C语言

1.static

- 在C中，凡是在任何代码块之外声明的变量总是存储于静态内存中，也就是不属于堆栈的内存，这类变量称为静态（static）变量。
- 在C中，静态变量，即全局变量和static变量，是在程序运行前创建的。

2.extern

- extern：**声明**变量或函数是在其它文件或本文件的其他位置定义;
- 存在全局静态存储区；是外部变量（全局变量）；
- extern说明符扩展了全局变量的作用域;
- 声明可以多次，定义只有一次。
- **int a;**在函数体内，是定义。其他情况下，是声明。

> 关于include

- C语言编译的第一步，就是替换include和define，这里的替换方法就是复制粘贴头文件的内容。所以，如果头文件定义了变量，那多次引用会造成重复定义。
- 解决方法：使用条件编译#ifndef。

C++语言

1.static

> static全局变量和全局变量的区别

- 普通的全局变量，它作用在整个源程序，如果一个源程序有多个源文件，那这个全局变量在所有源文件中都有效。
- static全局变量限制了它的作用域只在定义该变量的源文件内， 只能此文件的函数公用，其他的源文件不能使用该static全局变量。其他文件还能命名相同名字的变量，不会发生命名冲突。

> static局部变量和局部变量的区别

- static局部变量驻留在全局数据区，直到程序结束。但它的作用域还是局部作用域。
- static局部变量在首次声明时被初始化，内存在全局数据区，之后再调用也不会初始化或重新分配内存。
    - 只声明不初始化会进行默认初始化。

> static成员

- 类里static声明表示，函数或变量属于类，而不是对象。
- static成员间可以互相访问，普通成员函数可以访问static成员，static成员函数无法访问普通成员

2.restrict

- 通过加上restrict关键字，编程者可提示编译器：在该指针的生命周期内，其指向的对象不会被别的指针所引用。

3.显式类型转换

- static_cast/dynamic_cast/const_cast/reinterpret_cast
- static_cast：任何具有明确定义的类型转换，只要不包含底层const，都可以用static_cast。

```
int j;
double slope = static_cast<double>(j);

double d;
void *p = &d;
double * dp = static_cast<double*>(p);
```

- const_cast：只能改变对象的底层const

```
const char *pc;
char *p = const_cast<char*>(pc);
// 对常量对象执行const_cast 造成的行为是未定义的

// 顶层const 表示指针本身是个常量
// 底层const 表示指针指向的对象是个常量
int i = 0;
int *const p1 = &i;      // 顶层const,p1不可变
const int ci = 42;       // 顶层const ci不可变
const int * p2 = &ci;    // 底层const,p2可变
// 拷贝操作时，顶层const不受影响，拷贝指针和普通数据类型不会影响源变量的const属性。 底层const必须有相同的底层const属性，否则会改变源变量的const属性
int &r = ci;             // 普通int &r 不能绑定到int常量
const int & r2 = i;      // 底层const 可以绑定到普通int
```

- reinterpret_cast：运算对象位模式提供低层次上的重新解释

- 感觉像C的union。

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
// 危险行为！！！
string str(pc);
```

- dynamic_cast：将基类的指针或引用安全地转换为派生类的指针或引用。

```
dynamic_cast<type*>(e);
dynamic_cast<type&>(e);          // e必须是左值
dynamic_cast<type&&>(e);         // e不能是左值
// e必须是type的公有派生类、公有基类或e就是type类型
// 指针转换失败会返回0
Base *bp;
if(Derived *dp = dynamic_cast<Derived*>(bp)){

}
// 引用转换失败会报bad_cast错
const Base &b;
try{
    const Derived &d = dynamic_cast<const Derived&>(b);
}
catch(bad_cast){}
```

4.左值和右值

- 当对象被用作左值时，用的是对象的身份（在内存中的位置）。
- 当对象被用作右值时，用的是对象的值（内容）。
- 需要右值的地方，可以用左值来替代。但不能把右值当成左值使用。

5.类

5.1 构造函数

- 只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数。

5.2 拷贝构造函数

- 当类没有定义任何拷贝构造函数时，编译器会自动生成默认拷贝构造函数。
  - 当类声明了其他类型的构造函数，却没有声明拷贝构造函数，即同上。
- 拷贝构造函数的第一个参数必须是一个引用类型，此参数通常是const的。拷贝构造函数通常是非explicit的。

5.3 拷贝赋值运算符

- 当类没有定义任何拷贝赋值运算符时，编译器会合成一个。

```
class Foo{
public:
    // 拷贝构造函数
    Foo(const Foo&){}
    // 拷贝赋值运算符
    Foo & operator= (const Foo &){
        ...;
        // 返回左侧对象的引用 要求是成员函数
        return *this;
    }
}
```

5.4 析构函数

- 内置指针指向的对象不会在析构的隐式销毁阶段被delete，析构函数体要主动delete动态分配的内存空间。

```
class Foo{
public:
    ~Foo(){
        // 执行函数体 然后逆序销毁成员
    }
}
```

> 三/五法则：要么定义前三个拷贝控制操作，要么定义五个；
>
> 如果需要析构函数，那么几乎肯定需要拷贝和拷贝赋值；
>
> 如果需要拷贝构造函数或者拷贝赋值运算符，那么几乎肯定需要另一个；
>
> 如果只定义移动构造函数和移动赋值运算符，那么合成的拷贝构造函数和拷贝赋值运算符是删除的；

5.5 移动构造函数

- 只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非static数据成员都可以移动时，才会合成移动构造函数和移动赋值运算符。
    - 可以移动 == 内置类型 或 定义了移动操作的类

```
// 移动构造函数 本质是 接管源对象内存 所以不需要拷贝。
// 源对象的指针需要置空 避免源对象析构时 摧毁移动目标对象的内存空间
// noexcept表示函数不会抛出异常。这是为了让 进行 “谨慎”内存操作 的函数 可以选择调用移动构造，而不用担心异常造成的内存操作中断 导致错误。
StrVec::StrVec(StrVec && s) noexcept : elements(s.elements),
first_free(s.first_free), cap(s.cap){
    s.elements = s.first_free = s.cap = nullptr;
}
```

5.6 移动赋值运算符

```
StrVec& StrVec::operatoer= (StrVec && rhs) noexcept{
    if(this != &rhs){
        free();          // 释放已有元素
        elements = rhs.elements;
        first_free = rhs.first_free;
        cap = rhs.cap;
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
```

6.C++11六种Memory order

> C++ Concurrency in Action

6.1 mem_order_seq_cst

- 顺序一致性：不允许重新排序线程的步骤。对于单个线程来说，程序顺序就是实际执行顺序。对于所有线程，所有语句的执行序列全局一致。

6.2 mem_order_acquire/release/acq_rel

- happens-before：多线程程序中，如果A happens-before B，则要求A操作造成的所有内存效果，应该对B操作的执行线程可见且在B之前发生。
    - happens-before是严格偏序的，可传递、非自反、非对称。
    - 要在线程间建立该关系，需要原子操作。
- 对于acquire/release：针对同一个对象的多个原子操作不会被重新排序。
- acquire操作A之后的 原子或非原子 load/store操作 不能被重排到操作A之前。
- release操作A之前的 原子或非原子 load/store操作 不能被重排到操作A之后。
- 对于单个线程来说，程序顺序就是实际执行顺序。对于所有线程，所有语句的执行序列**不一定**全局一致。

6.3 mem_order_relaxed

- 对于单个线程来说，不同变量间没有happens-before关系，程序顺序不一定是实际执行顺序。
- 同一变量在同一线程内，具有happens-before关系。

6.4 mem_order_consume

- memory_order_consume完全依赖于数据

    - 两种新关系用来处理数据依赖：前序依赖(dependency-ordered-before)和携带依赖(carries-a-dependency-to)。
- 就像前列(sequenced-before)，携带依赖对于数据依赖的操作，严格应用于一个独立线程和其基本模型；
- 如果A操作结果要使用操作B的操作数，而后A将携带依赖于B。当A携带依赖B，并且B携带依赖C，就额可以得出A携带依赖C的关系。

```
struct X{
    int i;
    std::string s;
};
std::atomic<X*> p;
std::atomic<int> a;
void create_x()
{
  X* x=new X;
  x->i=42;
  x->s="hello";
  a.store(99,std::memory_order_relaxed);  // 1
  p.store(x,std::memory_order_release);  // 2
}
void use_x()
{
  X* x;
  while(!(x=p.load(std::memory_order_consume)))  // 3
    std::this_thread::sleep(std::chrono::microseconds(1));
  assert(x->i==42);  // 4
  assert(x->s=="hello");  // 5
  // 4和5不会触发assert 6可能会
  assert(a.load(std::memory_order_relaxed)==99);  // 6
}
int main()
{
  std::thread t1(create_x);
  std::thread t2(use_x);
  t1.join();
  t2.join();
}
// p携带依赖于x，所以可以保证45
// a的加载是memory_order_relaxed的，且p的加载是memory_order_consume，无法确定顺序。
```

7.内存对齐

- 自然对齐：一个变量的内存地址正好位于它长度的整数倍。如：32位机，int变量的地址为0x00000004。

100.链接库

- 作者：qin meng
  链接：https://www.zhihu.com/question/22940048/answer/222625910
  来源：知乎

gcc使用-Wl传递连接器参数，ld使用-Bdynamic强制连接动态库，-Bstatic强制连接静态库。所以部分静态，部分动态连接这么写：

```
gcc ... -Wl,-Bstatic -l<your-static-lib> -Wl,-Bdynamic -l<your-dynamic-lib> ...
```

举个例子，你想静态连接libA.a同时动态连接libB.so，(先保证你的连接路径-L里面能找到对应的静态或者动态库)，这么写：

```
gcc ... -Wl,-Bstatic -lA -Wl,-Bdynamic -lB ...
```

这里需要注意，强制静态或者动态连接标记之后的链接库都将按照前面最近的一个标记进行链接，所以如果后面出现了一个libC，没有指定连接标记，那么libC将会被动态连接：

```
gcc ... -Wl,-Bstatic -lA -Wl,-Bdynamic -lB ... -lC
```

如果参数里面没指定强制的连接方式标记，那么gcc将按照默认的优先级去链接，**优先动态链接**，所以如果你这么写，且同时存在libC.so和libC.a那么libC将被动态链接：

```
gcc ... -lC
```

由于-B连接标记会改变默认连接方式，所以在Makefile里面如果有人这么干：

```
LIBS += -Wl,-Bstatic -lC
```

那么他后面的LIBS+=的库就都只能以静态方式连接了，有时候这是不行的，因为没有静态库，所以会有人这么应对：

```
LIBS += -Wl,-Bdynamic -lD
```

这样就改回来了。但是这种胡乱改的行为是非常不好的，比较好的行为应该这样：

```
LIBS += -l<auto-link-lib>
STATIC_LIBS += -l<static-lib>
DYN_LIBS += -l<dynamic-lib>
LDFLAGS := ${LIBS} -Wl,-Bstatic ${STATIC_LIBS} -Wl,-Bdynamic ${DYN_LIBS}
```

这样当你不关心怎么连接的时候用LIBS，当你想静态连接的时候用STATIC_LIBS，当你想动态连接的时候用DYN_LIBS。