C++11标准

一、C++标准的历史

1.1990年，The Annotated C++ Reference Manual，B.Stroustrup

2.1998年，ISO/IEC 15882:1998，C++98 \ C++98/03

3.2003年，ISO/IEC 15882:2003，C++03 /

4.2011年，ISO/IEC 15882:2011，C++11

二、C++标准的目标

1.C++98/03

1)比C语言更适合系统编程，且对C完全兼容；

2)支持数据抽象；

3)支持面向对象编程；

4)支持泛型编程。

2.C++11

1)更适合系统开发及库开发；

2)更加易学易用；

3)保持稳定，对C和C++98/03全面兼容；

4)扩充标准库，提供更多与时代需求相一致的功能。

三、编译选项

GCC编译器缺省语言标准是C++98/03，如果需要使用C++11标准，那么就需要显式指明：

GCC < 4.8: -std=c++0x

GCC >= 4.8: -std=c++11

四、类型推导

在程序代码中不显式指明变量的类型，而是由编译器根据代码的上下文隐式推导出该变量的类型，其推导规则类似于旧标准对函数模板类型参数的隐式推断。

1.auto

旧标准中auto关键字的作用和C语言一样，表示自动变量，是关于变量存储位置的类型饰词，通常不写，因为局部变量的默认存储就是auto。

void foo (void) {

int a; // \ 栈区

auto int b; // / 自动变量

static int c; // 数据区，静态变量

register int d; // 寄存器，寄存器变量

}

新标准中auto关键字已经不再表示变量的存储属性，而是用于类型推导。

1)auto的基本用法

参见：auto1.cpp

2)auto同指针或引用结合使用

参见：auto2.cpp

3)auto使用的限制

参见：auto3.cpp

4)何时使用auto

参见：auto4.cpp

2.decltype

1)计算表达式的类型

sizeof操作符的值是一个整数，表示类型的长度(字节数)。

typeid操作符的值是一个对象，其中包含了类型信息。

decltype操作符的值是一个类型，可用于其它对象的声明。

参见：decltype1.cpp

2)对于函数表达式，decltype将返回该函数返回值的类型，而对于左值表达式，decltype则返回该表达式的左值引用。

参见：decltype2.cpp

3)何时使用decltype

参见：decltype3.cpp

五、模板改进

1.类型别名

旧语法：typedef 源类型 目标类型;

typedef unsigned int uint\_t;

新语法：using 目标类型 = 源类型;

using uint\_t = unsigned int;

int a;

int& b = a;

旧语法中的typedef其源类型必须是具体类型，而新语法中的using其源类型除了具体类型以外还可以是模板或者半模板。

参见：using.cpp

2.函数模板和类模板一样都可以带有缺省参数，而且如果满足隐式推断的条件，函数模板参数的缺省值不一定非要写在参数表的最后。无法隐式推断的模板参数取缺省值，否则取隐式推断的类型，只要隐式推断生效，模板参数的缺省值即被忽略。

参见：default.cpp

3.连续出现的模板右尖括号不会再被误以为是右移运算符。

参见：right.cpp

六、初始列表

1.旧语法中对不同类型的变量初始化的形式也不同。

int a = 123; // 赋值式初始化

double b (1.23); // 构造式初始化

int c[] = {1, 2, 3}; // 花括号初始化

struct {

struct {

int year;

int mon;

int day

} bday;

char name[128];

int age;

} student = {{2000,1,1},"张飞",16}; // 花括号初始化

2.新语法以旧语法中的花括号初始化形式为基础，设计了列表初始化语法，统一了不同的初始化形式：

数据类型 变量 {初始化列表};

参见：init1.cpp

聚合类型：

1)任意类型的数组；

2)满足特定条件的类：

A.无自定义构造函数；

B.无私有和保护的非静态成员变量；

C.无基类

D.无虚函数

E.无通过"="或"{}"在类声明部分被初始化的非静态成员变量

3)聚合类型的元素或成员可以是聚合类型也可以是非聚合类型

4)对聚合类型使用列表初始化，相当于对其中的元素逐一初始化，而对非聚合类型使用列表初始化，相当于用初始化列表中的值作为参数，调用相应的构造函数

3.变长初始化表

参见：init2.cpp

template<typename T> class initializer\_list<T>;

轻量级列表容器，其中仅保存初始化列表元素的引用，而非其副本。

initializer\_list作为一个轻量级的链表容器，不但可以用在构造函数中，也可以作为普通函数的参数，传递不定数量的实参。相对于传统标准容器，效率更高。

vector<int> a {10, 20, 30};

void foo (vector<int> a) { ... }

foo ({10, 20, 30});

vector<int> bar (void) {

return {10, 20, 30};

}

bar ();

注意辨析轻量级容器和重量级容器的区别。

参见：init3.cpp

另外列表初始化还可以防止类型收窄，即对可能造成信息损失的类型转换，提示警告或直接报错。

参见：init4.cpp

七、范围循环

1.基于范围的for循环

for (元素类型 元素对象 : 容器对象) {

循环体;

}

如果循环体由单条语句或单个结构块组成，可以省略花括号。

用元素对象依次结合容器对象中的每一个元素，每结合一个元素执行一次循环体，直至容器中的所有元素都被结合完为止。

不依赖于下标运算，通用。

不需要访问迭代器，透明。

不需要定义处理函数，简洁。

参见：for1.cpp

2.范围循环的注意事项

参见：for2.cpp

vector<int> vi {...};

for (size\_t i = 0; i < vi.size (); ++i) // 不好

...

size\_t size = vi.size ();

for (size\_t i = 0; i < size; ++i) // 好

...

3.使自己定义的容器类型支持范围循环

一个类只要提供了分别用于获取起始和终止迭代器的begin和end函数，就可以支持基于范围的for循环。

参见：for3.cpp

八、函数绑定

1.函数对象

1)能够被当做函数调用的不一定就是函数，它们也可能是：

A.存放函数入口地址的函数指针；

B.实现了小括号运算符的类对象，亦称仿函数；

C.可被转换为函数指针的类对象。

2)可调用对象

象函数指针、仿函数以及可被转换为函数指针的类对象都被称为可调用对象，而它们的类型就被称为可调用类型。

3)函数对象

所谓函数对象，实际上就是一个function类模板的实例化对象，其中封装了以上三种可调用对象中的任何一种，对函数对象的调用，其实就是对被其封装的可调用对象的调用。

参见：fun.cpp

2.函数绑定器

1)函数绑定器可以将一个可调用对象和期望传递给该可调用对象的全部或部分参数绑定为一个函数对象，对该函数对象的调用就是在调用被其绑定的可调用对象。

函数对象 bind (可调用对象, 期望参数表)

期望参数表中可以包含期望传递给可调用对象的0到多个参数。

函数对象 () <=> 可调用对象 (期望参数表)

参见：func.cpp

2)placeholders::\_n是一个占位符，其值将由传递给函数对象的第n个(从1开始算)参数取代。

参见：bind1.cpp

3)无论类的普通成员函数还是静态成员函数都可以通过bind与其参数一起被绑定为函数对象，但是要注意作为类的普通成员函数需要连同传递给它的this指针一起绑定。

参见：bind2.cpp

4)函数绑定应用

参见：bind3.cpp

九、lambda表达式（嵌入式函数/匿名函数）

1.语法形式

lambda表达式定义了一个匿名函数，其本质就是一个匿名的仿函数对象：

[捕获表] (参数表) 选项 -> 返回类型 { 函数体 }

cout << [] (int x) -> int {

return x \* x;

} (100) << endl; // 10000

------------------------------

auto fun = [] (int x) -> int {

return x \* x;

};

class <编译器生成类名> {

public:

int operator() (int x) const {

return x \* x;

}

private:

成员变量来自捕获表;

};

lambda表达式的值就是<编译器生成类名>类型的对象。

cout << fun (100) << endl; // 10000

cout << fun (200) << endl; // 40000

------------------------------

int fun (int x) {

return x \* x;

}

cout << fun (100) << endl; // 10000

cout << fun (200) << endl; // 40000

参见：lambda1.cpp

2.捕获列表

[] - 不捕获任何外部变量

[d] - 按值捕获外部变量d

[&d] - 按引用捕获外部变量d

[this] - 捕获this指针

[=] - 按值捕获所有的外部变量(包括this指针)

[&] - 按引用捕获所有的外部变量(包括this指针)

[=,&d] - 按值捕获除d以外的外部变量(包括this指针)，按引用捕获外部变量d

[&,d] - 按引用捕获除d以外的外部变量(包括this指针)，按值捕获外部变量d

mutable: 在lambda中可以修改按值捕获的外部变量。这样的lambda表达式，其参数表不可省略。

3.不捕获任何外部变量的lambda表达式可以被隐式转换为原型相同的函数指针。

参见：lambda2.cpp

十、泛型元组

tuple可以理解为是对老版本pair类模板的扩展，其中的元素个数不再限于两个，且其功能更加丰富。

参见：tuple.cpp

十一、右值引用

1.左值和右值

可以取地址的值就是左值，左值通常具名；不可取地址的值就是右值，右值通常匿名。

参见：rvr1.cpp

2.左值引用和右值引用

1)左值引用只能引用左值，不能引用右值

类型& 左值引用 = 左值;

int a;

int& lvr = a;

int& lvr = 10; // 错误

2)右值引用只能引用右值，不能引用左值

类型&& 右值引用 = 右值;

int a;

int&& rvr = 10;

int&& rvr = a; // 错误

3)常左值引用既可以引用左值，也可以引用右值，万能引用

int a;

int const& lvr = a;

int const& lvr = 10;

4)任何临时值只具语句级生命期，引用可将其寿命延长至和该引用的生命期一样。

参见：rvr2.cpp

--------------

int a, b = 10;

a = b;

a + b

&a

--------------