程序运行结束，变量所处的全局内存会被全部回收。

3)  而在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以C++标准定为，全局或静态对象是有首次用到时才会进行构造，并通过atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对静态局部变量进行初始化的。

## extern

* 被 extern 限定的函数或变量是 extern 类型的
* 被 extern "C" 修饰的变量和函数是按照 C 语言方式编译和链接的
* extern "C" 的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

[Day1\_基础\Day1\extern C .h](Day1_基础/Day1/extern%20C%20.h)

总结出使用方法，在C语言的头文件中，对其外部函数只能指定为extern类型，C语言中不支持extern "C"声明，在.c文件中包含了extern "C"时会出现编译语法错误。所以使用extern "C"全部都放在于cpp程序相关文件或其头文件中。

### C++调用C函数：

//xx.h extern int add(...)  
//xx.c int add(){}  
//xx.cpp extern "C" {#include "xx.h"}

### C调用C++函数

//xx.h extern "C"{ int add();}  
//xx.cpp int add(){}  
//xx.c extern int add();

# 一些运算符

## assert()

断言，是宏，而非函数。assert 宏的原型定义在 <assert.h>（C）、（C++）中，其作用是如果它的条件返回错误，则终止程序执行。可以通过定义 NDEBUG 来关闭 assert，但是需要在源代码的开头，include <assert.h> 之前。

[Day1\_基础\Day1\assert断言.h](Day1_基础/Day1/assert断言.h)

## sizeof()

* sizeof 对数组，得到整个数组所占空间大小。
* sizeof 对指针，得到指针本身所占空间大小。

### strlen和sizeof区别？

* sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
* sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
* 因为sizeof值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。

  int main(int argc, char const \*argv[]){     
      const char\* str = "name";  
      sizeof(str); // 取的是指针str的长度，是8  
      strlen(str); // 取的是这个字符串的长度，不包含结尾的 \0。大小是4  
      return 0;  
  }

# C和C++

## 区别

* C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
* 标准C++中的字符串类取代了标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。
* C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
* C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
* 在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。也就是C++可以重载，C语言不允许。
* C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而C语言中，必须要在函数开头部分。而且C++允许重复定义变量，C语言也是做不到这一点的
* 在C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
* C++相对与C增加了一些关键字，如：bool、using、dynamic\_cast、namespace等等。

## C++里的struct和class

总的来说，struct 更适合看成是一个数据结构的实现体，class 更适合看成是一个对象的实现体。

### 区别

#### 最本质的一个区别就是默认的访问控制

* 默认的继承访问权限。struct 是 public 的，class 是 private 的。
* struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

#### 相同点

* 两者都拥有成员函数、公有和私有部分
* 任何可以使用class完成的工作，同样可以使用struct完成

#### 不同点

* 两者中如果不对成员不指定公私有，struct默认是公有的，class则默认是私有的
* class默认是private继承，而struct模式是public继承
* class可以作为模板类型，struct不行

### C++和C的struct区别

C语言中：struct是用户自定义数据类型（UDT）；C++中struct是抽象数据类型（ADT），支持成员函数的定义，（C++中的struct能继承，能实现多态）

C中struct是没有权限的设置的，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数

C++中，struct增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为public（为了与C兼容）

struct作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在C中必须在结构标记前加上struct，才能做结构类型名（除：typedef struct class{};）;C++中结构体标记（结构体名）可以直接作为结构体类型名使用，此外结构体struct在C++中被当作类的一种特例。

## 结构体存储空间？

* 非静态成员的数据类型大小之和。
* 编译器加入的额外成员变量（如指向虚函数表的指针）。
* 为了边缘对齐优化加入的padding。

### 结构体内存对齐

* 内存对齐是编译器为了便于CPU快速访问而采用的一项技术
* 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
* 未特殊说明时，按结构体中size最大的成员对齐（若有double成员，按8字节对齐。）

### 原因

#### 平台原因(移植原因)

不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

#### 性能原因

数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

### 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），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。

### #pragma pack(n)

* 设定结构体、联合以及类成员变量以 n 字节方式对齐。
* [Day1\_基础\Day1\pragma\_pack字节对齐.h](Day1_基础/Day1/pragma_pack字节对齐.h)

### C++中类的内存分布情况

C++中类的数据成员和成员函数内存分布情况

C++类是由结构体发展得来的，所以他们的成员变量（C语言的结构体只有成员变量）的内存分配机制是一样的。下面我们以类来说明问题，如果类的问题通了，结构体也也就没问题啦。类分为成员变量和成员函数，我们先来讨论成员变量。

一个类对象的地址就是类所包含的这一片内存空间的首地址，这个首地址也就对应具体某一个成员变量的地址。（在定义类对象的同时这些成员变量也就被定义了），举个例子：

#include <iostream>  
using namespace std;  
  
class Person  
{  
public:  
    Person()  
    {  
        this->age = 23;  
    }  
    void printAge()  
    {  
        cout << this->age <<endl;  
    }  
    ~Person(){}  
public:  
    int age;  
};  
  
int main()  
{  
    Person p;  
    cout << "对象地址："<< &p <<endl;  
    cout << "age地址："<< &(p.age) <<endl;  
    cout << "对象大小："<< sizeof(p) <<endl;  
    cout << "age大小："<< sizeof(p.age) <<endl;  
    return 0;  
}  
//输出结果  
//对象地址：0x7fffec0f15a8  
//age地址：0x7fffec0f15a8  
//对象大小：4  
//age大小：4

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象的内存。

这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。要是成员函数占用类的对象空间，那么将是多么可怕的事情：定义一次类对象就有成员函数占用一段空间。

我们再来补充一下静态成员函数的存放问题：静态成员函数与一般成员函数的唯一区别就是没有this指针，因此不能访问非静态数据成员，就像我前面提到的，所有函数都存放在代码区，静态函数也不例外。所以有人一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。

## struct和typedef struct

[Day1\_基础\Day1\typedefStruct.h](Day1_基础/Day1/typedefStruct.h)

### C语言

* typedef struct Student S;此时 S 等价于 struct Student，但两个标识符名称空间不相同。
* 另外还可以定义与 struct Student 不冲突的 void Student() {}

### C++

由于编译器定位符号的规则（搜索规则）改变，导致不同于C语言。

* 如果在类标识符空间定义了 struct Student {...};，使用 Student me; 时，编译器将搜索全局标识符表，Student 未找到，则在类标识符内搜索。即表现为可以使用 Student 也可以使用 struct Student。
* 若定义了与 Student 同名函数之后，则 Student 只代表函数，不代表结构体。

## C语言实现C++类

### C 实现 C++ 的面向对象特性（封装、继承、多态）

* 封装：使用函数指针把属性与方法封装到结构体中
* 继承：结构体嵌套
* 多态：父类与子类方法的函数指针不同

# union联合体

联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

* 默认访问控制符为 public
* 可以含有构造函数、析构函数
* 不能含有引用类型的成员
* 不能继承自其他类，不能作为基类
* 不能含有虚函数
* 匿名 union 在定义所在作用域可直接访问 union 成员
* 匿名 union 不能包含 protected 成员或 private 成员
* 全局匿名联合必须是静态（static）的
* [Day1\_基础\Day1\union联合体.h](Day1_基础/Day1/union联合体.h)

# 位域Bit mode

* Bit mode: 2; // mode 占 2 位
* 类可以将其（非静态）数据成员定义为位域（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。
* 位域在内存中的布局是与机器有关的
* 位域的类型必须是整型或枚举类型，带符号类型中的位域的行为将因具体实现而定
* 取地址运算符（&）不能作用于位域，任何指针都无法指向类的位域

# :: 范围解析运算符

* 全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间
* 类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的
* 命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的
* [Day1\_基础\Day1\范围解析运算符.cpp](Day1_基础/Day1/范围解析运算符.cpp)

# enum 枚举类型

## 定作用域的枚举类型

* enum class open\_modes { input, output, append };

## 不限定作用域的枚举类型

* enum color { red, yellow, green };
* enum { floatPrec = 6, doublePrec = 10 };

# 内存分配和管理

## malloc

申请指定字节数的内存。申请到的内存中的初始值不确定。

用于分配、释放内存

## calloc

为指定长度的对象，分配能容纳其指定个数的内存。申请到的内存的每一位（bit）都初始化为 0。

## realloc

更改以前分配的内存长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。

## alloca

在栈上申请内存。程序在出栈的时候，会自动释放内存。但是需要注意的是，alloca 不具可移植性, 而且在没有传统堆栈的机器上很难实现。alloca 不宜使用在必须广泛移植的程序中。C99 中支持变长数组 (VLA)，可以用来替代 alloca。

## malloc、free

* 申请内存，确认是否申请成功

char \*str = (char\*) malloc(100);

assert(str != nullptr);

* 释放内存后指针置空

free(p);

p = nullptr;

## new、delete

* new / new[]：完成两件事，先底层调用 malloc 分配了内存，然后调用构造函数（创建对象）。
* delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
* new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

### delete this 合法吗？

合法，但：

* 必须保证 this 对象是通过 new（不是 new[]、不是 placement new、不是栈上、不是全局、不是其他对象成员）分配的
* 必须保证调用 delete this 的成员函数是最后一个调用 this 的成员函数
* 必须保证成员函数的 delete this 后面没有调用 this 了
* 必须保证 delete this 后没有人使用了

## malloc、realloc、calloc的区别

### malloc函数

void\* malloc(unsigned int num\_size);

int \*p = malloc(20\*sizeof(int));申请20个int类型的空间；

### calloc函数

void\* calloc(size\_t n,size\_t size);  
  
 int \*p = calloc(20, sizeof(int));

省去了人为空间计算；malloc申请的空间的值是随机初始化的，calloc申请的空间的值是初始化为0的；

### realloc函数

void realloc(void \*p, size\_t new\_size);

给动态分配的空间分配额外的空间，用于扩充容量

## new / delete 与 malloc / free的异同

### 相同点

* 都可用于内存的动态申请和释放

### 不同点

* 前者是C++运算符，后者是C/C++语言标准库函数
* new自动计算要分配的空间大小，malloc需要手工计算
* new是类型安全的，malloc不是。例如：

int \*p = new float[2];  //编译错误  
int \*p = (int\*)malloc(2 \* sizeof(double)); //编译无错误

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换将void\*指针转换成我们需要的类型。

* new调用名为operator new的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为operator delete的标准库函数释放该对象所用内存。后者均没有相关调用
* 后者需要库文件支持，前者不用
* new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象。
* new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL。

## malloc与free的实现原理？

1、 在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk、mmap、，munmap这些系统调用实现的;

2、 brk是将数据段(.data)的最高地址指针\_edata往高地址推,mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系；

3、 malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推；malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配；brk分配的内存需要等到高地址内存释放以后才能释放，而mmap分配的内存可以单独释放。当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩。

4、 malloc是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

## new和delete是如何实现的？

### new的实现过程是

* 首先调用名为operator new的标准库函数，分配足够大的原始为类型化的内存，以保存指定类型的一个对象；
* 接下来运行该类型的一个构造函数，用指定初始化构造对象；最后返回指向新分配并构造后的的对象的指针

### delete的实现过程

* 对指针指向的对象运行适当的析构函数；
* 然后通过调用名为operator delete的标准库函数释放该对象所用内存

## malloc和new的区别？

* malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。
* malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
* malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

## delete和delete[]区别？

* delete只会调用一次析构函数。
* delete[]会调用数组中每个元素的析构函数。

## delete p、delete [] p、allocator

1、 动态数组管理new一个数组时，[]中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数；

2、 new动态数组返回的并不是数组类型，而是一个元素类型的指针；

3、 delete[]时，数组中的元素按逆序的顺序进行销毁；

4、 new在内存分配上面有一些局限性，new的机制是将内存分配和对象构造组合在一起，同样的，delete也是将对象析构和内存释放组合在一起的。

allocator将这两部分分开进行，allocator申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。

## malloc申请的存储空间能用delete释放吗

不能，malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free的。

malloc /free的操作对象都是必须明确大小的，而且不能用在动态类上。

new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。

当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常

## 定位 new

定位 new（placement new）允许我们向 new 传递额外的地址参数，从而在预先指定的内存区域创建对象。（place\_address 是个指针，initializers 提供一个（可能为空的）以逗号分隔的初始值列表）

* new (place\_address) type、
* new (place\_address) type (initializers)
* new (place\_address) type [size]
* new (place\_address) type [size] { braced initializer list }

## 什么是内存泄露，如何检测与避免

### 内存泄露

一般我们常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。应用程序般使用malloc,、realloc、 new等函数从堆中分配到块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了

### 避免内存泄露的几种方式

* 计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计数，如果不为0则表示存在内存泄露
* 一定要将基类的析构函数声明为虚函数
* 对象数组的释放一定要用delete []
* 有new就有delete，有malloc就有free，保证它们一定成对出现

### 检测工具

Linux下可以使用Valgrind工具

Windows下可以使用CRT库

# 运行时类型信息 (RTTI)

## dynamic\_cast

* 用于多态类型的转换

## typeid

* typeid 运算符允许在运行时确定对象的类型
* type\_id 返回一个 type\_info 对象的引用
* 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数
* 只能获取对象的实际类型

## type\_info

* type\_info 类描述编译器在程序中生成的类型信息。此类的对象可以有效存储指向类型的名称的指针。type\_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。类型的编码规则和排列顺序是未指定的，并且可能因程序而异。
* 头文件：typeinfo
* [Day1\_基础\Day1\运行时类型信息.h](Day1_基础/Day1/运行时类型信息.h)

# 函数调用

## 压栈过程

#include <iostream>  
using namespace std;  
  
int f(int n)   
{  
    cout << n << endl;  
    return n;  
}  
  
void func(int param1, int param2)  
{  
    int var1 = param1;  
    int var2 = param2;  
    printf("var1=%d,var2=%d", f(var1), f(var2));  
}  
  
int main(int argc, char\* argv[])  
{  
    func(1, 2);  
    return 0;  
}  
//输出结果  
//2  
//1  
*//var1=1,var2=2*

当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地址、main的参数、mian函数中的变量、进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈；

当func()调用f()的时候，编译器此时会将func()函数的运行状态进行压栈，再将的返回地址、f()函数的参数从右到左、f()定义变量依次压栈

从代码的输出结果可以看出，函数f(var1)、f(var2)依次入栈，而后先执行f(var2)，再执行f(var1)，最后打印整个字符串，将栈中的变量依次弹出，最后主函数返回。

## 临时变量返回值

首先需要明白一件事情，临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了。

C语言里规定：16bit程序中，返回值保存在ax寄存器中，32bit程序中，返回值保持在eax寄存器中，如果是64bit返回值，edx寄存器保存高32bit，eax寄存器保存低32bit。

由此可见，函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命周期没有关系。

如果我们需要返回值，一般使用赋值语句就可以了。