# C++ exercise code

## basic concept

### aboutarraypoint.cpp

|  |  |
| --- | --- |
| /\*  \* 关于复杂对象指针数组和数组指针  \* \*/  class MyClass  {  public:  int num;  char name[10];  };  int main()  {  //指针数组  MyClass \*parr = new MyClass[5];    //二级指针 | MyClass \*\*pparr = new MyClass\*[5]; //数组的每个元素都是一个指向MyClass的指针  //二维数组,数组每个元素都是MyClass  MyClass p2arr[5][5];  //指向数组的指针  MyClass (\*pp2arr)[5] = p2arr;  //强制转换将二级指针转换为指向数组指针  pp2arr = (MyClass (\*)[5])pparr;  //二位数主，每个数组的元素都是一个MyClass指针  MyClass \*p\_p2arr[5][5];  //声明指向数组的指针  MyClass \*(\*p\_pp2arr)[5] = p\_p2arr;  MyClass \*\*\*p\_pparr = (MyClass \*\*\*)p\_p2arr;  return 0;  } |

### 类型转换.cpp

|  |  |
| --- | --- |
| #include <iostream>  class Father  {  public:  int num;  void PrintName() {  std::cout << "this is father." << " num = " << num <<std::endl;  }  };  class Son : public Father  {  public:  int num;  void PrintName() {  std::cout << "this is father." << " num = " << num <<std::endl;  }  };  int main()  {  //指针没有初始化，就使用，这是危险的事  Father \*father;  Son \*son = reinterpret\_cast<Son \*> (father);  son->PrintName();  return 0;  }  int main03(int argc, char \*argv[])  {  //指针类型转换  int num = 5;  char \*p = reinterpret\_cast<char \*>(&num);  for(int i = 0; i < 4; i++) {  std::cout << static\_cast<int>(\*(p + i)) << " ";  } | std::cout <<"\n";  return 0;  }  int main02(int argc, char \*argv[])  {  const int num = 10;  int numarr[3] = {1,2,3};  const int \*p = numarr;  std::cout << \*p << " " << \*(p+1) << " "<< \*(p+2) <<std::endl;  //\*p = 10;  //\*(p+1) = 10; //报错，p为const类指针  int \*pnew = const\_cast<int \*>(p);  \*pnew = 10;  \*(pnew+1) = 11;  \*(pnew+2) = 12;  //去掉指针的const属性是成功的，但是对变量来说是不行的  //const变量是放在符号表中的，在编译的时候就确定了值，虽然在内存中修改  //变量的值，但是在运行的时候不是在内存中拿得，而是在寄存器取得  for(int i=0 ; i<3; i++) {  std::cout <<numarr[i] << std::endl;  }  //static\_cast<type>相当与c语言的中强制类转换  //std::cout << static\_cast<int>(100.0) << std::endl;  //去掉const属性  //int num01 = const\_cast<int> (num) ; //报错  return 0;  }  int main01(int argc, char \*argv[])  {  //static\_cast<type>相当与c语言的中强制类转换  std::cout << static\_cast<int>(100.0) << std::endl;    return 0;  } |

### virtualfunc.cpp

|  |  |
| --- | --- |
| /\*  \* 虚方法 虚析构函数可以解决父类指针指向子类delete发生的内存泄漏问题  \* 构造函数不能为虚析构函数  \* \*/  #include <iostream>  class Base  {  public:  int num;  void PrintNum() {  std::cout << "this is Base num = " << num << std::endl;  }  void Hi() {  std::cout << "Hi" << std::endl;  }  virtual void SayHello() {  std::cout << "virtual SayHello" << std::endl;  }  virtual ~Base() {  std::cout << "Base ～" << std::endl;  }  };  class Son : public Base  {  public:  void SayHello() {  std::cout << "this is Son SayHello" << std::endl;  }  virtual ~Son() {  std::cout << "~Son()" << std::endl;  }  };  class GreadSon : public Son  {  public:  int num;  void PrintNum() {  std::cout << "this is Greadson num = " << num << std::endl;  }  void GreadSonHi() {  std::cout << "Son Hi" << std::endl;  }  void SayHello() {  std::cout << "Greadson sayhello" << std::endl;  }  };  class SonSon : public GreadSon  {  public:  SonSon(int a, int b, int c)  {  this->a = a; | this->b = b;  this->c = c;  }  void SayHello() {  std::cout << "a = " << a << " b = " << b << " c = " << c << " " << std::endl;  }  void GetA() {  std::cout << " a = " << a << std::endl;  }  private:  int a;  int b;  int c;  };  int main()  {  Son son;  son.SayHello();  //指针父类调用父类的方法  //son.Base::SayHello();  //Base \*base = new Son;  //delete base;    //段异常错误 C语言类型的强制类型转换  //((GreadSon \*)base)->SayHello();  Base \*base(nullptr);  base = new GreadSon;  //base->SayHello();  base->PrintNum();  //用c++的强制类型转换则没有段错误  //(static\_cast<GreadSon \*>(base))->SayHello();    //虽然用把父类指针调用子类的特有的函数没有发生段异常错误，但是这是危险的事  //一旦这个方法中用到了子类的成员变量，就会可能发生down，但也能没有发生  //但这时危险的事  //(static\_cast<GreadSon \*>(base))->GreadSonHi();  //在ubuntu 14.04 LTS下测试没有down，但是在vs环境可能down  //(static\_cast<GreadSon \*>(base))->PrintNum();  //将父类指针转为子类指针  Base \*pbase = new Base;  GreadSon \*pson = static\_cast<GreadSon \*>(pbase);  //pson->PrintNum();  SonSon \*sonson = static\_cast<SonSon \*>(pbase);//new SonSon(1, 2, 3);  sonson->SayHello();  //在ubuntu 14.04 LTS 测试下没有问题，但是在vs下可能有问题存在  //sonson->GetA();  SonSon sonson02(1, 2, 3);  sonson02.GetA();  sonson02.Base::PrintNum();  return 0;  } |

### virtual\_class\_deep\_unstanderd.cpp

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class Base {  public:  virtual void f() { cout << "Base::f" << endl;};  virtual void g() { cout << "Base::g" << endl;};  virtual void h() { cout << "Base::h" << endl;};  };  class Drived : public Base {  public:  virtual void f() { cout << "Drived::f" << endl;};  // virtual void g() { cout << "Drived::g" << endl;};  // virtual void h() { cout << "Drived::h" << endl;};  };  typedef void (\*Fun)();  int main(void)  {  Base b;  Fun pFun = NULL;  cout << "virtual table address:" << (int \*)(&b) << endl;  cout << "virtual table first function address:" << (int \*)\*(int \*)(&b) << endl;  pFun = (Fun)\*((int \*)\*(int \*)(&b) + 0);  pFun();  //attention: add 2 because in x64 OS, pointer sizese 8  pFun =(Fun)\*((int \*)\*(int \*)(&b) + 2);  pFun();  pFun =(Fun)\*((int \*)\*(int \*)(&b) + 4);  pFun();  Drived d;  pFun = (Fun)\*((int \*)\*(int \*)(&d) + 0);  pFun();  pFun = (Fun)\*((int \*)\*(int \*)(&d) + 2);  pFun();  pFun = (Fun)\*((int \*)\*(int \*)(&d) + 4);  pFun();    pFun = (Fun)\*((int \*)\*(int \*)(&d) + 6);  cout << pFun << endl;    //this also invoke drived f function.  (static\_cast<Base \*>(&d))->f();  return 0;  }  class Base1 {  public:  virtual void f() { cout << "Base1:f" << endl;};  virtual void g() { cout << "Base1:g" << endl;};  virtual void h() { cout << "Base1:h" << endl;};  };  class Base2 { | public:  virtual void f() { cout << "Base2:f" << endl;};  virtual void g() { cout << "Base2:g" << endl;};  virtual void h() { cout << "Base2:h" << endl;};  };  class Base3 {  public:  virtual void f() { cout << "Base3:f" << endl;};  virtual void g() { cout << "Base3:g" << endl;};  virtual void h() { cout << "Base3:h" << endl;};  };  class Derive : public Base1, public Base2, public Base3 {  public:  virtual void f() { cout << "Devive:f" << endl;};  //virtual void g() { cout << "Devive:g" << endl;};  };  int main02(void)  {  Fun pFun = NULL;  Derive d;  long \*\*pVtab = (long \*\*)&d;  //Base1's vtable  pFun = (Fun)pVtab[0][0];  pFun();  pFun = (Fun)pVtab[0][1];  pFun();  pFun = (Fun)pVtab[0][2];  pFun();  //the tail of vtable  pFun = (Fun)pVtab[0][3];  cout << pFun << endl;    //Base2's vtable  pFun = (Fun)pVtab[1][0];  pFun();  pFun = (Fun)pVtab[1][1];  pFun();  pFun = (Fun)pVtab[1][2];  pFun();    //Base3's vtable  pFun = (Fun)pVtab[2][0];  pFun();  pFun = (Fun)pVtab[2][1];  pFun();  pFun = (Fun)pVtab[2][2];  pFun();  pFun = (Fun)pVtab[2][3];  cout << pFun << endl;  return 0;  } |

### using.cpp

|  |  |
| --- | --- |
| \*  \* using 别名  \* \*/  #include <iostream>  namespace space //隔离模板，避免冲突  {  template<class T> using ptr = T\*; //模板的简写  }  int add(int a, int b)  {  return a+b;  } | typedef int(\*padd)(int a, int b);  using FUN = int (\*)(int a, int b); //利用using起别名  int main()  {  padd p = add;  std::cout << p(1, 2) << std::endl;  FUN fun = add;  std::cout << p(2, 2) << std::endl;  space::ptr<int> pint(new int(5));  std::cout << \*pint << std::endl;  return 0;  } |

### tuplearray.cpp

|  |  |
| --- | --- |
| /\*\*  \* 多元数组 tuple  \*/  #include <iostream>  #include <typeinfo>  #include <map>  int main()  {  int num = 10;  double doublenum = 20.0;  char ch = 'A';  char \*str = "hello tuple";  //必须是个静态数组 | std::tuple<int, double, char, const char \*> mytuple(num, doublenum, ch, str);  //const int num = 3;  //下标只能是常量  auto data0 = std::get<0>(mytuple);  auto data1 = std::get<1>(mytuple);  auto data2 = std::get<2>(mytuple);  auto data3 = std::get<3>(mytuple);  std::cout << typeid(num).name() << std::endl;  decltype(data0) dataA; //获取data0数据类型 再次创建  std::cout << data0 << " " << data1 << " "<< data2 << " "<< data3 << " ";  return 0;  } |

### templatereference.cpp

|  |  |
| --- | --- |
| /\*  \* 模板引用 std:ref 不是std number function  \* \*/  #include <iostream>  using namespace std;  template<class T>  void com(T arg)  {  std::cout << "com = " << &arg << "\n";  arg++;  }  template<class T>  void com01(T &arg)  {  std::cout << "com = " << &arg << "\n";  arg++;  } | int main()  {  int count = 10;  //不能修改count的值  com(count);  std::cout << count << "\n";  //传递引用也不能修改count值  int &ccount = count;  com(count);  std::cout << count << "\n";  //模板引用  com01(count);  std::cout << count << "\n";  com(std::ref(count));  std::cout << count << "\n";  return 0;  } |

### templateoverload.cpp

|  |  |
| --- | --- |
| //模板函数的重载  #include <iostream>  #include <array>  using std::array;  template<typename T>  void showarray(array<T,10> myarray, int n)  {  using namespace std;  cout << "TTT" << endl;  for (int i = 0; i<n; i++) {  cout << myarray[i] << " ";  }  cout<< endl;  }  template<typename T>  void showarray(array<T \*,10> myarray, int n)  {  using namespace std;  cout << "T\*T\*T\*" <<endl;  for(int i = 0; i < n; i++) {  cout << \*myarray[i] << " "; | }  cout << endl;  }  int main(int argc, char \*argv[])  {  array<int, 10> intarray = {1,2,3,4,5,6,7,8,9,10};  array<int \*, 10> pintarray;  for(int i = 0; i < 10; i++) {  pintarray[i] = &intarray[i];  }  array<int \*\*, 10> ppintarray;  for(int i = 0; i < 10; i++) {  ppintarray[i] = &pintarray[i];  }  showarray(intarray, 10);  showarray(pintarray, 10);  showarray(ppintarray, 10);  return 0;  } |

### templatemeta.cpp

|  |  |
| --- | --- |
| /\*  \* 模板元编程  \* \*/  //主要思想  //  //利用模板特化机制实现编译期条件选择结构，利用递归模板实现编译期循环结构，模板元程序则由编译器在编译期解释执行。  //  //优劣及适用情况  //  //通过将计算从运行期转移至编译期，在结果程序启动之前做尽可能多的工作，最终获得速度更快的程序。也就是说模板元编程的优势在于：  //  //1.以编译耗时为代价换来卓越的运行期性能（一般用于为性能要求严格的数值计算换取更高的性能）。通常来说，一个有意义的程序的运行次数（或服役时间）总是远远超过编译次数（或编译时间）。  //  //2.提供编译期类型计算，通常这才是模板元编程大放异彩的地方。  //  //模板元编程技术并非都是优点：  //  //1.代码可读性差，以类模板的方式描述算法也许有点抽象。  //  //2.调试困难，元程序执行于编译期，没有用于单步跟踪元程序执行的调试器（用于设置断点、察看数据等）。程序员可做的只能是等待编译过程失败，然后人工破译编译器倾泻到屏幕上的错误信息。  //  //3.编译时间长，通常带有模板元程序的程序生成的代码尺寸要比普通程序的大，  //  //4.可移植性较差，对于模板元编程使用的高级模板特性，不同的编译器的支持度不同。  //模板元把运行时消耗的时间，在编译期间进行优化，所以运行速度快，但是编译时间会慢  #include <iostream>  //斐波那契函数  template<int N>  struct data  {  enum a{ res = data<N - 1>::a::res + data<N - 2>::a::res };  }; | template<>  struct data<1>  {  enum a{res = 1};  };  template<>  struct data<2>  {  enum a{res = 2};  };  int GetData(int num)  {  if( num == 1) {  return 1;  }  else if( num == 2) {  return 2;  }  else {  return GetData(num -1) + GetData(num - 2);  }  }  int main()  {  const int num = 40;  int result = data<num>::res; //内部不能包含变量  std::cout << result << std::endl;  std::cout << GetData(40) << std::endl;  return 0;  } |

### TemplateMakeFile

|  |
| --- |
| EXE := obj  CC := g++  CFALGS := -Wall -g  STANDARD := --std=c++11  # all cpp source files.  SOURCE := $(wildcard \*.cpp)  # substitute .cpp prefix to .o prefix  OBJS := $(patsubst %.cpp, %.o, $(SOURCE))  all:$(EXE)  $(EXE):$(OBJS)  $(CC) $(CFALGS) $(STANDARD) $(OBJS) -o $(EXE)  .PHONY: clean  clean:  -rm -rf $(EXE) $(OBJS) \*.o |

### templatecover.cpp

|  |  |
| --- | --- |
| /\*  \*函数模板覆盖  \* \*/  #include <iostream>  struct info  {  char name[40];  double db;  int data;  };  template<typename T>  void swap(T &a, T &b)  {  std::cout << "通用函数模板" << std::endl;  T temp = a;  a = b;  b = temp;  }  //模板为空，明确参数类型，覆盖函数 模板的类型  void swap(info &info01, info &info02)  {  std::cout << "特有函数模板" << std::endl;  //通用模板可以实现通用，针对自己的数据类型做出优化  info tmp = info01; | info01 = info02;  info02 = tmp;  }  //通过对通用函数的指定实例化swap<info> ，同样可以调用通用函数  //实例化调用 swap<T>  int main02()  {  info info01 = {"linux", 20.9, 10};  info info02 = {"unix", 30.9, 30};  swap<info>(info01, info02);  std::cout << info01.name << " " << info01.db << " " << info01.data <<std::endl;  std::cout << info02.name << " " << info02.db << " " << info02.data <<std::endl;  return 0;  }  //调用特有函数模板  int main01()  {  info info01 = {"linux", 20.9, 10};  info info02 = {"unix", 30.9, 30};  swap(info01, info02);  std::cout << info01.name << " " << info01.db << " " << info01.data <<std::endl;  std::cout << info02.name << " " << info02.db << " " << info02.data <<std::endl;  return 0;  } |

### rtti.cpp

|  |  |
| --- | --- |
| /\*  \* 实时类型检测  \* typeid,dynamic\_cast 必须依赖于虚函数表  \* 类型不匹配转换失败，返回为空，类型安全  \* \*/  #include <iostream>  #include <typeinfo>  class A  {  public:  int num;  virtual void run()  //void run()  {  std::cout << "Arun\n";  }  };  class B : public A  {  public:  int num = 0;  void run()  {  std::cout << "Brun\n";  }  void test()  {  std::cout << num << std::endl;  std::cout << "btest\n";  }  };  int main(int argc, char \*argv[])  {  A a1;  B b1;  A \*p1 = &a1;  A \*p2 = &b1;  B \*p3(nullptr);  p3 = static\_cast<B \*>(p1); //直接赋值地址，不安全，于虚函数无关  //p3->test();  p3 = reinterpret\_cast<B \*>(p2);  p3->test(); | return 0;  }  int main03(int argc, char \*argv[])  {  A a1;  B b1;  A \*p1 = &a1;  A \*p2 = &b1;  B \*p3(nullptr);  //类的空指针可以调用不调用内部数据成员的函数，一旦函数中使用了内部的成员变量，就会shutdown  //p3->test();  p3 = dynamic\_cast<B \*>(p2);  //dynamic\_cast必须要有虚函数表，根据虚函数表进行转换，否则不能转换  //在这里如果run函数不是虚函数，语法编译就不能通过  //如果转换失败，p3为空，类的空指针可以调用不调用内部数据成员的函数  //转换成功，就是地址  std::cout << p3 << std::endl;  p3->test();  return 0;  }  int main02(int argc, char \*argv[])  {  A a1;  B b1;  A \*p1 = &a1;  A \*p2 = &b1;  std::cout << typeid(p1).name() << " " << typeid(p2).name() << std::endl;  std::cout << ( typeid(p1).name() == typeid(p2).name() )<< std::endl;  //根据虚函数表判断类型  std::cout << typeid(\*p1).name() << " " << typeid(\*p2).name() << std::endl;  return 0;  }  int main01(int argc, char \*argv[])  {  B b1;  b1.num = 10; //覆盖现象  b1.A::num = 20;  std::cout << b1.num << " " << b1.A::num << std::endl;  std::cout << &b1.num << " " << &b1.A::num << std::endl;  return 0;  } |

### referenceinsidefunctionbind.cpp

|  |  |
| --- | --- |
| /\*  \* 引用内部函数绑定机制  \*\*/  #include <iostream>  #include <functional>  using namespace std;  using namespace std::placeholders;  //仿函数，创建一个函数指针，引用一个结构体内部的或者一个类内部的共有函数  struct MyStruct  {  void add(int a)  {  cout << a << endl;  }  void add01(int a, int b)  {  cout << a + b << endl;  }  void add02(int a, int b, int c)  {  cout << a + b + c << endl;  }  }; | int main()  {  MyStruct mystruct;  //创建函数指针，类结构体，数据私有，代码共享  //函数通过调用，调用需要传递对象名进行区分  void (MyStruct::\*p)(int a) = &MyStruct::add;  p(1);  return 0;  }  int main01()  {  MyStruct mystrcut01;  //auto自动变量，地址，函数指针bind绑定  auto func = bind(&MyStruct::add, &mystrcut01, \_1); //参数位置  auto func02 = bind(&MyStruct::add01, &mystrcut01, \_1, \_2);  auto func03 = bind(&MyStruct::add02,mystrcut01, \_1, \_2, \_3);  func(2);  func02(2, 3);  func03(2, 3, 4);  return 0;  } |

### overwriteoperator.cpp

|  |  |
| --- | --- |
| /\*  \* 运算符重载 << >> ++ --  \* \*/  #include <iostream>  using namespace std;  class Complex  {  public:  Complex(int realnum, int virtualnum):realnum(realnum), virtualnum(virtualnum)  {    }  Complex()  {  this->realnum = 0;  this->virtualnum = 0;  }  Complex(const Complex &com)  {  this->realnum = com.realnum;  this->virtualnum = com.virtualnum;  std::cout << "copy constructor" << std::endl;  }  friend ostream & operator<<(ostream &out, const Complex &com);  friend istream & operator>>(istream &in, Complex &com);  Complex operator+(Complex &com02);  void operator=(Complex &com02);  //前置++  Complex operator++();  //后置++ int占位参数  Complex operator++(int);  //前置--  Complex operator--();  //后置--  Complex operator--(int);  private:  int realnum;//实数  int virtualnum;//虚数  };  ostream & operator<<(ostream &out, const Complex &com)  {  out << com.realnum << " + " << com.virtualnum << "i" <<"\n";  return out;  }  istream & operator>>(istream &in, Complex &com)  {  std::cin >> com.realnum;  std::cin >> com.virtualnum;  return in;  }  Complex Complex::operator+(Complex &com02)  {  return Complex(this->realnum + com02.realnum, this->virtualnum + com02.virtualnum);  }  void Complex::operator=(Complex &com02)  {  std::cout << "assign =" << std::endl;  this->realnum = com02.realnum;  this->virtualnum = com02.virtualnum;  }  Complex Complex::operator++()  {  ++this->realnum;  ++this->virtualnum;  return \*this;  }  Complex Complex::operator++(int) | {  Complex tempComplex;  tempComplex.realnum = this->realnum++;  tempComplex.virtualnum = this->virtualnum++;  return tempComplex;  }  Complex Complex::operator--()  {  --this->realnum;  --this->virtualnum;  return \*this;  }  Complex Complex::operator--(int)  {  Complex com;  com.realnum = this->realnum--;  com.virtualnum = this->virtualnum--;  return com;  }  class ComplexArray  {  public:  ComplexArray()  {  for (int i = 0; i < 5; i++) {  mComplexList[i] = new Complex;  }  }  Complex \*mComplexList[5];  //重载[]  Complex \* operator[](int index);  };  Complex \* ComplexArray::operator[](int index)  {  return this->mComplexList[index];  }  int main()  {  Complex com(1, 2);  std::cout << com ;    //std::cin >> com;  //std::cout << com ;  Complex com01(2, 3);  Complex com02 = com + com01;  std::cout << com02;  Complex com03(com02);  std::cout << com03;  com = com01;  std::cout << com;  //前置++  ++com;  std::cout << com;    //后置++  com++;  std::cout << com;  //前置--  --com;  std::cout << com;  //后置--  com--;  std::cout << com;  ComplexArray array;  std::cout << \*array[1];  return 0;  } |

### overloadfunction.cpp

|  |  |
| --- | --- |
| /\*  \* 重载函数原理  \* \*/  #include <iostream>  void add(int num01, int num02)  {  } | void add(int num02)  {  }  int main()  {  return 0;  } |

### newarray.cpp

|  |  |
| --- | --- |
| \*\*  \* new type array  \*/  #include <iostream>  #include <array>  #include <string>  #include <stdlib.h>  #include <vector> //c++的标准库  using std::vector;  using std::array;  using std::string;  int main()  {  vector<string> str01;  str01.push\_back("notepadqq");  str01.push\_back("firefox");  str01.push\_back("git");  //反向迭代器  vector<string>::reverse\_iterator rbegin = str01.rbegin();  vector<string>::reverse\_iterator rend = str01.rend();  //注意：rend不是指向最后一个数据，而是指向结尾的下一个结点  A: if (rbegin != rend) {  std::cout << \*--rend << "";  goto A;  }  return 0;  }  int main06()  {  vector<string> str01;  str01.push\_back("notepadqq");  str01.push\_back("firefox");  str01.push\_back("git");  vector<string>::iterator ibegin, iend; //迭代器  ibegin = str01.begin(); //data start  iend = str01.end(); //data end  //正向迭代  //for(; ibegin != iend; ibegin++) { | // std::cout << \*ibegin << " "; //获取指针指向的数据  //}  //反响迭代  while(iend != ibegin) {  std::cout << \*--iend << " ";  }  return 0;  }  int main05()  {  vector<string> str01; //动态字符串数组  //可以反复利用  str01.push\_back("notepadqq");  str01.push\_back("firefox");  str01.push\_back("git");  str01.pop\_back(); //删除最后一个  for(int i = 0; i < str01.size(); i++) {  std::cout << str01[i] << " ";  }  return 0;  }  int main04()  {  array<int, 5> myint1 = {1, 2, 3, 4, 5};  array<int, 5> myint2 = {6, 7, 8, 9, 10 };  array<int, 5> myint3 = {11, 12, 13, 14, 15};  //第一种方式初始化  //array<array<int,5>,3> myint = {myint1, myint2,myint3};  //第二种方式初始化  array<array<int, 5>, 3> myint = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11  , 12, 13, 14, 15};  for(int i = 0; i < 3; i++) {  for(int j = 0; j < 5; j++) {  std::cout<< myint[i][j] << " ";  }  std::cout << "\n";  }  return 0;  } |

### newadvanceuse.cpp

|  |  |
| --- | --- |
| /\*  new的高级用法  \*/  #include <iostream>  #include <new>  const int buf(512); //限定一个常量整数 512  int N(5); //数组的长度  char buffer[buf] = {0}; //存储在静态区  //总结：  //p1，p3，p5作为指针变量在戦区，存储的地址指向堆区  //需要手动回收内存  //p2,p4,p6 作为指针变量在戦区，存储的地址在静态区。也叫缓冲区  //自动释放内存，用于分配用完了就不会再用的数据  //避免了内存泄漏，自动释放内存。牺牲了内存访问的独立性  using namespace std;  class MyClass  {  public:  MyClass()  {  std::cout << "创建\n";  }  ~MyClass()  {  std::cout << "销毁\n";  }  };  int main()  {  char \*pcathe = new char[1024];  char \*pcatheend = pcathe + 1024;  std::cout << (void \*)pcathe << " " << (void \*)pcatheend << std::endl;  //限定区域内存分配，覆盖模式 | MyClass \*pmyclass = new (pcathe)MyClass[10];  //不需要delete[]pmyclas，会自动覆盖  std::cout << pmyclass << "\n";  pmyclass = new (pcathe)MyClass[10];  std::cout << pmyclass << "\n";  return 0;  }  int main01()  {  double \*p1, \*p2;  p1 = new double[N]; //分配内存大小为N个元素  p2 = new (buffer)double[N]; //指定区域分配内存大小  for(int i = 0; i < N; i++) {  p1[i] = p2[i] = i + 10.8; //数组初始化  std::cout << "p1 value " << &p1[i] << " " << p1[i];  std::cout << " p2 value " << &p2[i] << " " << p2[i] << std::endl;  }  double \*p3 = new double[N]; //分配内存大小为N个元素  double \*p4 = new (buffer)double[N]; //指定区域分配内存大小  for(int i = 0; i < N; i++) {  p3[i] = p4[i] = i + 10.8; //数组初始化  std::cout << "p3 value " << &p3[i] << " " << p3[i];  std::cout << " p4 value " << &p4[i] << " " << p4[i] << std::endl;  }  double \*p5 = new double[N]; //分配内存大小为N个元素  double \*p6 = new (buffer)double[N]; //指定区域分配内存大小  for(int i = 0; i < N; i++) {  p5[i] = p6[i] = i + 10.8; //数组初始化  std::cout << "p5 value " << &p5[i] << " " << p5[i];  std::cout << " p6 value " << &p6[i] << " " << p6[i] << std::endl;  }  //注意上述代码已造成内存泄漏  return 0;  } |

### mutable.cpp

|  |  |
| --- | --- |
| /\*  mutabel key word  \*/  #include <iostream>  using std::cout;  using std::cin;  using std::endl;  class A  {  public:  A() = default;  ~A()  {    } | void TestMutable() const  {  mutablevar++;  cout << "test mutable var. mutablevar value is " << mutablevar << endl;  }  private:  mutable int mutablevar = 0;  };  int main(int argc, char \*\*argv)  {  const A a;  a.TestMutable();  a.TestMutable();  a.TestMutable();  return 0;  } |

### Makefile

|  |
| --- |
| CC=g++  CFLAG := -Wall -g  inheritclass:inheritclass.cpp  $(CC) $(CFLAG) $^ -o $@.out  .PHONY: clean  clean:  -rm -rf \*.o \*.out |

### main.cpp

|  |  |
| --- | --- |
| #include <QtGui/QImage>  #include <QtCore/QCoreApplication>  #include <QtCore/QFileInfo>  #include <QtCore/QFile>  #include <QtCore/QDir>  #include <QtCore/QDebug>  #include <QtCore/QTime>  #define PICTURE\_DIR "channel0"  #define SUFFIX\_FILE ".prot"  #define PEER\_INDEX\_COUNT (8+4+4) //peer index byte count  #define INDEX\_TABLE\_BYTE\_COUNT 1608 //8(redundance) + (8+4+4)\*100  bool WritToDisk()  {  QDir currentdir = QDir(QCoreApplication::applicationDirPath());  QFileInfoList fileinfolist = ((QDir)(currentdir.filePath(PICTURE\_DIR))).entryInfoList(QDir::Files);  //use memeset function  QByteArray contentarray;  for (int i=0; i<INDEX\_TABLE\_BYTE\_COUNT; ++i) { //fill 0x00 -> 1608  contentarray.append((char)0x00);  }  int packfileindex = 0;  quint64 filebytescount = INDEX\_TABLE\_BYTE\_COUNT;  QFile \*filepoint = NULL;  for (int i=0; i<fileinfolist.count(); ++i) {  if (! fileinfolist[i].suffix().isEmpty()) {  continue;  }  QString filename = fileinfolist[i].baseName();  quint64 time = QTime(0, 0, 0).msecsTo(QTime::fromString((filename)));  int temppackfileindex = time/(2000\*100); //2000ms\*100; calc packfile index;  if (packfileindex != temppackfileindex) { //packfile index changed.  packfileindex = temppackfileindex;  contentarray.clear();  filebytescount = INDEX\_TABLE\_BYTE\_COUNT;  //close last file point  filepoint->close();  delete filepoint;  filepoint = NULL;  for (int j=0; j<INDEX\_TABLE\_BYTE\_COUNT; ++j) { //fill 0x00 -> 1408  contentarray.append((char)0x00);  }  }  int table\_index = (time-(2000\*100\*temppackfileindex))/2000; // calc table index, from 0 start  if (filepoint == NULL) {  filepoint = new QFile (QString::number(temppackfileindex)+SUFFIX\_FILE);  }  if (! filepoint->isOpen()) {  if (! filepoint->open(QIODevice::WriteOnly)) {  qDebug() << "file open.";  continue;  }  }  QFile imagefile(fileinfolist[i].absoluteFilePath());  if (! imagefile.open(QIODevice::ReadOnly)) {  qDebug() << "imagefile open.";  continue;  }  QByteArray imagebyte = imagefile.readAll();  imagefile.close();  int imagecount = imagebyte.count();  QByteArray indexinfoarray;  indexinfoarray.append(filename);  indexinfoarray.append(filebytescount & 0xFF);  indexinfoarray.append((filebytescount >> 8) & 0xFF);  indexinfoarray.append((filebytescount >> 16) & 0xFF);  indexinfoarray.append((filebytescount >> 24) & 0xFF);  indexinfoarray.append(imagecount & 0xFF);  indexinfoarray.append((imagecount >> 8) & 0xFF);  indexinfoarray.append((imagecount >> 16) & 0xFF);  indexinfoarray.append((imagecount >> 24) & 0xFF);  if (0 == filebytescount - INDEX\_TABLE\_BYTE\_COUNT) { //for the first time need write table index bytes.  //write table index  if (filepoint->write(contentarray) != contentarray.size()) {  continue;  }  }  if (! filepoint->seek(table\_index\*PEER\_INDEX\_COUNT+8)) {  continue;  } | if (filepoint->write(indexinfoarray) != indexinfoarray.count()) {  continue;  }  if (! filepoint->seek(filebytescount)) {  continue;  }  if (filepoint->write(imagebyte) != imagebyte.count()) {  continue;  }  filebytescount += imagecount;  if (i == fileinfolist.count()-1) { // last file, need to close file, delete file point  filepoint->close();  delete filepoint;  filepoint = NULL;  }  } //for i  return true;  }  bool ReadFromDisk(const QString &pImageName)  {  if (pImageName.isEmpty()) {  return false;  }  quint64 time = QTime(0, 0, 0).msecsTo(QTime::fromString((pImageName)));  int temppackfileindex = time/(2000\*100); //2000ms\*100; calc packfile index;  int table\_index = (time-(2000\*100\*temppackfileindex))/2000; // calc table index, from 0 start  QDir currentdir = QDir(QCoreApplication::applicationDirPath());  QFile file(currentdir.filePath(QString::number(temppackfileindex)+SUFFIX\_FILE));  if (! file.open(QIODevice::ReadOnly)) {  return false;  }  if (! file.seek(8+table\_index\*PEER\_INDEX\_COUNT)) {  file.close();  return false;  }  QByteArray array = file.read(PEER\_INDEX\_COUNT);  if (array.isEmpty()) {  file.close();  return false;  }  quint64 position = (array[8] & 0xFF) | ((array[9] << 8) & 0xFFFF) | ((array[10] << 16) & 0xFFFFFF) | ((array[11] << 24) & 0xFFFFFFFF);  quint64 length = (array[12] & 0xFF) | ((array[13] << 8) & 0xFFFF) | ((array[14] << 16) & 0xFFFFFF) | ((array[15] << 24) & 0xFFFFFFFF);  if (length == 0) {  qDebug() << "Not find file name " << pImageName;  return false;  }  if (! file.seek(position)) {  file.close();  return false;  }  QByteArray imagebyte = file.read(length);  if (imagebyte.isEmpty()) {  file.close();  return false;  }  QImage image = QImage::fromData(imagebyte);  if (! image.save("/home/liushixiong/Desktop/thumbnailpack/2.png")) {  qDebug() << "save image error.";  return false;  }  return true;  }  int main(int argc, char \*argv[])  {  QCoreApplication app(argc, argv);  // if (! WritToDisk()) {  // qDebug() << "Write to disk error.";  // }  // if (! ReadFromDisk("00:03:42")) {  // qDebug() << "Not find.";  // }  return app.exec();  } |

### lambda.cpp

|  |  |
| --- | --- |
| /\*  \* lambda表达式  \* \*/  #include <iostream>  #include <vector>  #include <functional>  #include <algorithm>  using namespace std;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*notice:  \*[](){cout << "hello world!" << endl}; 这是一个函数指针  \*[](){cout << "hello world!" << endl}(); 这是函数调用  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int main01()  {  auto fun1 = [](){ cout <<"hello world!" << endl; };  fun1();  auto func2 = [](int a, int b){ cout << a + b << endl << endl; };  func2(1, 2);  return 0;  }  int main02()  {  vector<int> myvector;  myvector.push\_back(1);  myvector.push\_back(2);  myvector.push\_back(11);  auto fun1 = [](int v){cout << v << endl;};  for\_each(myvector.begin(), myvector.end(), fun1);  return 0;  }  int main03()  {  vector<int> myvector;  myvector.push\_back(1);  myvector.push\_back(2);  myvector.push\_back(11);  int a = 10;  //= 表示知道a的存在，可以引用，只能读，不可以写或者是修改，引用当前块语句内部的局部变量  auto fun1 = [=](int v){ v += a; cout << v << endl;};  for\_each(myvector.begin(), myvector.end(), fun1);  cout << a << endl;  return 0;  } | int main05()  {  vector<int> myvector;  myvector.push\_back(1);  myvector.push\_back(2);  myvector.push\_back(11);  int a = 10, b = 0;  //&引用块语句中的所有变量， &a表示引用块语句中a变量  auto fun1 = [&a](int v) { a += 20, v += a; cout << v << endl;};  //auto fun1 = [&](int v) { a += 20, b = 10, v += a; cout << v << endl;};  for\_each(myvector.begin(), myvector.end(), fun1);  cout << "a:" << a << " b:" << b <<endl;  return 0;  }  class test  {  public:  vector<int> myvector;  int num;  public:  test()  {  num = 12;  myvector.push\_back(10);  myvector.push\_back(11);  }  void add() {  //[]引用this  int x = 3;  auto fun1 = [this, x](int v) { cout << v + x + this->num << endl;};  for\_each(myvector.begin(), myvector.end(), fun1);  }  };  int main()  {  //lambda返回值  auto fun1 = []()->double{ cout << "china" << endl; return 1;};  cout << fun1() << endl;  int a = 10;  auto fun2 = [a](int v)mutable->double{ v += a; cout << v << endl; a = 3; return 3; };  cout << fun2(1) << endl;  cout << a << endl;  return 0;  } |

### inheritclass.cpp

|  |  |
| --- | --- |
| /\*  \* 关于类的继承  \* \*/  #include <iostream>  #include <stdio.h>  #include <string.h>  class Father  {  public:  Father()  {  mChar = new char[10];  strcpy(mChar, "linux");  std::cout << "this is father constructor" << std::endl;  }  //将父类析构函数声明为虚函数，这样在父类在析构的时候，可以调用子类的析构函数，避免内存泄漏的危险  virtual ~Father()  {  delete mChar;  std::cout << "Father desconstruct..." << std::endl;  }  Father(const Father &pFather)  {  std::cout << "Father copy ...." << std::endl;  }  Father& operator=(const Father &pFather)  {  std::cout << "Father operator = " << std::endl;  return \*this;  }  void PrintName()  {  std::cout << "this is father." << std::endl;  }  void GetChar()  {  std::cout << mChar << std::endl;  }  protected:  void ProFun()  {  std::cout << "this is father profunc." << std::endl;  }  private:  void PriFun()  {  std::cout << "this is father prifunc......" << std::endl;  }  private:  char \*mChar;  };  class Son: public Father  {  public:  Son()  {  std::cout << "Son construct...\n";  }  ~Son()  {  std::cout << "Son desconstruct...\n";  }  Son(const Son &pSon)  {  std::cout << "Son copy construct...\n";  }  Son& operator=(const Son &pSon)  {  std::cout << "Father operator = .\n";  return \*this;  }  void PrintName()  {  std::cout << "this is son." << std::endl;  ProFun();  }  void WhoAmI() {  std::cout << "this is Son." << std::endl;  }  protected:  void ProFun()  {  std::cout << "this is son profunc." << std::endl;  //调用父类的保护方法  this->Father::ProFun();  } | private:  void PriFun()  {  std::cout << "this is son prifunc......" << std::endl;  }  };  class Sonson : protected Son  {  public:  void PrintName(){  std::cout << "this is Sonson." << std::endl;  this->Son::WhoAmI();  }  };  int TestFun()  {  Father \*father = new Son();  delete father;  return 0;  }  int TestFun01()  {  Father father = Son();  father.PrintName();  //此时父类已经将资源释放了，这里调用没有shutdown，也是很危险的，其他平台可能就会shutdown  father.GetChar();  return 0;  // 将子类对象赋值给父类是危险的行为，构造子类的时候先是构造父类对象，然后子类对象析构，再析构父类(此时父类可能释放资源了), 可能将匿名对象转为父类对象，这是匿名对象的析构可能会将父类的已经释放的资源在释放一次，导致bug出现。  // console print content  //  // this is father constructor  // Son construct...  // Son desconstruct...  // Father desconstruct...  // this is father.  // Father desconstruct...  }  void TestFunc02()  {  //编译器报错  //父类对象不能赋值给子类对象  //Son son = Father(); //error  }  void TestFunc03()  {  //编译器报错  //父类指针对象不能赋值给子类指针对象  //Son \*son = new Father();//error  }  int main()  {  Son son;  Father &father = son;  father.PrintName();  return 0;  }  int main02()  {  //TestFun();  //TestFun01();  //TestFunc02();  //TestFunc03();  Son son;  Son son01;  //赋值运算符 只会调用子类的不会调用父类的赋值运算符  son = son01;  return 0;  }  int main01()  {  //Father father;  //father.PrintName();  Son son;  son.PrintName();  //子类想调用父类重名的方法 子类对象中存在了一个指向父类的对象的指针  //son.Father::PrintName();  //protected保护方法只能在内部使用  //son.ProFun(); //报错 error  std::cout << "-------------------------------" << std::endl;  Sonson sonson;  sonson.PrintName();  //如果是public继承WhoAmI可以访问，如果是protected继承，这个方法不能被访问，只能在内部使用，protect继承，是一脉相传  //sonson.WhoAmI();  //private 继承，下面的子类不能在继承  return 0;  } |

### Hanoi.cpp

|  |  |
| --- | --- |
| /\*  \* 汉诺塔 递归算法  \* \*/  #include <iostream>  void Hanoi(int num, char A, char B, char C)  {  if(num <= 1) {  std::cout << A << "-->" << C << std::endl;  return ;  }  else {  Hanoi(num - 1, A, C, B); | std::cout << A << "-->" << C <<std::endl;  Hanoi(num - 1, B, A, C);  }  }  int main()  {  int n;  std::cin >> n;  std::cout << "n = " << n <<std::endl;  Hanoi(n,'A', 'B', 'C');  return 0;  } |

### functemplate.cpp

|  |  |
| --- | --- |
| //函数模板选择问题  #include <iostream>  //函数模板可以对类型进行优化重载，根据类型会覆盖  //如果仍然需要使用模板函数，需要实例化  template<typename T>  T add(T a, T b)  {  std::cout << "T add" << std::endl;  return a + b;  }  //普通函数  int add(int a, int b) | {  std::cout << "int add" << std::endl;  return a + b;  }  int main()  {  int a = 10, b = 20;  double ab = 10.0, ac = 20.0;  add(ab, ac);  add(a, b);  add<int>(a, b);  return 0;  } |

### escapecharacter.cpp

|  |  |
| --- | --- |
| /\*  \* 转义字符  \* \*/  #include <iostream>  #include <string>  #include <stdlib.h>  int main()  { | //在linux下路径都是45度的斜杠  std::string path = "~/code/c\_code/a.out";  //在windows下  //path = R"(c:\program Files\Tencent\QQ\...)";  system(path.c\_str());  double dou = 1.2;  std::cout << ++dou <<std::endl;  return 0;  } |

### emptyclass.cpp

|  |  |
| --- | --- |
| /\*  \* 不含任何属性和方法的空类  \* \*/  #include <iostream>  //空类不含任何属性和方法，但是要注意它含有两个方法，构造和析构函数  class MyClass  {  public:  int a;  virtual void test();  // void test1()  // {  // ;  // }  };  int main() | {  //空类大小 1  //std::cout << sizeof(MyClass) <<std::endl;  //添加int a属性 大小为4  //std::cout << sizeof(MyClass) <<std::endl;  //如果再添加方法 大小还是为4，因为方法是共有的，再代码区中  std::cout << sizeof(MyClass) <<std::endl;  //但是要注意一旦添加虚方法之后大小变为16，光有一个虚方法大小为8，猜测一个指针大小为8(再x64平台下8,再x86平台下为4)      return 0;  } |

### cppunion.cpp

|  |  |
| --- | --- |
| /\*  \* c++ 的联合体  \* \*/  #include <iostream>  /\*  \* union 的本质也是个类，可以有内部函数  \* 内部数据是共享的，不同对象之间是独立的，函数代码也是共享  \* 具备结构体的所有功能，不能实现继承  \* \*/  union MyUnion  {  int num;  double db;  void go()  {}  };  union MyUnionA  { | int num;  double db;  void go()  {}  };  int main()  {  //大小为8  std::cout << sizeof(MyUnion) << std::endl;  MyUnion A;  A.num = 10;  std::cout << A.num << std::endl;  //打印db时出现乱码现象，是因为把int类型当做double解析  std::cout << A.db << std::endl;    return 0;  } |

### cppstruct.cpp

|  |  |
| --- | --- |
| /\*  \* c++ 结构体  \* \*/  #include <iostream>  //编译匿名结构体时，要注意g++ 要加上-std=c++11 选项，以c++11编译  //匿名结构体在windows平台上不允许有默认值，但是linux平台下可以  //匿名结构体一般用在对象是唯一的或者是很少的制定数量  struct {  int age = 20;  char name[20] = "linux";  }CEO,CTO;  //结构体本身不可以再嵌套自身，但是可以包含指向自身的指针，链表等其他数据结构都是这样实现的  //测试结构体的时候有构造和析构函数  //没有构造和析构函数 | typedef struct \_Best{  // Best()  // {  // std::cout << "struct Best....\n";  // }  // ~Best()  // {  // std::cout << "struct ~Best....\n";  // }  }Best;  int main()  {  std::cout << CEO.age << std::endl;  std::cout << CEO.name << std::endl;  std::cout << CTO.name << std::endl;  std::cout << CTO.age << std::endl;  return 0;  } |

### constructor.cpp

|  |  |
| --- | --- |
| /\*  \* c++中构造函数  \* \*/  #include <iostream>  class classA  {  private:  int a;  int b;  public:  classA(int x, int y):a(x), b(y)  {  // a = x;  // b = y;  }  void print()  {  std::cout << a << " " << b << std::endl;  }  };  //delete可以禁用默认生成的函数，禁用构造函数无法实例化  //禁用拷贝构造，可以实现禁止别人拷贝你  //default 默认存在  class A | {  public:  //A()= delete;//默认删除构造函数，无法实例化  A() = default; //默认存在  //A(const A &a) = delete; //删除拷贝构造函数  };  int main()  {  return 0;  }  int main02()  {  A a;  A aa(a);  return 0;  }  int main01()  {  classA A(10, 20);  A.print();  //编译器提供的是浅拷贝。  classA B(A);  B.print();    return 0;  } |

### commonmacro\_assert.cpp

|  |  |
| --- | --- |
| /\*  \* 常见的宏和静态断言  \* \_\_FUNCTION\_\_ 不能使用  \*\*/  #include <iostream>  #include <assert.h>  #include <stdio.h>  using namespace std;  //common macro  int main01()  {  int num = 100;  cout << \_\_FILE\_\_ << endl;  cout << \_\_LINE\_\_ << endl; | cout << \_\_DATE\_\_ << endl;  cout << \_\_TIME\_\_ << endl;  //cout << \_\_FUNCTION\_\_\_ << endl;  return 0;  }  #define M  int main()  {  char num = 10;  cout << sizeof(num) << endl;  #ifdef M  //满足这个条件，静态断言为真，则编译通过。否则为假，编译不能通过  static\_assert(sizeof(num) <= 4, "linux error");  #endif  return 0;  } |

### commonfunctemplate.cpp

|  |  |
| --- | --- |
| #include <iostream>  //通用可变参数模板 处理不限定的参数，处理不同类型  //空函数，接口，最后结束递归, 新版本编译，这是个坑切记  void showall()  {  }  template<typename T, typename...Args>  void showall(const T &value, const Args &...args)  {  std::cout << value << std::endl;  showall(args...);  } | //T ＆vlaue，Args &...args ---> 可以修改原来的数据  //T value , Args ...args --->只可以修改副本的数据  //const T value , const Args ...args --->不可以修改副本，不可以修改原数据  //const T ＆value，const Args ＆...args ---> 引用数据不可以修改  int main(int argc, char \*argv[])  {  int num1 = 10, num2 = 9, num3 = 11;  double db1 = 10.8, db2 = 10.9;  char str[40] = "linux";  showall(num1);  std::cout << "\n\n\n";  showall(num1, db1, str);  return 0; |

### color.cpp

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  #include <stdlib.h>  #include <stdio.h>  int main(int argc, char \*\* argv)  {  char tmp[10] = {0};  sprintf(tmp, "%d", 0);  std::string tmpstr = tmp;  std::string default\_console = "\033["+tmpstr+"m";  for (int i = 30; i <= 37; i++)  {  sprintf(tmp, "%d", i);  std::string tmpstr = tmp;  std::string color = "\033[0;"+tmpstr+"m"; | std::cout<<color<<"test "<<i<<std::endl;  sprintf(tmp, "%d", i);  tmpstr = tmp;  color = "\033[1"+tmpstr+"m";  std::cout<<color<<"test "<<i<<" bold"<<std::endl;  std::cout<<default\_console<<std::endl;  }  std::cout << "Done" << std::endl;  return 0;  } |

### classwrapper.cpp

|  |  |
| --- | --- |
| /\*  \* 类包装器  \* \*/  #include <iostream>  template<typename T, typename F>  T run(T t, F f)  {  return f(t);  }  class Mul  {  public:  int number;  Mul(int num) : number(num)  {}  int operator()(int num)  {  return num \* number;  }  };  int add(int num) | {  return num + 10;  }  int main()  {  int num = 101;  Mul mul(11);  std::cout << run(num, mul) << std::endl;  std::cout << run(num,Mul(101)) << std::endl;  return 0;  }  int main01()  {  auto num = 100;  auto func = add;//int (\*pfunc) = add;  int (\*pfunc)(int num) = add;  std::cout << run(num, add) << std::endl;  std::cout << func(1010) << std::endl;  std::cout << pfunc(101) << std::endl;  return 0;  } |

### classtypeconvert.cpp

|  |  |
| --- | --- |
| /\*  \* 类型转换函数，将复杂类型转换为基本类型  \* 类型转换函数格式，没有返回值，方式return T，没有参数 \  \* 只能是成员函数  \* operator T() {  \* //TODO SOMETHING  \* return A;  \* }  \*  \* \*/  #include <iostream>  class A  {  public:  int a1;  int a2;  void Print() {  std::cout << "a1 = " << a1 << " a2 = " << a2 << std::endl;  }  };  class Complex  {  public:  Complex();  Complex(int num);  Complex(int real, int virtualnum);  operator int();  operator A();  private:  int real;  int virtualnum;  };  Complex::Complex() { | real = 0;  virtualnum = 0;  }  Complex::Complex(int num) {  real = num;  virtualnum = num;  }  Complex::Complex(int real, int virtualnum) {  this->real = real;  this->virtualnum = virtualnum;  }  Complex::operator int() {  return real;  }  Complex::operator A() {  A a;  a.a1 = this->real;  a.a2 = this->virtualnum;  return a;  }  int main()  {  Complex com(1, 2);  Complex com01 = 1;  int num = com;  std::cout << num << std::endl;  A a = (A)com; //等价于 A a = com;  a.Print();  return 0;  } |

### classnesting.cpp

|  |  |
| --- | --- |
| /\*  \* 类嵌套，以及类模板嵌套  \* \*/  #include <iostream>  class myclass  {  public:  class newclass  {  public:  int num = 10;  }new1; //类的嵌套 直接初始化  };  class newnewlass : public myclass  {  public:    };  template<class T>  class templateclass  {  public:  class A  {  public:  int num;  }new1; //非模板类可以直接初始化  A a;    template<class V>  class B | {  public:  V x;  }; //不能直接初始化  B<T> b; //T来初始化V  B<double> bb;  };  int main(int argc, char \*argv[])  {  templateclass<int> tmpclass;  tmpclass.new1.num = 10;  std::cout << tmpclass.new1.num << std::endl;  tmpclass.b.x = 20;  std::cout << tmpclass.b.x << std::endl;  tmpclass.bb.x = 20.1;  std::cout << tmpclass.bb.x << std::endl;  return 0;  }  int main02(int argc, char \*argv[])  {  newnewlass new01;  std::cout << new01.myclass::new1.num << std::endl;  return 0;  }  int main01(int argc, char \*argv[])  {  myclass my;  std::cout << my.new1.num << std::endl;  return 0;  } |

### classandcache.cpp

|  |  |
| --- | --- |
| /\*  \* 类中的成员于内存的关系  \* \*/  #include <iostream>  /\*  \* 类中代码都是放在代码区（包括静态成员），普通成员变量都是于具体的对象关联  \*  \* \*/  class MyClass  {  public:  static void staticfun();  void commonfun();  };  void MyClass::staticfun()  {  std::cout << "myclass staticfun" << std::endl;  }  void MyClass::commonfun()  {  std::cout << "myclass commonfun" << std::endl;  }  class MyClassB  {  public:  MyClassB(int num, int &myint):num(num), mynum(myint)  {  const int \*pint = &num;  int \*pnum = const\_cast<int \*>(pint);  //类中const属性可以去掉，但是外部就不行  \*pnum = 12;  std::cout << num << std::endl;  } | //const构造时初始化或者使用默认值  const int num;  //构造时初始化  int &mynum;  //可以不初始化  static int staticnum;  //可以不初始化  const static int staticconstnum;  //也可以不初始化---->主要包含static就可以不初始化  static const int conststaticnum;  };  int MyClassB::staticnum = 1;  //只要包含const在初始化的时候就要加上const属性，而static则不需要加上的  const int MyClassB::staticconstnum = 2;  const int MyClassB::conststaticnum = 3;    int main()  {  MyClass::staticfun();  MyClass myclass;  //利用函数指针指向类中的静态函数，于具体的对象没有关系  void (\*pfunc)() = &MyClass::staticfun;  pfunc();  //利用函数指针指向类中的普通函数  void (MyClass::\*pcommonfunc)() = &MyClass::commonfun;  //注意()优先级高于\*,要指定调用的对象。  (myclass.\*pcommonfunc)();  int num = 10;  MyClassB classb(0,num);  //静态常量区只读不能写，但是静态去可读可写  const int \*pstaticnum = &(MyClassB::conststaticnum);  int \*pnum = const\_cast<int \*>(pstaticnum);  \*pnum = 13;  std::cout << MyClassB::staticnum << std::endl;  return 0;  } |

### class\_assignment.cpp

|  |  |
| --- | --- |
| //子类与父类的之间赋值情况，A为父类，B为子类  //1.实例之间的赋值  // A a = B() ----> 没有编译错误，先构造父类，在构造子类，还会有个拷贝构造的过程。中间调用的虚函数还是子类实现的。  // B b = A() ----> 有编译错误，类型不匹配  //  //2.指针之间的赋值  // A \*a = new B() ----> 多态的基础，直接操作的是地址  // B \*b = new A() ----> 可以的操作，但是个危险的操作，子类操作的成员，一旦超过父类的地址空间，就会出现异常情况  //  //3.引用之间的赋值  //B b;  //A &a = b ----> 与实例之间的赋值类似，只是没有拷贝构造的过程。  //B &b = A() ----> 编译错误，即使做强制转换还是错误  #include <iostream>  #include <string>  using namespace std;  class Parent  {  public:  Parent()  {  mName = "Parent";  cout << "Parent ctor..." << endl;  }  ~Parent()  {  cout << "Parent dtor..." << endl;  }  Parent(const Parent &pParent)  {  mPid = pParent.mPid;  mName = pParent.mName;  cout << "Copyctor..." << endl;  }  virtual void GetName()  { | cout << mName << endl;  }  protected:  int mPid;  string mName;  };  class Child : public Parent  {  public:  Child()  {  mName = "Child";  cout << "Child ctor..." << endl;  }  ~Child()  {  cout << "Child dtor..." << endl;  }  public:  int Age = 0;  };  int main(int argc, char \*argv[])  {  // Parent p = Child();  // p.GetName();  // Child c = static\_cast<Child>(Parent()); //error 利用强制转换也会发生编译错误  // Child \*d = static\_cast<Child \*>(new Parent()); //未定义的行为，危险行为  // cout << d->Age << endl;  // Child c;  // Parent &p = c;  // p.GetName();  Parent p;  Child &c = static\_cast<Child>(p);  return 0;  } |

### cc++funcpointer.cpp

|  |  |
| --- | --- |
| /\*  \* c和c++函数指针  \* c++中的类成员函数指针数组  \* \*/  //#define CFUNCPOINTER  #ifdef CFUNCPOINTER  #include <stdio.h>  #else  #include <iostream>  #include <typeinfo>  class com  {  private:  int a;  int b;  public:  com(int x, int y) : a(x), b(y)  {}  int add(int a, int b)  {  return a + b;  }  int sub(int a, int b)  {  return a - b;  }    int mul(int a, int b)  {  return a \* b;  }  int div(int a, int b)  {  return a / b;  }  };  int addC(int a, int b)  {  return a + b;  }  void run()  {  printf("run\n");  }  #endif  int main() | {  #ifdef CFUNCPOINTER  int (\*padd)(int, int) = addC;  void (\*prun)(void) = run;  printf("%d\n", padd(1, 2));  //\*p编译器自动将\*p解释为p  printf("%d\n", (\*padd)(1, 2));  //不管前面都多少的\*，编译器都会自动解释为p  printf("%d\n", (\*\*\*\*\*\*\*\*\*padd)(1, 2));  //& 没有\*不可以执行的，超过两个地址也不可以  printf("%d\n", (&(\*\*padd))(1, 2));  //&p不能  //printf("%d\n", &padd(1, 2));  //padd和\*padd的地址是一样的  printf("%p, %p, %p", &padd, padd, \*padd);  //取地址，取就是cpu即将调用函数执行c语言内嵌ASM  //一些老版本的编译器，&padd和padd, \*padd是同样的效果  prun();  #else  #define COM1  #ifdef COM1  int (\*padd)(int, int) = addC;  //只能有一个取地址符号，\*多少无所谓和c是一样的  std::cout << (\*\*\*\*\*\*\*\*\*\*\*\*\*\*&padd)(10, 20) << "\n";  com com1(100, 20);  //auto会自动判断类型，这个函数指针中隐藏了一个参数\_thiscall表示当前class  //所以下面声明函数指针的时候，要加上com::表示函数指针的类型  auto pfun = &com::add;  int (com::\*pfun01) (int, int) = &com::sub;  //指定com实例化对象调用函数指针  std::cout << (com1.\*pfun)(10, 20) << std::endl;  std::cout << (com1.\*pfun01)(10, 20) << std::endl;  std::cout << typeid(pfun).name() << std::endl;  std::cout << typeid(pfun01).name() << std::endl;  #else  typedef int (com::\*pfun)(int, int);  com com2(100, 20);  //指向类成员函数指针数组  //pfun pfunarray[4] = {&com::add, &com::sub, &com::mul, &com::div};  int (com::\*pfunarray[])(int, int) = {&com::add, &com::sub, &com::mul, &com::div};  for (int i = 0; i < 4; i++) {  std::cout << (com2.\*pfunarray[i])(10, 20) << std::endl;  }  //指向类成员函数指针的指针  int (com::\*\*ppfunc)(int, int) = pfunarray;  for (; ppfunc < pfunarray + 4; ppfunc++) {  std::cout << (com2.\*\*ppfunc)(10, 20) << "\n";  }  #endif  #endif  return 0;  } |

### c++thread.cpp

|  |  |
| --- | --- |
| /\*  \* c++ 线程  \* \*/  #include <iostream>  #include <thread> //包含头文件  #include <vector>  using namespace std;  using namespace std::this\_thread;  void CallByThread()  {  while(1) { | //cout << "thread id = " << get\_id() << endl;  printf("hi \n");  }  }  int main()  {  auto n = thread::hardware\_concurrency();  cout << n <<endl;  thread thread01(CallByThread);  thread thread02(CallByThread);  thread01.join();  thread02.join();  return 0;  } |

### auto\_ptr.cpp

|  |  |
| --- | --- |
| /\*  \* 智能指针  \*\*/  #include <iostream>  #include <memory>  int main()  {  for(int i = 0; i < 1000; i++){ | double \*p = new double;  std::auto\_ptr<double> autop(p);  //创建智能指针管理指针p指向内存  //智能指针  }  //新型智能指针，可以直接new一个对象进去  std::auto\_ptr<double> autop(new double);  return 0;  } |

### advancearray.cpp

|  |  |
| --- | --- |
| /\*\*  \*高级数据 + lambda表达式  \*/  #include <iostream>  #include <vector>  #include <algorithm> //算法 加入头文件 lambda表达式  int main()  {  //动态无规则数组管理  std::vector<int> myvector01 ={1, 2, 3};  std::vector<int> myvector02 ={4, 5, 6};  std::vector<int> myvector03 ={7, 8, 9, 10, 11};  std::vector<std::vector<int>> myvecvec = {myvector01, myvector02, myvector03};  for(int i = 0; i < myvecvec.size(); i++) {  for(int j = 0; j < myvecvec[i].size(); j++) {  std::cout << myvecvec[i][j] << " ";  }  std::cout << "\n";  }  return 0;  }  int main02()  {  std::vector<int> myvector(5); //分配5个空间，默认为0  myvector.push\_back(1); //在尾部插入 | myvector.push\_back(2); //在尾部插入  myvector.push\_back(3); //在尾部插入  myvector.pop\_back(); //弹出一个元素  myvector.insert(myvector.begin() + 1, 999);  myvector.erase(myvector.begin() + 7);  for(int i = 0; i < myvector.size(); i++) {  std::cout << myvector.at(i) << std::endl;  }  return 0;  }  int main01()  {  std::vector<int> myvector;  myvector.push\_back(11);  myvector.push\_back(22);  myvector.push\_back(33);  int res = 0; //结果  //lambda []表示要外部传进去的参数,()表示内部参数,{}操作逻辑，代码  std::for\_each(myvector.begin(), myvector.end(), [&res](int num){res += num;});  std::cout << res << std::endl;  return 0;  } |

### abstraticclass.cpp

|  |  |
| --- | --- |
| /\*  \* 纯虚函数--->就是抽象类  \* \*/  #include <iostream>  class Base  {  public:  virtual void showname() = 0;  virtual void showaddr() = 0;  virtual void showage() = 0;  };  class GreatFather: public Base  {  public:  //这里要定义虚函数，不能声明虚函数，虽然没有具体的实现  virtual void Hi() {  std::cout << "GreatFather hi" << std::endl;  }  void showname() {  std::cout << "this is GreatFather." << std::endl;  }  void showaddr() {  std::cout << "Shanghai PuDong." << std::endl;  }  void showage() {  std::cout << "I am 23." << std::endl;  }  }; | class Father: public GreatFather//: public Base  {  public:  void showname() {  std::cout << "this is father." << std::endl;  }  void Hi() {  std::cout << "father hi" << std::endl;  }  };  int main()  {  //x64平台下指针为8个字节，在x86平台下为4个字节，不管有个多少虚函数，都是一个指针指向虚函数表  std::cout << sizeof(Base) << std::endl;  }  int main01()  {  //拥有纯虚函数的类不能实例化  //Base base;  //继承Base类，只要有一个纯虚函数没有实现，就还是抽象类  GreatFather greatfather;  Father father;  father.showname();  father.Hi();  //调用父类的方法  father.GreatFather::Hi();  //father.showname();  return 0;  } |

### aboutarraypoint.cpp

|  |  |
| --- | --- |
| /\*  \* 关于复杂对象指针数组和数组指针  \* \*/  class MyClass  {  public:  int num;  char name[10];  };  int main()  {  //指针数组  MyClass \*parr = new MyClass[5];    //二级指针 | MyClass \*\*pparr = new MyClass\*[5]; //数组的每个元素都是一个指向MyClass的指针  //二维数组,数组每个元素都是MyClass  MyClass p2arr[5][5];  //指向数组的指针  MyClass (\*pp2arr)[5] = p2arr;  //强制转换将二级指针转换为指向数组指针  pp2arr = (MyClass (\*)[5])pparr;  //二位数主，每个数组的元素都是一个MyClass指针  MyClass \*p\_p2arr[5][5];  //声明指向数组的指针  MyClass \*(\*p\_pp2arr)[5] = p\_p2arr;  MyClass \*\*\*p\_pparr = (MyClass \*\*\*)p\_p2arr;  return 0;  } |