电子信息学院

C++编程基础 & 数据集合类型

C++ PROGRAMMING FUNDAMENTAL & DATA COLLECTIONS

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwwang@aliyun.com
课程QQ群：珞珈EIS数据结构与算法，668792335

电子信息学院 Table of Contents

本章位置

熟练运用高级编程语言对于算法与数据结构的学习与实践非常重要，而算法与数据结构的系统理论方法则是专业程序设计的基础。本章介绍对完成本课程的编程实践而言非常重要的C++语言的基本内容，重点讨论C++的面向对象机制和现代C++的新特性。

电子信息学院 Table of Contents

---

0. C++简介

一、现代C++概述

- C++是由 Bjarne Stroustrup于1979年开始设计开发的，1983年更名为 C++。
- C++语言是一种从C语言基础上发展起来的面向对象编程语言，在保持与C语言向后兼容的条件下，支持面向对象编程OOP技术。
- C++程序具有灵活、快速、高效的特点：
  - C++支持访问低级别硬件功能，从而最大限度地提高速度并最大程度地降低内存需求；
  - C++可以在高的抽象级别上描述待求解的问题。C++提供高度优化的标准库，极大地方便了众多领域的应用开发。
  - C++自创建以来逐渐成为世界上最常用的编程语言之一，众多嵌入式程序、Windows 客户端应用，甚至用于其他编程语言的库和编译器也使用 C++ 编写。

---

1. 面向对象编程与数据结构

概述

- 面向过程的程序设计：数据的描述和对数据的操作是分离的，数据的描述用数据类型表示，对数据的操作则用函数表示。例如，在描述栈（stack）时，先定义栈的数据表示，再用一系列分离的函数实现对栈的各种操作。
- 这种方式重用性差、可移植性差、数据维护困难。
- 针对这些问题逐渐发展出了OOP思想，具有抽象、信息隐藏和封装、继承和多态等特性，C++、C#、Java以及Python等主流编程语言均支持面向对象技术。

面向对象编程与数据结构（II）

概述

- 数据结构的三个要素（逻辑结构、存储结构以及对数据所进行的操作）相互依存、互为一体，OOP能够更深入地刻画数据结构。
- 例：string类描述字符串对象，而串连接、串比较等操作则设计为该类的方法。关于数据的描述和对数据的操作都封装在以类为单位的模块中，增强了代码的重用性、可移植性，使数据和算法易于维护。string类的客户端只需要知道该类对外的接口即可方便地构造和使用字符串这种数据结构。
- 得力于面向对象技术，C++标准库实现了多种复杂的数据结构与算法，广大应用编程人员可以方便地用于广泛的编程实践中。

面向对象编程与数据结构（III）

概述

- C++支持一维数组和多维数组，数组提供了基本的数据顺序存储机制。
- C++的指针提供了按存储地址操作数据对象的方式，直接对数据的链式存储结构提供了支持。现代C++还提供智能指针，避免直接使用指针所带来的安全隐患。
- C++标准库中包含多种容器类型，如vector、list、set、map等，可以在更高的抽象级别上描述和操作数据。标准库中的algorithm模块定义了很多的常用算法，为各种容器类型的操作（如复制、计数、排序和查找等常用的功能）提供了统一、高效和实用的方法。
- 总之，使用现代C++语言可以让软件设计人员以面向对象的方式实现和应用各种复杂的算法与数据结构。

2. 现代C++的新特性概述

概述

- C++标准在不断演变，现代C++一般指C++11及以后的标准（C++14/17/20）。
- 现代C++引入大量实用的特性，主要体现：
  - 增强或者改善的语法特性
  - 改善的或者新增的标准库
- 现代C++的程序代码更加简单、安全，而且速度仍像以往一样快速。

1）用auto替代显式类型名称

```
vector<string>::iterator k = v.begin();
// old-style，嵌套类型
auto i = v.begin();      // modern C++；
auto pred = [](int i){return i%2==0;};
// 函数对象，Lambda表达式表示的匿名函数
```


2）基于范围的for循环

- C++11引入基于范围（range based）的for 循环，用于方便地遍历集合。例如：

```
vector<int> v {1,2,3};
for(auto& item : v)   // for(const auto item : v)
    cout << item;
```


3） 统一的初始化方式

- 现代 C++可统一地使用大括弧的形式对数组、复合类型及容器类型实例进行初始化。例如：

```
vector<int> v {1,2,3};
list<Student> sl { {"518001", "王　　兵", "男",18,92},
     {106002, "张芳", "女",17,95}};
```


4）string和string_view

- C++标准库中的字符串类string或wstring可以消除与C样式字符串关联的所有错误，而且C++串类对速度进行了高度优化。
- 在仅需要以只读权限访问字符串时，可以使用string_view来提高性能（C++17以后）。

5）智能指针

- C++标准库提供了三种智能指针类型：unique_ptr、shared_ptr和weak_ptr。
- 智能指针可处理对其拥有的内存的分配和删除。
- 下例通过调用make_unique()在系统堆上为数组data分配100个int单元的内存空间，当变量data超出范围时，其析构函数被自动调用，它释放为data分配的内存。

```
unique_ptr<int[]> data = make_unique<int[]>(100);
```


6）资源管理的RAII原则

- 资源管理是C++程序的重要任务，设计不完善则易造成资源泄露，成为编程中一种常见的错误。
- 现代C++强调应用RAII（Resource Acquisition Is Initialization，资源获取即初始化）原则，其理念是，资源由对象“拥有”。对象在初始构造时向系统申请分配资源，而在销毁析构过程中释放资源。
- 智能指针类型，其实即是对RAII原则的应用示范。

7）移动语义

- C++程序包含许多数值赋值和复制。对于拥有堆内存、文件句柄等资源的对象，这种复制可能会伴随大量的在内存块间复制数据的负荷，这是传统C++中的负担和不足。
- 现代C++引入移动语义，以避免进行不必要的内存复制。移动操作会将资源的所有权从原对象转移到新对象，而不是复制。
- 在实现拥有资源的类时，可以为之定义移动构造函数和重载移动赋值运算符。

8）标准库容器和算法

- C++标准库包含的容器类型都遵循RAII原则，并且对性能进行了高度优化和充分的测试。
- 标准库包含许多常见操作（如查找、排序、筛选和随机化）的算法，容器类型为安全遍历元素提供迭代器，支持与算法模块广泛地协调应用。

9）函数对象与Lambda表达式

- 高阶函数是含有以另一函数为参数的函数，在C样式编程中，使用函数指针来传递这样的参数。函数指针不便于维护和理解，且不是类型安全的。
- 现代C++提供函数对象（重写了( )运算符的类，又称为可调用类型），其实例可以像函数一样被调用。函数对象的实例所引用的“值”，是与其类型兼容的某个函数，既可以是在源代码的其他位置定义的已命名函数，也可以是在调用的位置定义的匿名函数。lambda表达式是表示和创建匿名函数并赋值给函数对象实例的最简便方法。

二、C++语言编程基础

- 生成C++程序的过程：
  1）C++源代码文件（cpp文件和h文件）编辑
  2）编译源代码为目标代码文件（obj文件）
  3）将编译后的目标代码添加到静态库文件(.lib文件)或
  与静态库链接到可执行文件 (.exe文件)或动态加载库文件 (.dll文件)。
- 为进行C++程序设计，可以使用多种方法和工具。编辑、编译、链接等几个步骤可以分别用单独的专门实用程序完成。

1. 命令行编译和运行: 编写第一个C++程序

基础

- 打开“开发人员命令提示”，进入源代码所在的文件夹，然后输入命令： cl Hello.cpp
  如果程序没有包含语法错误，则将创建一个 Hello.exe 文件。
- 要运行程序，请输入命令：Hello
- 命令行编译示例
  - 编译 File.cpp 以产生目标文件 File.obj：cl /c File.cpp
  - 编译当前目录中所有的 C++ 文件，以产生可执行文件：cl /Fe:App.exe *.cpp
  - 编译 APP.cpp 并与库dsa.lib链接以产生可执行文件：cl App.cpp dsa,lib
  - 由目标文件构建库文件dsa.lib：lib /out:dsa.lib *.obj

```
// A "Hello World!" program in C++
#include <iostream>
using namespace std;
int main(){
 vector<string> v{"Hello","C++","World","!"};
for(string& item: v)
   cout << item << endl; }
```

要点：1）代码注释；
2）程序入口main 函数；
3）include头文件
4）输出流cout；
5）命名空间std。

应用程序类型

基础

应用程序类型
- 控制台应用程序
- Window应用程序
- Web应用程序

程序库
- 静态库文件.lib
- 动态链接库.dll

控制台应用程序

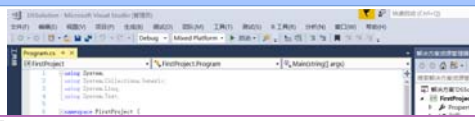
```
int main(){
  cout << "Hello World!" << endl;
  return 0;
}
```

“Hello World” C++程序

Windows应用程序

概述

```
using System.Windows.Forms;
class HelloWorldForm: Form{
    public HelloWorldForm(){
        initializeComponents();
        this.label1.Text="Hello world";
    }
}
```

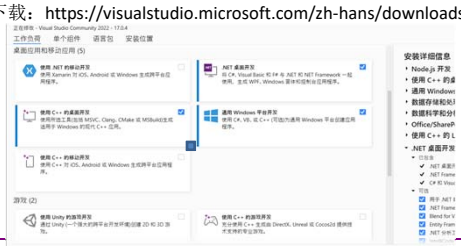


Visual Studio 2015-2022 下载与安装

基础

- Visual Studio是一整套工具组件（IDE）。它对开发功能强大、性能可靠的企业网络解决方案进行了简化。
- 利用Visual Studio，用户可以轻松地创建具有自动伸缩能力的应用程序和组件。
- 下载：https://visualstudio.microsoft.com/zh-hans/downloads/

安装



2. C++的数据类型与流程控制

基础

1）C++的基本数据类型

2）C++的操作符与表达式

3）C++的流程控制

1）C++的基本数据类型

基础

a) 值类型模型
b) 指针类型
c) 类型转换

value type：变量分配在栈中，所具空间包含其数据。拷贝赋值，对一个变量的操作不会影响其他变量。

int a=5; int b = a; a=9;

pointer type：指针变量包含地址，指向实际数据空间。通过指针变量和new操作符来声明创建和操作对象。

```
int number = 5;
string* pstr = new string("hello");
```

(a) 值类型变量　　(b) 指针类型变量

内置基本类型

基础

包括整型、浮点型、布尔型、字符型，这些类型与其他语言的基本数据类型相似。

简单类型	描 述	示 例
char	8位ASCII编码字符	char val = 'h';
unsigned char	8位无符号整数	unsigned char val1 = 12;
short	16位有符号整数	short val = 12;
unsigned short	16位无符号整数	unsigned short val1 = 12;
int	32位有符号整数	int val = 12;
unsigned int	32位无符号整数	unsigned int val1 = 12;
float	32位单精度浮点数	float val = 1.23F;
double	64位双精度浮点数	double val1 = 1.23; double val2 = 4.56D;
bool	布尔类型	bool val1 = true; bool val2 = false;

类型转换

基础

隐式类型转换是系统默认的、不需要任何声明就可以进行的转换，它是由编译器根据不同类型数据间转换规则自动完成的，又称为自动转换。例如：

```
short a=100;
float c=a;    //“小类型”到“大类型”隐式转换
```

显式类型转换就是强制执行从一种数据类型到另一种数据类型的转换，也称为强制类型转换。例如：

```
int  i,h;
double  x=4.5;
i=(int)x;  //不同类型显式转换
h = x;  // x “大类型”到“小类型”不能隐式转换
```


2）C++的操作符与表达式

基础

- 操作符主要分为4类：
  - 算术操作符
  - 位操作符
  - 关系操作符
  - 逻辑操作符。
- 下表按优先级从高到低的次序列出C++定义的所有操作符，分隔符的优先级最高，表中“左→右”表示从左向右的操作次序（结合性）。

操作符的优先级表

基础

优先级	运 算 符	结合性
0	::（作用域解析）	无
1	. 或 -> [] () ++（后缀递增） --（后缀递减）	左⇨右
2	++ -- ~ ! + －（一元） & new delete sizeof	右⇨左
3	* / %	左⇨右
4	+ －（二元）	左⇨右
5	<< >>	左⇨右
6	< > <= >=	左⇨右
7	== !=	左⇨右
8	&	左⇨右
9	^	左⇨右
10	|	左⇨右
11	&&	左⇨右
12	||	左⇨右
13	? :	右⇨左
14	= *= /= %= += －= <<= >>= &= ^= |=	右⇨左

操作符(函数)重载

- C++的某些操作符的功能内在是通过调用某个相应的操作符函数来实现的，操作符，即操作符函数，可以被重载。操作符重载使得自定义类型可以用简单的操作符来方便地表达某些常用的操作。例如，常见的输出操作：cout << a;

  本质上是通过调用函数operator<<(cout,a)来完成的。

表达式

基础

- 为完成一个计算结果的一系列操作符和操作数的组合称为表达式。
- 表达式可以分为赋值表达式和逻辑表达式两种。
- New操作符：new操作符用于创建某类型的一个新实例。它有两种形式：
  - 对象创建表达式
  - 数组创建表达式
- 例：

```
class A{};
Student* ps = new Student();  A*  p = new A( );
int*  y  = new int[10];
```

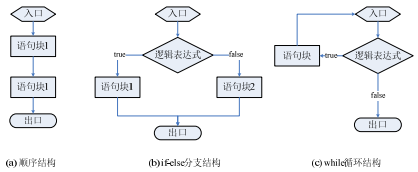
- delete操作符：

```
delete ps;
delete [] y;
```

C++的流程控制

基础

- 按程序的执行流程，程序的控制结构可分为3种：顺序结构，分支结构和循环结构。



(a) 顺序结构　(b) if-else分支结构　(c) while循环结构

```
for (string& s: args){
    <语句>;
} // 基于范围的for循环
```


3. C++的标准输入/输出流

- C++标准I/O库用流（stream）抽象数据载体及其输入输出，流是字节序列，而I/O发生在流上。输入指数据字节流从设备流向内存，输出指数据流从内存流向设备。这些设备为iostream 类的实例。
- 头文件<iostream>定义了cin、cout、cerr和clog对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流，使用它们可从控制台读取或向控制台输出数据信息。

1）标准输出流（cout）

```
cout << "这是字符串，后面接一个变量: "
     << a << endl;
```

重载<<运算符（流插入）：

```
ostream& operator<<(ostream& os,
                    const Student& rhs);
```

2) 标准输入流（cin）

```
cout << "请输入姓名和年龄： "; cin >> name >> age;
```

>>运算符（流提取）已关于各种内置类型重载，使之能正确解译和提取用户输入的内容，并将值存储在给定的变量中。

三. 面向对象技术

1. 类与对象
2. 类的继承
3. 抽象函数、抽象类
4. 多态性
5. 异常处理

1. 类与对象

OOP

类与对象是面向对象编程的灵魂。
类class：　描述所属对象的蓝图
对象object：　根据蓝图生成的具体实例instance
要使用对象，必须先定义类，然后再创建类的对象（实例）。C++标准库已包含大量的类。

1）**对象的创建和使用**
2）**类的声明**
3）**类成员的访问权限**
4）**实例成员和类成员**
5）**属性与索引**
6）**数组**
7）**String类**

1) 对象的创建和使用

OOP

- 对象是类的实例，属于某个已知的类。
- 对象声明的方式一：<类名> <对象名>;
  例： vector<string> vt； random rd;
- 对象声明的方式二： <类名>* <指向对象的指针变量> = new <与类名有相同名字的构造函数>(<参数列表>);
- new操作符用来创建新的对象，在堆中为之分配内存。并调用构造函数来初始化对象。
  stack* ps = new stack();
- 通过对象或指针调用类中定义的公有成员方法：
  <对象名>.<方法名>(<参数列表>)　s.Push("World")
  <指针名>-><方法名>（<参数列表>）ps->pop();

对象的生存周期

- 方式一：对象诞生于定义语句，函数退出时自动销毁。对象局部于函数
- 方式二：对象诞生于new，销毁于delete。通过指针操作，可以跨函数，但必须精准控制
- C#中，程序员只需创建所需对象，而不需显式地销毁它们。.NET的垃圾回收（gc）机制自动销毁不再使用的对象，收回对象所占的资源。
- C++没有GC

2) 类/结构的声明

OOP

在C++中使用class定义类，一般格式为:

```
[template模板声明] class <类名> [: <基类名>] {
        数据成员
        方法成员
    }
```

- C++的类是一种封装包括数据成员，方法成员的自定义类型（数据结构）。
  数据成员: 常量和域（成员变量）。
  方法成员: 函数，操作符函数，构造函数。
- 类的定义= 类声明 + 类主体
- 类声明：[template声明] class <类名> [: <基类名>]

C++程序组成

- 源程序一般分成实现文件.cpp和头文件.h：
  - #include指令包含头文件
  - 可选：引用命名空间或声明命名空间。 namespace
  - 类或结构的声明定义
  - 函数的声明/定义
- 类是组织相关数据和操作的模块单元。类中可以有：
  - 变量（字段，data field）声
  - 构造函数
  - 成员方法（method）

```
class Student{
  int studentID; string name;float score;//
    成员变量
  Student(){ //不带参数的构造函数
     name = "none";studentID=-1}
  // 下面是带参数的构造函数
  Student(int sid ,string sname){
     name = sname; studentID = sid; }
  string str();//公有方法
}
```


类的主体

OOP

- 域: [<修饰符>] [static] **<变量类型> <变量名>**
- 方法:
  [<修饰符>] [static] **<返回值类型> <方法名> (<参数列表>)** {<方法体>}
- 构造方法：具有与类名相同的名字，用于对类的实例进行初始化。

类/结构的定义举例

成员初始化列表

```
struct Student {
  int _studentID; string _name; double _score;
  Student() :_name("no name"){//缺省构造函数
    _studentID = 0; _score = -1.0;}
  Student(int id, const char* name, double
          score) :_name(name) {//构造函数
    _studentID = id; _score = score; }
  Student(const Student& s) :_name(s._name) {
    studentID=s._studentID; _score= s._score; }
  bool operator<(const Student& y) const {
    return _studentID < y._studentID;    }
  string str();
};
```

使用类

```
int main() {
  Student s1(172001,"Zhang San");
  Student s2;
  cout<<"Student #1:"<< s1.str();
  cout<<"Student #2:"<< s2.str();
}
```

输出结果：

```
Student #1: 172001-Zhang San
Student #2: -1-no name
```

类定义举例（二）

```
class Complex{
private:
  double _rp; // 复数的实部
  double _ip; // 复数的虚部
  static double eps;  // 缺省精度
public:
  Complex(double r=0, double i=0): _rp(r),
_ip(i){   }
  double real() const{ return _rp; }
  double& real(){ return _rp; }
  double imag() const{ return _ip; }
  double& imag(){ return _ip; }
  …… // 实现复数操作的其他相关方法
};
```

枚举类型enum class

基础

把一系列相关的符号常量组织成为一个单一类型称为枚举类型。enum class 枚举类型名{数据1，数据2，…，数据n};

例如，定义一个名为Color的表示颜色的枚举类型：

```
enum class Color{ Red,Green,Blue,White,Black};
Color c1  //定义一个枚举变量c1;
c1 = Color::Green;
 enum class Days {Sun,Mon,Tue,Wed,Thu,Fri, Sat};
 //第一个成员值默认为0,第二个为1，……
enum class Days {Mon=1,Tue,Wed,Thu,Fri, Sat, Sun };
enum class Days {mon=1,wed=3,sun，thu=8};
 // mon:1,wed:3,sun:4,thu:8
```


3) 实例成员和类成员

OOP

- 当创建类的一个对象时，每个对象拥有了一份自己特有的数据成员拷贝。这些为特有的对象所持有的数据成员称为实例数据成员，没有用关键字static修饰的成员就是实例成员。 **s.Count**
- 不为特有的实例所持有的数据成员称为类数据成员，或称为静态数据成员，在类中用static修饰符声明。不为特有的对象所持有的方法成员称为静态方法成员。
- 静态数据成员和静态方法成员通过类名引用获取。DateTime::Now()

4) 类成员的访问权限控制

OOP

- C++用多种访问修饰符来表达类中成员的不同访问权限，以实现面向对象技术所要求的抽象、信息隐藏和封装等特性。将类设计成一个黑匣子。

权限修饰符	本类	子类	其他类
公有的（public）	✓	✓	✓
保护的（protected）	✓	✓	
私有的（private）	✓		

2. 类的继承

OOP

继承（inheritance）是面向对象编程语言的基本要素之一

1) 类的继承

2) 派生类的声明

3) this指针/嵌套类

1) 类的继承

OOP

- 从现有类出发定义一个新类，称新类继承了现有的类。
  - 基类（base class）或父类
  - 派生类（derived class）或子类。
- class <派生类名>: [继承方式] <基类名>

```
class A {
    ……
}
class B: public A {
    ……
}
```

2) 隐藏和重写，虚方法

OOP

- 隐藏: 派生类继承基类中所有可被派生类访问的成员，当派生类成员与基类成员同名时，称派生类成员将基类同名成员隐藏。
- 虚方法与重写: 派生类可以重写基类中声明的虚方法，即为方法提供新的实现。这些方法在基类中用 virtual声明，而在派生类中，将被重写的方法用 override声明。

3） this指针/ 嵌套类

OOP

- 在类中可以通过this指针访问自身的成员。
- 一个类可以嵌套定义于另一个类中，称为嵌套类。与一般的类相同，嵌套类可以具有成员变量和成员方法。通过建立嵌套类的对象，可以存取其成员变量和调用其成员方法。

vector<int>::iterator it = v.begin();

3. 抽象函数与抽象类

OOP

1） 抽象方法
2） 抽象类

抽象方法与抽象类

OOP

- 当需要定义一个抽象概念时，可以声明一个抽象类，该类只描述抽象概念的结构，而不实现抽象方法。抽象类可以作为一个基类被它的所有派生类共享，而其中的方法由每个派生类去实现。
- 当声明一个方法为抽象方法时，则不需提供该方法的实现，但这个方法必须被派生类实现。声明抽象方法：virtual void f1() = 0;//纯虚函数
- 任何包含抽象方法的类即为抽象类。抽象类不能直接被实例化。
- 抽象类的派生类必须实现基类中的抽象方法，或者仍为抽象类。

4. 多态性(polymorphism)

OOP

- 多态是指“一个接口，多个方法”，即一个方法可能有多个版本，某个方法调用（invoke）可能是这些版本中的任何一种。

1）方法的重载(method overloading):同一范围中有多个同名的方法，但有不同的参数:

```
void sort(Iterator first, Iterator last);
void sort(Iterator first, Iterator last, Compare pred);
```

- 一个方法的名称和它的形参的个数及类型组成该方法的签名（Signature）。同一范围内方法的签名应是唯一的。
- 方法重载的价值在于，它允许通过使用一个普通的方法来访问一系列相关的方法。当调用一个方法时，具体调用哪一个版本则根据调用方法的参数由编译程序决定，编译程序将选择与调用的实参相匹配的重载方法。

2）方法的覆盖

OOP

- 通过为方法的定义引入virtual（虚方法）和override（方法覆盖/重写）关键字提供父类/子类间的方法多态的机制。 method overriding
- 方法的覆盖(method overriding): 类的虚方法是可以在该类的子类中改变其实现的方法。被子类改变的虚方法必须在方法头加上override来表示。
- 当通过指向对象实例的指针来调用某个虚函数时，实例的运行时类型(run-time type)决定哪个函数体被调用。

5. 异常处理

- 程序运行时错误的报告和处理方法首选使用异常（exception）。它使检测到错误的代码和处理错误的代码相互分离。
- <stdexcept>定义了多个标准异常类。exception基类：
  - runtime_error： overflow_error、range_error、underflow_error
  - logic_error： domain_error、length_error、out_of_range、invalid_argument
- 使用try/catch语句结构来处理异常，格式：

```
try{块}
若干catch 子句 {块}
```

指定在try块内发生异常时执行的代码

引发异常

throw 语句

功能：引发（报告）异常

格式：throw ；

throw 表达式 ；

- 不带表达式的 throw 语句只能用在 catch 块中，在这种情况下，它重新抛出当前正在由 catch 块处理的异常。
- 带表达式的 throw 语句抛出表达式的值。表达式必须是类型exception或从exception派生的类型的值。

```
if (i < 0 || i >= _length)
  throw out_of_range("index out of range: ");
```


四、 C++模板与泛型编程

- 泛型编程具备可重用性、类型安全和效率等方面的优点，这是非泛型编程所不具备的。C++使用模板（template）支持泛型编程，包括函数模板和类模板。
- 泛型应用类型参数的概念，用于设计与类型无关的类和方法（泛型编程）。在定义函数模板或类模板时，用某个符号作为类型参数，帮助完成类或函数的操作定义，客户端在应用模板时指定这些操作应使用的具体类型。
- 泛型通常与集合以及作用于集合的函数一起使用。C++标准库已包含很多基于泛型的集合模板类，如vector模板类。

泛型类应用举例：构造特定类型的列表

```
vector<int> a;  声明并构造int型数列表
a.push_back(86); a.push_back(100);向列表添加元素
vector<int> nums {0,1,2,3};声明并初始化
vector <string> s; 声明并构造字符串列表
s.push_back("Hello"); s.push_back("C++11");
vector< Student> st; 声明并构造Student列表
st.push_back(Student (8001, "王兵" , 92));
```


构造特定类型的列表

```
vector<Student> sl { {3016, "张超", 89},
   {3053, "马飞", 80}, {3041, "刘羽", 96},
   {3025, "赵备", 79}, {3039, "关云", 85}};
sl.insert(begin(sl)+2, Student(3000, "马超", 95));
for (const auto& item: sl)
    cout << item;
cout << endl;
```

当然，也可以创建自定义泛型类型和方法，以提供自己的通用解决方案，设计类型安全的高效模式。

泛型方法

- C++语言中泛型的优越性在下面的一段例子中应能较好的显示出来。对于同样的运算逻辑（例子中是交换两个变量的内容），但仅是数据的类型不一样，（传统模式）可能就需要定义一堆相似的函数或重载的函数；而应用泛型特性则可仅定义一个泛型方法（模板函数例子中是myswap）。

```
void swapint(int& x,int& y){
    int t = x;x = y;y = t;}
void swapdouble(double& x, double& y) {
     double t = x; x = y;y = t;}
template<typename T>
void myswap(T& x,T& y){T t = x;x = y;y = t;}
int main() {
  int a=3,b=7;cout<<"a="<<a<<"\tb="<<b<<endl;
  swapint(a, b); cout << "after swap"<<…;
  // 无泛型机制的年代
  double ad=3.5,bd=7.5; cout<<"ad="<<…<< endl;
  swapdouble(ad, bd);cout<< "after swap"<<…
  myswap(a, b);cout << "after swap"<<…;
  // 应用泛型机制的年代
  myswap(ad, bd);cout << "after swap" <<…;}
```

泛型方法

五、函数对象与Lambda表达式

- （狭义的数据）变量可以存储不同的值。函数以代码形式存储在内存，广义地说，各种函数也是数据。这类数据在传统C/C++中通过指向函数的指针来操作（传递、调用）。
- 函数对象指任何可调用的类型，该类型中重载了operator()函数，操作符()被称为调用操作符。
- 函数对象是面向对象的，函数对象实例只能赋以签名兼容的函数，具有类型安全的优点。
- 函数对象实例可以赋以不同的值（即具体函数），因而通过函数对象实例可以调用不同的函数。高阶函数
- 在C#中，这种特殊类型称作委托（delegate）。委托实例c可以引用不同的方法(f, g, h)，因而，通过同一个委托实例可以调用不同的方法实体。

两个常用的函数对象类型

- Predicate型函数对象和Comparison型函数对象。
  - function<bool(Student&)> predicateVar;
  - function<bool(Student&, Student&)> comparisonVar;
- Predicate型函数对象从形式上用来表示具有一个参数、返回值为布尔类型的函数，这类函数从功能上定义断言对象k满足特定条件的操作规则，用不同的返回值表达不同的断言结果（一般情况，返回值为true，说明对象k满足特定条件；返回值为false，说明对象k不满足特定条件）。

函数对象实例化：赋值签名兼容的函数

- Comparison型函数对象从形式上用来表示具有两个同类型的参数、返回值为布尔类型的函数，这类函数从功能上定义比较相同类型的两个对象（x和y）的操作规则，用不同的返回值表达不同的比较结果（x等于y，返回值为true；x不等于y，则返回值为false）。
- 函数对象实例可以赋值以与之签名兼容的函数，包括匿名函数，这称为函数对象实例化。如果已经有具有“bool comp1(Student& x, Student& y);”形式签名函数，则可以定义Comparison型函数对象实例c，用下列语句实例化使之指向comp1函数：
  function<bool(Student&, Student&)> c = comp1;

匿名函数与Lambda

- 如果临时需要一个能完成比较操作的函数，不一定非要为这样简短的代码设计一个专门函数名称，这时可以用匿名函数的形式定义所需的功能。
- Lambda表达式可方便简洁地表达匿名函数。例如，下面的匿名函数定义了以绝对值大小来决定孰大孰小的规则，并实例化委托变量c：
  function<bool(int, int)> c= [](int x,int y) {return abs(x) < abs(y); };
- 如果需要将一个整数容器（如vector或数组）按元素绝对值的大小排序，可以调用C++标准库中的sort算法，将委托实例c作为第三个参数传递给sort()函数，调用语句为：
  sort(first, last, c); //sort(first, last)是按自然序排序

Lambda表达式

- Lambda以捕获子句开头，捕获子句在一对“[]”符号中表示可以从周围范围捕获变量。接着，在Lambda表达式中可以声明匿名函数的形参表和函数体，函数体置于一对花括弧中，函数的返回类型常采用自动推导方式。
  - Predicate型举例：将匿名函数作为第三个参数传递给find_if函数，以查找出容器v中的第一个偶数位置。

```
vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto result= find_if(begin(v),end(v),
        [](int i){return i%2==0;});
```