## Android特训营NDK进阶之C++大纲

|  |  |
| --- | --- |
| 当前版本： | V1.0 |
| 制作单位： | 菜鸟窝 |
| 编写人员： | Jesson老师 |
| 审核人： | Jesson老师 |
| 签署日期： | 2018.11.21 |

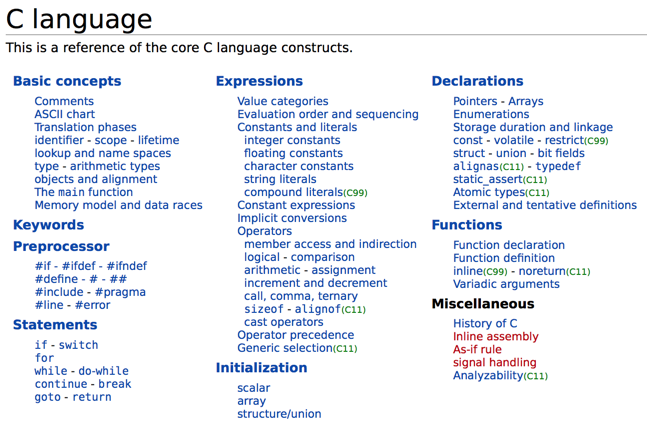
修订记录

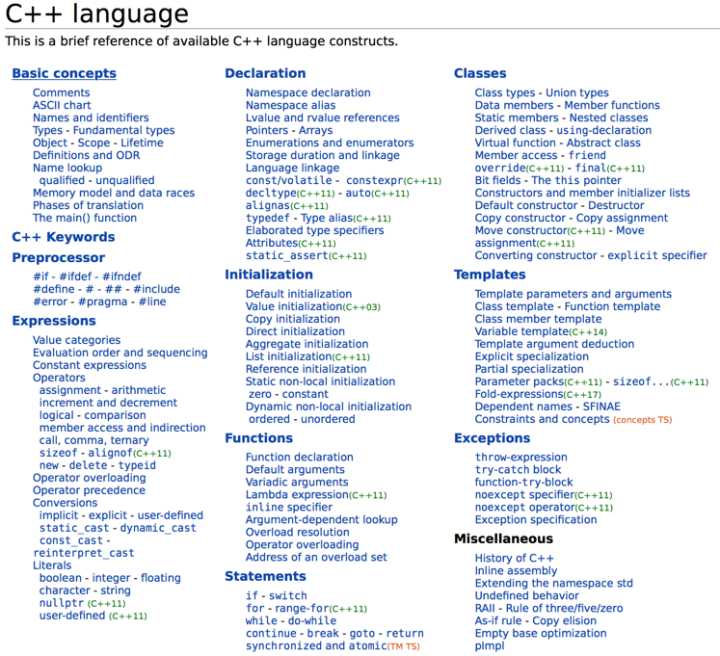
|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 描述 | 作者 | 版本号 |
| 2018.11.21 | 初稿 | Jesson老师 | V1.0 |

版权所有 翻印必究

### 说在前面的话

如何学习C++？C和C++到底有什么区别？https://en.cppreference.com/w/c/language https://en.cppreference.com/w/cpp/language





C++是C的超集，也可以说C是C++的子集，因为C先出现。按常理说，C++编译器能够编译任何C程序，但是C和C++还是有一些小差别。例如C++增加了C不具有的关键字。这些关键字能作为函数和变量的标识符在C程序中使用，尽管C++包含了所有的C，但显然没有任何C++编译器能编译这样的C程序。

如何学习？

* 面向过程思维转变成面向对象的思维
* 一定要注重思考问题、思考概念、思考设计哲学
* 多练习、多敲代码

### **C++对C语言的扩展知识**

先看一个C++经典导学代码，求圆的面积和周长：

|  |
| --- |
| #include <iostream>   **using namespace** std; //包名 **class** Circle {  **double** radius ; //成员变量 **public** : //类的访问控制  **void** Set\_Radius( **double** r ) { radius = r ; } //成员函数  **double** Get\_Radius() { **return** radius ; } //通过成员函数设置成员变量  **double** Get\_Girth() { **return** 2 \* 3.14f \* radius ; } //通过成员函数获取成员变量  **double** Get\_Area() { **return** 3.14f \* radius \* radius ; } } ;  **int** main() {  Circle A, B ; //用类定义对象  A.Set\_Radius( 6.23 ) ; //类的调用  cout << "A.Radius = " << A.Get\_Radius() << endl ;  cout << "A.Girth = " << A.Get\_Girth() << endl ;  cout << "A.Area = " << A.Get\_Area() << endl ;  B.Set\_Radius( 10.5 ) ;  cout << "B.radius = " << B.Get\_Radius() << endl ;  cout << "B.Girth=" << B.Get\_Girth() << endl ;  cout << "B.Area = " << B.Get\_Area() << endl ;  **return** 0; } |

思考题：C是面向过程的，执行语句是一步一步执行的，也可以是函数之间的调用，那么上面的代码怎么执行？（面向对象的思维解决问题）

成员函数是否可以不要？为什么c++中设计了成员函数？

扩展知识总结：

1. 命名空间总结

A)当使用<iostream>的时候，该[头文件](http://baike.baidu.com/view/668911.htm" \t "_blank)没有定义全局命名空间，必须使用namespace std；这样才能正确使用cout。若不引入using namespace std ,需要写std::cout。

B)c++标准为了和C区别开，也为了正确使用命名空间，规定[头文件](http://baike.baidu.com/view/668911.htm" \t "_blank)不使用后缀.h。

C)C++命名空间的定义：namespace name { … }

D)using namespace NameSpaceA;

E)namespce定义可嵌套。

代码实战1：熟悉namespace用法

|  |
| --- |
| #include "namespace\_demo.h" #include <iostream> **using namespace** NameSpaceA; **int** main(){  printf("a = %d\n", name);  printf("a = %d\n", NameSpaceB::age);     NameSpaceB::NameSpaceC::Teacher teacher;  teacher = {"aaa", 3};    printf("t1.name = %s\n", teacher.name);  printf("t1.age = %d\n", teacher.age);   **return** 0; } |

1. 常量 const int number = 100; 在c里面可以通过指针进行修改，但是在C++里面不行，这个和编译器有关系。
2. 引用 int age = 18；int& age2 = age；

#### 引用概念：

* 在C++中新增加了引用的概念。引用可以看作一个已定义变量的别名，引用的语法：Type& name = var; 引用作为函数参数声明时不进行初始化。
* 普通引用在声明时必须用其它的变量进行初始化，引用作为函数参数声明时不进行初始化。
* C++编译器在编译过程中使用常指针作为引用的内部实现，因此引用所占用的空间大小与指针相同。

指针和引用的区别？理解引用的真实含义？

代码分析：main.cpp中的testReference();testReferenceB();testPointerReference();

### **零碎的知识点**

可变参数？看代码可变参数.cpp

Static关键字？C++怎么操作？静态属性在C++中一定要初始化，必须写int Student::age = 23；静态方法不能操作非静态成员。静态可以直接通过类名来操作。非静态可以操作静态。

对象的大小？1.对象的大小与结构体计算方式一致 2.static 静态变量和方法没有计算到类的大小里面3.C++类对象计算需要考虑成员变量大小，内存对齐、是否有虚函数、是否有虚继承等。4.类对象大小没有因为增加了静态成员而变化。因为静态成员是属于类成员共有的，不单独属于任何一个对象，对静态成员的存储不会选择在某个对象空间，而是存在于堆当中，因此不会对对象的大小造成影响。

Const:修饰类的函数 void test() const{} 主要是来修饰this指针不能被修改

Typedef：类型替换 typedef unsigned int u\_int; tt = 266;

迁移c的代码到c++：

|  |
| --- |
| #ifdef \_\_cplusplus  extern "C"{  #endif  void func(int x,int y);  #ifdef \_\_cplusplus  }  #endif  //\_\_cplusplus 是由c++编译器定义的宏，用于表示当前处于c++环境 |

静态函数和静态成员怎么使用？ 代码静态.cpp

### **C++类和对象**

学习c++一定要有面向对象的思维。我们从类的函数开始来梳理一个类是如何被定义？如何被初始化的？

类的构造函数：类的构造函数是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。

构造函数一般是被c++编译器自动调用，也可以被手动调用。

类的析构函数：C++中的类可以定义一个特殊的成员函数清理对象，这个特殊的成员函数叫做析构函数。析构函数没有参数也没有任何返回 类型的声明。析构函数在对象销毁时自动被调用。注意这个调用的时机。语法：～classname

什么时候析构函数被调用呢？

1，栈对象离开其作用域。 2，堆对象被手动 delete.

二个特殊的构造函数

1）默认无参构造函数

当类中没有定义构造函数时，编译器默认提供一个无参构造函数，并且其函数体为空

1. 默认拷贝构造函数(对象)
2. 当类中没有定义拷贝构造函数时，编译器默认提供一个默认拷贝构造函数，简单的进行成员变量的值复制。

拷贝构造函数：

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

1）通过使用另一个同类型的对象来初始化新创建的对象。Teacher t = t2;

2）复制对象把它作为参数传递给函数。

3）复制对象，并从函数返回这个对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。拷贝构造函数的最常见形式如下：

classname (const classname &obj) { // 构造函数的主体 }

在这里，obj 是一个对象引用，该对象是用于初始化另一个对象的。

代码实战：类的初始化以及多种函数的使用。

|  |
| --- |
| #include <iostream> #include <string.h>  **using namespace** std;  **class** Student { **public**:  Student(**int** age,**char** \*name)  {  printf("构造函数执行\n");  //this->name = name;  **this**->name = (**char**\*)(malloc(**sizeof**(**char**) \* 100));  strcpy(**this**->name,name);  **this**->age = age;  }   /\*\*注意这里的写法 有点像kotlin 赋值参数\*\*/  Student(**char** \*name) : name(name) {  **this**->name = name;  }   Student() {  printf("空参构造函数执行\n");  }   ~Student() //如果在对象内部堆内存上开辟了内存，一定要记得回收  {  printf("析构函数执行\n");  free(**this**->name); //free 对应malloc   }  Student(**const** Student& student)  {  printf("对象的拷贝构造函数执行\n");  **this**->name = student.name;  **this**->age = student.age-1; //方便测试  }  **public**:  **int** getAge()  {  **return this**->age;  }   **char**\* getName()  {  **return this**->name;  }   **void** setAge(**int** age) {  Student::age = age;  }   **void** setName(**char** \*name) {  Student::name = name;  }   **private**:  **int** age;  **char**\* name; };   //todo //int main(){ // //Student student; //第一种 // //Student student1(33,"jesson"); //第二种 // Student\* student = new Student(33,"jesson"); //第三种 // //Student\* student = (Student\*)malloc(sizeof(Student));//第四种 // //student->setAge(100); // //student->setName("jesson"); // delete(student); //注意delete 是回收new对象的 // cout<<student->getAge()<<endl; // cout<<student->getName()<<endl; //}   //-------拷贝构造函数-------- **int** main(){  Student student1(33,"jesson");  Student a = student1;  cout<<a.getAge()<<endl; }    **struct** Persron{  **int** age;  **char**\* name; };  //int main(){ // Persron persron = {24,"jesson"}; // Persron persron2 = persron; //这句话的意思是赋值，编译器的工作 // cout<<&persron<<"==="<<&persron2<<endl; //} |

友元函数：类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

|  |
| --- |
| class Box  {  double width;  public:  double length;  friend void printWidth( Box box );  void setWidth( double wid );  };  #include <iostream>  using namespace std;  class Box  {  double width;  public:  friend void printWidth( Box box );  void setWidth( double wid );  }；  // 成员函数定义  void Box::setWidth( double wid )  {  width = wid;  }  // 请注意：printWidth() 不是任何类的成员函数  void printWidth( Box box )  {  /\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/  cout << "Width of box : " << box.width <<endl;  }    // 程序的主函数  int main( )  {  Box box;  // 使用成员函数设置宽度  box.setWidth(10.0)；  // 使用友元函数输出宽度  printWidth( box );  return 0;  } |

操作符重载：C++ 运算符重载，函数重载（Function Overloading）可以让一个函数名有多种功能，在不同情况下进行不同的操作；运算符重载（Operator Overloading）也是一个道理，同一个运算符可以有不同的功能；实际上，我们已经在不知不觉中使用了运算符重载；例如，+号可以对不同类型（int、float 等）的数据进行加法操作；<<既是位移运算符，又可以配合 cout 向控制台输出数据；C++ 本身已经对这些运算符进行了重载。

|  |
| --- |
| #include "运算符重载.h"  #include <cstdio>  using namespace std;  void 运算符重载::print() const {  printf("%g + %g\n", m\_real, m\_imag);  }  运算符重载 运算符重载::operator+(const 运算符重载 &A) const {  return 运算符重载(m\_real + A.m\_real, m\_imag + A.m\_imag);  }  int main() {  运算符重载 A(1.24, 5.7), B(4, 3.14), C;  C = A + B;  C.print();  return 0;  } |

### **C++继承和派生**

派生类的目的是为了发展，派生类继承了基类的成员，实现了原有代码的重用，这只是一部分，而代码的扩充才是最主要的，只有通过添加新的成员，加入新的功能，类的派生才有实际意义。

以下是两种典型的使用继承的场景：  
1) 当你创建的新类与现有的类相似，只是多出若干成员变量或成员函数时，可以使用继承，这样不但会减少代码量，而且新类会拥有基类的所有功能；

1. 当你需要创建多个类，它们拥有很多相似的成员变量或成员函数时，也可以使用继承；可以将这些类的共同成员提取出来，定义为基类，然后从基类继承，既可以节省代码，也方便后续修改成员；

因为时间问题，我这里不做详细笔记了，大家参考我的代码 整理下。

### **C++多态与虚函数**

C++多态与虚函数，多态（Polymorphism）是面向对象（Object-Oriented，OO）思想”三大特征”之一，其余两个分别是封装（Encapsulation）和继承（Inheritance）–可见多态的重要性；或者说，不懂得什么是多态就不能说懂得面向对象。  
多态是一种机制、一种能力，而非某个关键字；它在类的继承中得以实现，在类的方法调用中得以体现；

### **C++模版**

C++是一门强类型语言，所以无法做到像动态语言（python javascript）那样子，编写一段通用的逻辑，可以把任意类型的变量传进去处理。泛型编程弥补了这个缺点，通过把通用逻辑设计为模板，摆脱了类型的限制，提供了继承机制以外的另一种抽象机制，极大地提升了代码的可重用性。

模板定义本身不参与编译，而是编译器根据模板的用户使用模板时提供的类型参数生成代码，再进行编译，这一过程被称为模板实例化。用户提供不同的类型参数，就会实例化出不同的代码。

普通函数模板：

|  |
| --- |
| template<typename T>  int compare(const T& left, const T& right) {  if (left < right) {  return -1;  }  if (right < left) {  return 1;  }  return 0;  }  compare<int>(1, 2); //使用模板函数 |

成员函数模板：

|  |
| --- |
| class Printer {  public:  template<typename T>  void print(const T& t) {  cout << t <<endl;  }  };  Printer p;  p.print<const char\*>("abc"); //打印abc |

模版类：

template<class T，class B> 定义一个模板类

为类定义一种模式。使得类中的某些数据成员、默写成员函数的參数、某些成员函数的返回值，能够取随意类型

常见的 容器比如 向量 vector <int> 或 vector <string> 就是模板类。

详见代码 Seqlist

### **C++ STL**

C++ STL（标准模板库）是一套功能强大的 C++ 模板类，提供了通用的模板类和函数，这些模板类和函数可以实现多种流行和常用的算法和数据结构，如向量、链表、队列、栈。

|  |  |
| --- | --- |
| **组件** | **描述** |
| 容器（Containers） | 容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。 |
| 算法（Algorithms） | 算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。 |
| 迭代器（iterators） | 迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。 |

在数据存储上，有一种对象类型，它可以持有其它对象或指向其它对像的指针，这种对象类型就叫做容器。很简单，容器就是保存其它对象的对象，当然这是一个朴素的理解，这种“对象”还包含了一系列处理“其它对象”的方法。

容器的分类：

1、顺序容器：是一种各元素之间有顺序关系的线性表，是一种线性结构的可序群集。顺序性容器中的每个元素均有固定的位置，除非用删除或插入的操作改变这个位置。顺序容器的元素排列次序与元素值无关，而是由元素添加到容器里的次序决定。顺序容器包括：vector(向量)、list（列表）、deque（队列）。

  2、关联容器：关联式容器是非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。但是关联式容器提供了另一种根据元素特点排序的功能，这样迭代器就能根据元素的特点“顺序地”获取元素。元素是有序的集合，默认在插入的时候按升序排列。关联容器包括：map（集合）、set（映射）、multimap（多重集合）、multiset（多重映射）。

  3、容器适配器：本质上，适配器是使一种不同的行为类似于另一事物的行为的一种机制。容器适配器让一种已存在的容器类型采用另一种不同的抽象类型的工作方式实现。适配器是容器的接口，它本身不能直接保存元素，它保存元素的机制是调用另一种顺序容器去实现，即可以把适配器看作“它保存一个容器，这个容器再保存所有元素”。STL 中包含三种适配器：栈stack 、队列queue 和优先级队列priority\_queue 。

容器类自动申请和释放内存，因此无需new和delete操作。