1. #include <iostream> 标准输入输出流
   1. using namespace std; -------命名空间,使用名为std的命名空间
   2. cout << “hello ..” << endl; cout:标准输出；<<左移运算符；endl结束换行
   3. system(“pause”) 阻塞功能
   4. retrun 0 返回正常退出
2. ::双冒号作用域运算符
   1. 全局作用域 直接加：：

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include<iostream>  using namespace std;  int atk = 200;  void test01()  {  int atk = 100;  cout << "攻击力为 ： " << atk << endl;  //双冒号 作用域运算符 ::全局作用域  cout << "全局攻击力为 ： " << ::atk << endl;  }  int main(){  test01();  system("pause");  return EXIT\_SUCCESS;  } |

1. namespace 命名空间
   1. 用途 解决名称冲突问题
   2. 必须在全局作用域下声明
   3. 命名空间下可以放入 函数、变量、结构体、类…

|  |
| --- |
| 1. namespace A 2. { 3. void func(); 4. int m\_A = 20; 5. struct Person 6. { 7. }; 8. class Animal{}; 9. namespace B 10. { 11. int m\_A = 10; 12. } 13. } |

* 1. 命名空间可以嵌套命名空间
  2. 命名空间是开放的，可以随时加入新的成员

|  |
| --- |
| namespace A //此A命名空间会和上面的命名空间A进行合并  {  int m\_B = 1000;  } |

* 1. 匿名命名空间 static

|  |
| --- |
| namespace  {  int m\_C = 0;  int m\_D = 0;  }  //当写了无名命名空间，相当于写了 static int m\_C ; static int m\_D;  //只能在当前文件内使用 |

* 1. 可以起别名

|  |
| --- |
| namespace veryLongName  {  int m\_A = 0;  }  void test04()  {  //起别名  namespace veryShortName = veryLongName;  cout << veryLongName::m\_A << endl;  cout << veryShortName::m\_A << endl;  } |

1. using声明和using编译指令-------（二义性的问题的理解）
   1. using LOL:: sunwukongID;
   2. 如果局部范围内还有 sunwukongID，会出现二义性问题，要注意避免

|  |
| --- |
| namespace KingGlory  {  int sunwukongId = 10;  }  void test01()  {  int sunwukongId = 20;  //using 声明 注意避免二义性问题  //写了using声明后 下面这行代码说明以后看到的sunwukongId 是用KingGlory下的  //但是 编译器又有就近原则  //二义性  //using KingGlory::sunwukongId;  cout << sunwukongId << endl;  } |

* 1. 编译指令
  2. using namespace LOL
  3. 如果局部范围内还有 sunwukongID ,使用局部的ID
  4. 如果打开多个房间，那么也要注意二义性问题

|  |
| --- |
| namespace LOL  {  int sunwukongId = 30;  }  void test02()  {  //int sunwukongId = 20;  //using编译指令  using namespace KingGlory; //打开王者荣耀房间  using namespace LOL;//打开LOL房间  //如果打开多个房间，也要避免二义性问题  cout << LOL::sunwukongId << endl;  } |

1. C++对C语言增强
   1. 全局变量检测增强（全局变量必须赋予初始值）
   2. 函数检测增强
      1. 参数类型检测（参数必须带有类型）
      2. 返回值检测
      3. 传参个数检测（C++中传入的参数必须和方法中指定的参数个数、类型和顺序相同）
   3. 类型转换检测增强
      1. malloc返回void\* ，C中可以不用强转，C++必须强转

|  |
| --- |
| void test03()  {  char \* p = (char\*)malloc(sizeof(64)); //malloc返回值是void\*，必须强制转为char\*  } |

* 1. struct增强
     1. C中不许有函数 C++可以
     2. 使用C必须加关键字 struct ，C++可以不加

|  |
| --- |
| struct Person  {  int m\_Age;  void plusAge(){ m\_Age++; }; //c++中struct可以加函数  };  void test04()  {  Person p1; //使用时候可以不加struct关键字  p1.m\_Age = 10;  p1.plusAge();  cout << p1.m\_Age << endl;  } |

* 1. bool数据类型增强
     1. C没有 C++有（c99标准中有）
     2. true 真 false假
     3. sizeof 1
  2. 三目运算符增强
     1. C中返回的是值
     2. C++中返回的是变量

|  |
| --- |
| void test06()  {  int a = 10;  int b = 20；  cout << "ret = " << (a < b ? a : b) << endl;  (a < b ? a : b) = 100; //b = 100 C++中返回的是变量  cout << "a = " << a << endl;  cout << "b = " << b << endl;  } |

* 1. const增强

c语言、c++\_中，全局的const是常量。都不可以通过指针更改

c语言中的const不可以用来初始化数组，c++中可以

* + 1. C语言中局部const是伪常量，可以通过指针修改
    2. C++中const会放入到符号表中 ，不可以通过指针修改
    3. C语言中const默认是外部链接，C++中const默认是内部链接，可以使用extrn增强其作用域

个人理解：简单来讲外部链接就是提供给其他文件使用的东西，例如全局变量，内部链接就是只给本文件使用的东西，例如c语言中的static变量

|  |
| --- |
| test.cpp  extern const int a = 10; //C++中的const默认内部链接 ,extern提高作用域 |
| 主函数：  int main(){  extern const int a;  cout << a << endl;  system("pause");  return EXIT\_SUCCESS;  } |

* + 1. const分配内存情况
       1. 对变量取地址，会分配临时内存

|  |
| --- |
| //1、const分配内存 取地址会分配临时内存  //2、extern 编译器也会给const变量分配内存  void test01()  {  const int m\_A = 10;  int \* p = (int\*)&m\_A; //会分配临时内存  } |

* + - 1. extern关键字下的const会分配内存
      2. 用普通变量初始化**const**变量

|  |
| --- |
| void test02()  {  int a = 10;  const int b = a; //会分配内存  int \* p = (int \*) &b;  \*p = 1000;  cout << "b = " << b << endl;  } |

* + - 1. 自定义数据类型会分配内存

|  |
| --- |
| struct Person  {  string m\_Name; //姓名  int m\_Age;  };  void test03()  {  const Person p1;  //p1.m\_Name = "aaa";  Person \* p = (Person\*)&p1;  p->m\_Name = "德玛西亚";  (\*p).m\_Age = 18;  cout << "姓名： " << p1.m\_Name << " 年龄： " << p1.m\_Age << endl;  } |

* + 1. 尽量用const代替define
       1. define宏没有作用域概念
       2. define宏常量没有类型

1. 引用基本语法
   * 1. 用途起别名
     2. Type &别名 = 原名

|  |
| --- |
| void test01()  {  int a = 10;  int &b = a;  b = 20;  cout << "a = " << a << endl;  cout << "b = " << b << endl;  } |

* + 1. 引用必须初始化

|  |
| --- |
| void test02()  {  //int &a; 必须初始化  int a = 10;  int &b = a; //引用初始化后不可以修改了  int c = 20;  b = c; //赋值！！！  } |

* + 1. 一旦初始化后 不能修改
    2. 对数组建立引用

|  |
| --- |
| void test03()  {  int arr[10];  for (int i = 0; i < 10;i++)  {  arr[i] = i;  }  //给数组起别名  int(&pArr)[10] = arr;  for (int i = 0; i < 10;i++)  {  cout << pArr[i] << " ";  }  cout << endl;  //第二种方式 起别名  typedef int(ARRAYREF)[10]; //一个具有10个元素的int类型的数组  ARRAYREF & pArr2 = arr;  for (int i = 0; i < 10; i++)  {  cout << pArr2[i] << " ";  }  cout << endl;  } |

* 1. 参数3种传递方式
     1. 值传递

|  |
| --- |
| void mySwap(int a, int b)  {  int tmp = a;  a = b;  b = tmp;  cout << "mySwap::a = " << a << endl;  cout << "mySwap::b = " << b << endl;  }  void test01()  {  int a = 10;  int b = 20;  mySwap(a, b); //值传递  cout << "a = " << a << endl;  cout << "b = " << b << endl;  } |

* + 1. 地址传递

|  |
| --- |
| void mySwap2(int \* a, int \* b)  {  int tmp = \*a;  \*a = \*b;  \*b = tmp;  }  void test02()  {  int a = 10;  int b = 20;  mySwap2(&a, &b);  cout << "a = " << a << endl;  cout << "b = " << b << endl;  } |

* + 1. 引用传递

|  |
| --- |
| //引用传递 类似传地址  void mySwap3(int &a, int &b) //&a = a &b = b  {  int tmp = a;  a = b;  b = tmp;  }  void test03()  {  int a = 10;  int b = 20;  mySwap3(a, b);  cout << "a = " << a << endl;  cout << "b = " << b << endl;  } |

* 1. 注意事项，  
     不要返回局部变量的引用 引用必须引一块合法的内存空间

|  |
| --- |
| int& doWork()  {  int a = 10;  return a;  }  void test04()  {  //int &a = 10; // 引用必须引一块合法的内存空间  int &ret = doWork();  cout << "ret = " << ret << endl; //第一次10是编译器做了优化  cout << "ret = " << ret << endl;  cout << "ret = " << ret << endl;  cout << "ret = " << ret << endl;  cout << "ret = " << ret << endl;  } |

* 1. 如果函数返回值是引用，那么函数的调用可以作为左值

|  |
| --- |
| int& doWork2()  {  static int a = 10;//写成静态的成员函数不会出现上面的错误  return a;  }  void test05()  {  int &ret = doWork2();    //如果函数的返回值是引用，那么这个函数调用可以作为左值  doWork2() = 1000; //相当于写了 a = 1000;  } |

* 1. 引用的本质 就是一个指针常量

1. 指针的引用-----学习理解
   1. 用一级指针引用 可以代替二级指针
2. 常量引用-----学习理解
   1. 使用场景 修饰形参为只读
   2. const int &a = 10;会分配内存