C++ primer 个人总结

1.右值引用

1.左值是指表达式结束后依然存在的持久化对象，右值是指表达式结束时就不再存在的临时对象。

2.区分方法：看能不能对表达式取地址，如果能，则为左值，否则为右值。

3.顶层const（top-level const）表示指针（或引用等）本身是个常量。底层const（low-level const）表示指针指的对象是一个常量。

1.左值引用， 使用 T&, 只能绑定左值。

2.右值引用， 使用 T&&， 只能绑定右值。

3.常量左值， 使用 const T&, 既可以绑定左值又可以绑定右值。

4.已命名的右值引用，编译器会认为是个左值。

2.const限定符的使用

1.顶层const（top-level const）表示指针（或引用等）本身是个常量。

2.底层const（low-level const）表示指针指的对象是一个常量。

3.constexpr以及常量表达式

1.constexpr特指编译期常量，而const指的是编译期常量和运行时常量。

4.auto与decltype类型指示符

1.auto能让编译器帮我们分析表达式的类型。auto定义多个变量时候，他们的类型必须是一样的。

2.decltype选择并返回操作数的数据类型。

5.++i与i++的区别

1.++i返回是左值，i++返回的是右值（临时变量）。

2.++i比i++速率高。因为i++需要申请一个临时变量去存储未自加的i。

6.内联(inline)函数和constexpr

1.内联函数在编译期直接在调用点替换展开，无须调用函数的额外开销(保存栈，地址等信息)。

2.constexpr是定义常量表达式的函数。

3.定义constexpr函数时需要注意：函数的返回类型以及所有形参的类型都得是字面值类型，而且函数体中必须有且仅有一条return语句。

4.编译器把对constexpr函数的调用替换成其结果值。

5.constexpr函数被隐式地指定为内联函数。

7.强制类型转换

1.static_cast：任何具有明确定义的类型转换（除底层const之外）。

2.const_cast：只能改变运算对象的底层const。

3.reinterpret_cast：为运算对象的位模式提供较低层次的重新解释。

4.dynamic_cast：运行时类型识别。动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常

8.信号处理

1.信号是由操作系统传给进程的中断，会提早终止一个程序。

1.SIGABRT：程序的异常终止，如调用 abort。

2.SIGFPE：错误的算术运算，比如除以零或导致溢出的操作。

3.SIGILL：检测非法指令。

4.SIGINT：接收到交互注意信号。

5.SIGSEGV：非法访问内存。

6.SIGTERM：发送到程序的终止请求。

2.C++ 信号处理库提供了 signal 函数，用来捕获突发事件。

```
    void (*signal (int sig, void (*func)(int)))(int);       //sig:信号的编号,func指向信号处理函数的指针
```

3.可以使用函数 raise() 生成信号。

```
    int raise (signal sig); //sig是要发送的信号的编号，这些信号包括：SIGINT、SIGABRT、SIGFPE、SIGILL、SIGSEGV、SIGTERM、SIGHUP
```

9.多线程编程

1.创建线程

```
    #include <pthread.h>
    pthread_create (thread, attr, start_routine, arg)   //thread:指向线程标识符指针,attr:设置线程属性,默认为NULL，start_routine:线程运行函数起始地址,arg:运行函数的参数，必须通过把引用作为指针强制转换为 void 类型进行传递。如果没有传递参数，则使用 NULL。
    //创建成功函数返回0，失败返回非0数
```

2.终止线程

```
    pthread_exit (status)   //
```

3.连接线程

```
    pthread_join(threadid, status)    //pthread_join() 子程序阻碍调用程序，直到指定的 threadid 线程终止为止。
    //只有创建时定义为可连接的线程才可以被连接。
```

4.分离线程

```
    pthread_detach(threadid)
```

10.c++11多线程编程

1.通过创建std::thread类的对象来创建其他线程，每个std::thread对象都可以与一个线程相关联。

```
    #include <thread>
    std::thread thObj(<CALLBACK>)      //CALLBACK：回调，可以为函数指针、函数对象、Lambda函数。
```

11.c++锁

1.互斥锁：互斥锁用于控制多个线程对他们之间共享资源互斥访问的一个信号量。

1.直接操作mutex，即直接调用mutex的lock/unlock函数。若互斥体被第二个锁请求锁住，则第二个锁所在线程被阻塞直至第一个锁解锁。

```
    mutex.lock();
    //临界区代码
    mutex.unlock();
```

2.lock_guard：简单锁，构造时请求上锁，释放时解锁，性能耗费较低。适用区域的多线程互斥操作。

3.unique_lock：更多功能也更灵活的锁，随时可解锁或重新锁上（减少锁的粒度），性能耗费比前者高一点点。适用灵活的区域的多线程互斥操作。

4.为输出流使用单独的 mutex。因为IO流不是线程安全的。

2.条件锁：条件锁就是所谓的条件变量，某一个线程因为某个条件为满足时可以使用条件变量使改程序处于阻塞状态。一旦条件满足以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。

1.wait()的实现接下来检查条件，并在满足时返回。如果条件不满足，wait()解锁互斥元，并将该线程置于阻塞或等待状态。

2.notify_one()/notify_all()：激活某个或者所有等待的线程，被激活的线程重新获得锁。

3.自旋锁

1.互斥锁是是一种sleep-waiting的锁。未申请到互斥锁的线程会处于阻塞状态，处理器会去处理其他任务而不是一直等待。

2.自旋锁是一种busy-waiting的锁。未申请到自旋锁的线程会一直请求自旋锁直到获得这个自旋锁。

4.读写锁

1.解决读者-写者问题。

5.递归锁

12.RAII

1.RAII全称是“Resource Acquisition is Initialization”，即是在构造函数中申请分配资源，在析构函数中释放资源。

栈溢出的原因以及解决方法

1.栈溢出的原因

1.函数调用层次过深,每调用一次,函数的参数、局部变量等信息就压一次栈。

2.局部变量体积太大。

2.解决方法

1.增加栈内存的数目；如果是不超过栈大小但是分配值小的，就增大分配的大小。

2.使用堆内存。

2.C++标准库

1.iostream、fstream以及sstream头文件

1.iostream：cin,cout,cerr,clog。

2.刷新输出缓冲：

1.程序正常结束。

2.缓冲区满。

3.使用endl(插入一个换行符)，flush和ends(插入一个空格符)来显式刷新缓冲。

4.当一个输出流被关联到另一个流。

5.使用unitbuf设置流的内部状态来清空缓冲区。

6.如果程序异常终止，输出缓冲区不会被刷新。

3.文件模式:1.只读(in)，2.只写(out)，3.截断(trunc)，4.追加写(app)，5.打开文件立刻定位到文件末尾(ate)，6.二进制(binary)。

2.lambda表达式

```
[函数对象参数] (操作符重载函数参数) mutable 或 exception 声明 -> 返回值类型 {函数体}
```

1 [函数对象参数]:标识一个 Lambda 表达式的开始，这部分必须存在，不能省略。

2 (操作符重载函数参数):标识重载的()操作符的参数，没有参数时，这部分可以省略。参数可以通过按值（如:(a,b)）和按引用(如:(&a,&b))两种方式进行传递。

3 mutable或exception声明:这部分可以省略。按值传递函数对象参数时，加上mutable修饰符后，可以修改传递进来的拷贝（注意是能修改拷贝，而不是值本身）。exception声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用 throw(int)。

4 -> 返回值类型:标识函数返回值的类型。当返回值为void，或者函数体中只有一处return的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

5 {函数体}:标识函数的实现，这部分不能省略，但函数体可以为空。

3.动态内存的管理以及智能指针的应用

1.智能指针

1.防止内存泄漏，方便管理堆内存。

2.智能指针：

1.shared_ptr：使用引用计数，每个share_ptr的拷贝都是指向相同的内存。当计数为0时，系统自动删除其所指向的堆内存。防止循环引用。

2.unique_ptr：同一时刻只能有一个unique_ptr指向一个对象。

3.weak_ptr：为了配合shared_ptr而引入，不能引起引用次数的增加。

4.动态数组与allocator类

1.动态数组

1.分配一个动态数组即是在分配一个new对象时在类型名之后加一对方括号，用来存放数组大小。数组分配成功后返回一个指向第一个对象的指针。

2.allocator类

1.allocator允许用户先分配内存，再构造对象。分配内存用a.allocate(n),该函数返回一个指向首个内存的指针，构造对象用a.construct(q++, args)。使用或访问内存之前必须构造对象。

2.a.destroy(q)用来释放对象。释放内存用a.deallocate(p,n),n必须和所分配的内存数量大小相等，且释放内存之前必须释放对象。

3.类设计者的工具

1.explict: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，只能显式的进行类型转换。

2.default：如果已经定义了有参的构造函数，那么编译器就不再生成默认的构造函数。使用=default让编译器生成默认的构造函数。

1.拷贝构造函数、移动构造函数（右值引用）以及动态内存管理类

1.构造函数

1.作用：赋初值,初始化对象的数据成员,由编译器帮我们调用。

2.①函数名和类名一样。②没有返回值。③支持有参/无参。④可以重载。

3.调用时机：在类的对象创建时刻,编译器帮我们调用构造函数。

2.析构函数

1.作用：用于释放资源。

2. ①和类名一样,不过得在前面加上~。②无参数,无返回值。③因为无参数,无返回值,所以不可以重载。

3.调用时机：快退出函数的时候,编译器帮我们调用

3.拷贝构造函数

1.拷贝构造函数是一种特殊的构造函数，具有单个形参，该形参（常用const修饰）是对该类类型的引用。

```
class CExample{
    CExample(const CExample & c)   //拷贝构造函数
    {
        ........
    }
}
```

1.通过使用另一个同类型的对象来初始化新创建的对象。

2.对象以值传递的方式传入函数参数。

3.对象以值传递的方式从函数返回。

2.当定义一个新对象并用一个同类型的对象对它进行初始化时，将显式使用拷贝构造函数。

3.当该类型的对象传递给函数或从函数返回该类型的对象时，将隐式调用拷贝构造函数。

4.必须定义拷贝构造函数的情况：类有一个数据成员是指针，或者是有成员表示在构造函数中分配的其他资源。

4.移动构造函数

```
class ClassName{
    ClassName(ClassName&&  tmp)  //移动构造函数
    {
        .......
    }
}
```

1.移动构造不另外开辟新空间，直接偷走临时对象的内存空间，占为己有，节省了开辟内存与赋值的时间。

2.调用时机：用到临时对象（右值）的时候就会执行移动语义。除了编译器创建的临时对象作为右值外，使用std::move也能得到一个左值的右值引用。

5.动态内存管理类

2.类中重载,覆盖，重定义及类型转换（注意避免重载的二义性）

1.函数重载

1.重载要求参数列表必须不同，函数的返回值不同不能作为作为重载的依据/

2.运算符重载

1.重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的

```
Box operator+(const Box&);
```

2.运算重载符不可以改变语法结构。

3.运算重载符不可以改变操作数的个数。

4.运算重载符不可以改变优先级。

5.运算重载符不可以改变结合性。

3.覆盖是存在类中，子类重写从基类继承过来的函数。被重写的函数必须是virtual的，不能是static函数。

4.重定义也叫做隐藏，子类重新定义父类中有相同名称的非virtual函数。

3.面向对象程序设计（数据抽象，数据封装，继承和动态绑定）

1.数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。

2.数据封装是面向对象编程中的把数据和操作数据的函数绑定在一起的一个概念，这样能避免受到外界的干扰和误用，从而确保了安全。

3.继承

1.单继承（父类含虚函数）

1.子类与父类拥有各自的一个虚函数表。

2.若子类并无overwrite父类虚函数，用父类虚函数。

3.若子类重写（overwrite）了父类的虚函数，则子类虚函数将覆盖虚表中对应的父类虚函数。

4.若子声明了自己新的虚函数，则该虚函数地址将扩充到虚函数表最后。

2.一般多继承（不含菱形继承）

1.若子类新增虚函数，放在声明的第一个父类的虚函数表中。

2.若子类重写了父类的虚函数，所有父类的虚函数表都要改变。

3.内存布局中，父类按照其声明顺序排列。

3.虚拟继承。虚继承解决了菱形继承中派生类拥有多个间接父类实例的情况。

- 类通过虚继承共享虚基类的状态。 1.虚继承的子类，如果本身定义了新的虚函数，则编译器为其生成一个新的虚函数指针（vptr）以及一张虚函数表。该vptr位于对象内存最前面（对比非虚继承：直接扩展父类虚函数表）。

2.虚继承的子类也单独保留了父类的vprt与虚函数表。

3.虚继承的子类有虚基类表指针（vbptr）。

4.动态绑定

1.静态类型：对象在声明时采用的类型，在编译期既已确定类型。

2.动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的类型。

3.静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期。

4.动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期。虚函数都是动态绑定。

5.静态绑定和动态绑定的区别：

1. 静态绑定发生在编译期，动态绑定发生在运行期。

2. 对象的动态类型可以更改，但是静态类型无法更改。

3. 要想实现动态，必须使用动态绑定。

4. 在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定。

4.虚函数

1.虚函数的主要功能是实现了多态机制。多态：用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。

2.关键字virtual修饰虚函数。

3.虚函数的作用：

1.定义子类对象，并调用对象中未被子类覆盖的基类函数。

2.在使用指向子类对象的基类指针，并调用子类中的覆盖函数时，如果该函数不是虚函数，那么将调用基类中的该函数；如果该函数是虚函数，则会调用子类中的该函数。

4.每个包含了虚函数的类都维护一个虚函数表，放置虚函数地址。虚函数表是属于类的。虚函数的地址按照声明顺序依次存放在虚表中。

5.虚函数(virtual)不能声明为static函数。

1.静态成员函数，可以不通过对象来调用，没有隐藏的this指针，而virtual函数一定要通过对象来调用，有隐藏的this指针。

2.静态函数是静态决议（编译的时候就绑定了）,而虚函数是动态决议的（运行时才绑定）。

6.纯虚函数就是没有函数体，同时在定义的时候，其函数名后面要加上“= 0”。包含纯虚函数的类称之为抽象类，不允许去实例化抽象类。

```
virtual int test()=0;
```

5.模板，类模板与函数模板

1.模板是创建泛型类或函数的蓝图或公式。

1.函数模板：

```
template <class type>
ret-type func-name(parameter list)
{
   // 函数的主体
}
```

2.类模板：

```
template <typename type>
class class-name {
. // 类的主体
.
```

```
.
}
```

2.模板定义语法中class和typename等价。

3.typename另外一个作用为：使用嵌套依赖类型。此时typename告诉编译器，其后面的字符串为一个类型名称，而不是成员函数或者成员变量。否则编译器不知道其后的字符串是类型还是一个成员名称，引发编译错误。

```
template<class T>
void MyMethod( T myarr )
{
    typedef typename T::LengthType LengthType;
    LengthType length = myarr.GetLength;
}
```

4.模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。

6.move的理解与转发

1.move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝。

2.完美转发(perfect forwarding)：是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。

1.std::forward：实现完美转化，按照参数本来的类型来转发出去，不管参数类型是T&&这种未定的引用类型还是明确的左值引用或者右值引用。

2.emplace_back：内部调用forward实现完美转发。

7.友元

1.类的友元是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。

2.友元类

```
friend  class  类名;
```

1.该友元类中所有函数都可以访问类的所有私有（private）成员和保护（protected）成员。

3.友元函数

```
friend  返回值类型  函数名(参数表);
```

4.友元关系不能被继承。

5.友元关系是单向的，不具有交换性。

6.友元关系不具有传递性。

4.高级主题

1.tuple类型，bitset类型

1.tuple类型

1.创建tuple

```
tuple<t1,t2,t3,t4...tn> tp=make_tuple(v1,v2,v3,v4...vn)
//或
tuple<t1,t2,t3,t4...tn> tp(v1,v2,v3,v4...vn)
```

2.解析tuple

```
get(t)      //  返回t的第i个数据成员
tie(v1,v2,v3,v4...vn)=tp  //获得tuple中的值并自动复制给v1,v2...
```

3.连接tuple

```
tuple_cat(tp1,tp2,....)
```

2.bitset类型

1.用字符串构造时，字符串只能包含 '0' 或 '1' ，否则会抛出异常。

2.在进行有参构造时，若参数的二进制表示比bitset的size小，则在前面用 0 补充；若比bitsize大，参数为整数时取后面部分，参数为字符串时取前面部分。

2.正则表达式

符号	含义
.(点)	匹配任意一个字符
[](中括号)	表示一个区间
*(星号)	配置一个或多个任意字符
+(加号)	配置零个或多个任意字符

符号	含义
?(问号)	配置零个或一个任意字符
{}(大括号)	表示一个计数区间
\	表示或，只能区多个中的一个
^(异或)	表示行的开始，或字符串中表示否定
$(美元符号)	表示行的末尾
(斜杠)	表示转义
\d	十进制数字0-9
\D	非十进制的字符
\w	表示一个字母或者数字
\W	表示一个非字母且非数字

1.正则表达式声明：

```
string str("\\d{4}");
regex pattern(str,regex::icase);        //声明为正则表达式

match_results<string::const_iterator> result;  //结果第一种保存方式
smatch result;      //结果第二种保存方式
```

1.匹配：regex_match(str1,result,str2)，如果str1(源字符串)和str2(正则表达式)匹配，返回true，否则返回false。result[0]是源字符串str1，result[1]是第一个匹配到的文本，result[2]是第二个匹配到文本.....

2.查找：regex_search(iter,iter_end,result,pattern1)，在迭代器iter和iter_end之间寻找与pattern1匹配的文本，并将结果保存在result中。result[0]是查找结果，result[0].second更新搜索起始位置。

4.替换：regex_replace(str,pattern,t),在str查找与pattern匹配的文本，并用t中的数据替换。

3.随机数

1.随机种子：随机种子(unsigned类型)是用来产生随机数的一个数。

2.C++中不能使用random()函数。因为random函数不是ANSI C标准，不能在gcc,vc等编译器下编译通过。

3.C++标准函数库提供一随机数生成器rand，返回0--RAND_MAX之间均匀分布的伪随机整数，RAND_MAX必须至少为32767。

4.srand（）可以指定不同的数（无符号整数变量）为种子。但是如果种子相同，则生成的伪随机数列也相同。

4.异常处理

1.C++ 通过 throw 语句和 try...catch 语句实现对异常的处理。

1.抛出一个异常

```
throw  表达式;
```

2.捕获异常

```
try {
    语句组
}
catch(异常类型) {
    异常处理代码
}
...
catch(异常类型) {
    异常处理代码
}
```

2.异常的再抛出

1.如果异常在本函数中没有被处理，则它就会被抛给上一层的函数。

3.标准异常类，从exception类派生出来。

1.bad_typeid：如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会抛出此异常。

2.bad_cast：在用 dynamic_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常。

3.bad_alloc：在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常。

4. out_of_range： 用 vector 或 string 的 at 成员函数根据下标访问元素时，如果下标越界，则会抛出此异常。使用[]访问则不会抛出out_of_range错误。

5.ios_base::failure

5.namespace命名空间

1.命名空间：实际上就是一个由程序设计者命名的内存区域，程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其他全局实体分隔开来。

```
namespace namespace_name {
   // 代码声明
}
```

2.using 指令：使用 using namespace namespace_name指令，告诉编译器，后续的代码将使用指定的命名空间中的名称，故此时使用命名空间时就可以不用在前面加上命名空间的名称。

6.多重继承以及虚继承

见面向对象程序设计节中继承。

7.new和delete的工作原理，重载的new和delete以及malloc、free的应用，定位new表达式

1.new 工作原理

1.简单数据类型（包括基本数据类型和不需要构造函数的类型）,简单类型直接调用operator new分配内存；可以通过new_handler来处理new失败的情况；new分配失败的时候不像malloc那样返回NULL，它直接抛出异常。要判断是否分配成功应该用异常捕获的机制。

2.复杂数据类型（需要由构造函数初始化对象），new复杂数据类型的时候先调用operator new分配内存，然后在分配的内存上调用构造函数。

2.new [] 工作原理

1.简单数据类型（包括基本数据类型和不需要析构函数的类型）,针对简单类型，new[]计算好大小后调用operator new。

2.复杂数据类型（需要由析构函数销毁对象），针对复杂类型，new[]会额外存储数组大小。

3.delete 工作原理

1.简单数据类型（包括基本数据类型和不需要析构函数的类型），delete简单数据类型默认只是调用free函数。

2.复杂数据类型（需要由析构函数销毁对象），delete复杂数据类型先调用析构函数再调用operator delete。

4.delete [] 工作原理

1.简单数据类型（包括基本数据类型和不需要析构函数的类型），根据new []额外保存的数组大小信息，对数组每个元素调用delete。

2.复杂数据类型（需要由析构函数销毁对象），根据new []额外保存的数组大小信息，对数组每个元素调用delete。

5.operator new和operator delete

6.new，delete和malloc，free对比

1.返回值不同：malloc 函数返回的是 void * 类型, new返回是对应的类的类型指针。

2.malloc需要指定分配的大小，new不需要指定大小。

3.malloc 只管分配内存，并不能对所得的内存进行初始化，new分配内存的时候同时进行初始化。

4.malloc是函数，而new是运算符。

5.内存分配的位置不同。new分配在自由存储区，malloc分配在堆上。

6.new内存分配失败时，会抛出bac_alloc异常，它不会返回NULL；malloc分配内存失败时返回NULL。

7.定位new表达式

1.在指定地址上分配内存，然后将大小合适的实例化对象放入该地址中。

```
    new (place_address) type(initializer-list)  //place_address必须是一个指针，initializer-list是类型的初始化列表。
```

8.重载的new和delete

1.目的：1.增加分配和归还的速度,2.堆碎片,3.检测运用上的错误,4.降低缺省内存管理器带来的空间额外开销.....

2.operator new()的返回值是一个void*,所做的是分配内存。它是编译器确保的动作，不在我们的控制范围之内。

3.operator delete()的参数是一个指向由operator new()分配的内存的void*,参数是一个void*是在调用析构函数后得到的指针,析构函数从存储单元里移去对象。operator delete()的返回类型是void。

4.重载new和delete

1.当重载operator new()和operator delete()时，我们只是改变了原有内存分配方法。

8.类中数据成员指针以及函数指针

1.函数指针

```
    int (*pf)(int,int); //pf是指向含两个形参的函数的函数指针
```

2.类中数据成员指针

```
    int class_name:: * pf // 指向class_name类中int数据成员的指针
```

3.类中成员函数指针

```
    int class_name:: *(pf)(int,int)  //指向class_name类中含两个形参的函数的函数指针
```

9.位域（传递二进制数据）以及volatile限定符（对类型额外修饰)

1.位域

```
    类型说明符 位域名：位域长度
    struct bs
    {                       //位域a占8位，位域b占2位，位域c占6位，无位域名占2位
```

```
    int a:8;
    int b:2;
    int c:6;
    int :2;
};
```

1.作用：节省内存资源，使数据结构更紧凑。

2.一个位域必须存储在同一个字节中，不能跨两个字节，故位域的长度不能大于一个字节的长度。

3.取地址操作符&不能应用在位域字段上。

4.位域字段不能是类的静态成员。

5.位域字段在内存中的位置是按照从低位向高位的顺序放置的。

6.当要把某个成员说明成位域时,其类型只能是int,unsigned int与signed int三者之一

7.位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。

8.位域的对齐

1.如果相邻位域字段的类型相同，且其位宽之和小于类型的sizeof大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止。

2.如果相邻位域字段的类型相同，但其位宽之和大于类型的sizeof大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍。

3.如果相邻的两个位域字段的类型不同,则各个编译器的具体实现有差异,VC6采取不压缩方式,GCC和Dev-C++都采用压缩方式。

4.整个结构体的总大小为最宽基本类型成员大小的整数倍。

5.如果位域字段之间穿插着非位域字段，则不进行压缩；（不针对所有的编译器）。

2.volatile限定符

1.易变性：下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取

2.不可优化性：volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

3.顺序性：能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。

10.链接指示：extern “C”（使用链接可以将C++代码与其他语言代码放在一起使用）

1.extern

1.extern可以置于变量或者函数前，以标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在其他模块中寻找其定义。

2.链接指定：当它与"C"一起连用时，如:

```
    extern "C" void fun(int a, int b);       //告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的。
```

3.与include相比，extern引用另一个文件的范围小，include可以引用另一个文件的全部内容。
