C++11新特性与RAII思想

前言

C++11的新特性很多，部分内容简单学习了解，敲打用例，部分内容重点学习，列出资料链接。
点击了解更多C++11新特性

C++ 11新特（部分）

内容
auto&decltype
deleted函数&defaulted函数
final&override
explicit
新枚举类
for循环的新格式
委托构造函数
继承构造函数
nullptr
std::function
std::bind
lambda表达式
新的数据结构
模板的改进
智能指针
右值引用
并发
原子操作`std::atomic`
RAII思想

---

auto

C++98中的auto用于声明变量拥有自动生命周期，显然这个功能没用，于是C++11删除这一用法，取而代之的是：**自动类型推断**。

用法

> 1. 对于变量，指定正在声明的变量的类型将从其初始化程序自动推断出来。
> 2. 对于函数，指定将从其返回语句推导出返回类型。
> 3. 对于非型模板参数，指定将从参数推导出该类型。

例子

代码：

```
#include<iostream>
#include<vector>

int main(void)
{
    std::vector<int> arr(10);
    int num = 0;

    for(std::vector<int>::iterator it = arr.begin(); it != arr.end(); ++it)
    {
        *it = num++;
    }

    //自动推断类型
    for(auto it = arr.begin(); it != arr.end(); ++it)
    {
        std::cout << *it << ' ';
    }

    return 0;
}
```

运行结果：

```
0 1 2 3 4 5 6 7 8 9
```

可以发现第二个循环体内auto成功推断出迭代器类型，不在需要像第一个循环体一样手动输入冗长的类型。

---

decltype

decltype是C++新增关键字，用于根据给出的表达式或对象推导类型（我称之为**半自动类型推导**）。

用法

> decltype(对象或表达式)

例子

代码：

```
#include<iostream>
#include<vector>

class A
{
    public:
    void f();
};

inline void A::f()
{
    std::cout << "这里是A的方法f" << std::endl;
}

int main(void)
{
    A a;
    a.f();

    decltype(a) b;
    b.f();

    return 0;
}
```

运行结果：

```
这里是A的方法f
这里是A的方法f
```

文件编译通过，对象b成功运行A类成员函数void f()，证明decltype成功推断出对象a的类型。

---

deleted函数

有时候编译器会隐式生成特殊的成员函数，在特殊成员函数声明后加= delete，使函数成为**deleted函数**，能禁止编译器隐式生成它。

例子

代码：

```
#include<iostream>
#include<vector>

class A
```

```
{
    public:
    A & operator=(const A &a) = delete;
};

int main(void)
{
    A a;
    A b;

    a = b;

    return 0;
}
```

编译结果：

```
,,test.cpp: In function 'int main()':
test.cpp:15:9: error: use of deleted function 'A& A::operator=(const A&)'
     a = b;
         ^
test.cpp:7:9: note: declared here
     A & operator=(const A &a) = delete;
         ^~~~~~~~
```

编译报错，证明成功阻止编译器重载等号=，导致a = b无法通过编译。

---

defaulted函数

有些编译器并不会帮助程序员隐式生成特殊成员函数，于是需要在函数声明语句后添加= default，让函数成为**defaulted函数**，让编译器为显示声明的defaulted函数自动生成函数体。

---

final

final用于修饰一个类，表示禁止该类进一步派生和虚函数的进一步重载。

例子

代码：

```
#include<iostream>
#include<vector>

class A
{
    public:
    virtual void f(void){}
```

```
};

class B : public A
{
    public:
    void f(void) final {}
};

class C final : public B
{
    public:
    void f(void) {}
};

class D : public C
{

};

int main(void)
{


    return 0;
}
```

编译结果：

```
,,test.cpp:19:10: error: virtual function 'virtual void C::f()' overriding
final function
     void f(void) {}
          ^
test.cpp:13:10: note: overridden function is 'virtual void B::f()'
     void f(void) final {}
          ^
test.cpp:22:7: error: cannot derive from 'final' base 'C' in derived type
'D'
 class D : public C
       ^
```

结果证明成员函数被`final`修饰后无法重写，类被修饰后无法派生。

---

override

`override`用于修饰派生类中的成员函数，标明该函数重写了基类函数。

例子

代码：

```
#include<iostream>
#include<vector>

class A
{
    public:
    virtual void f1(void) {}
    void f2(void) {}
};

class B : public A
{
    public:
    void f1(void) override {}
    void f2(void) override {}
};

int main(void)
{


    return 0;
}
```

编译结果：

```
,,test.cpp:15:10: error: 'void B::f2()' marked 'override', but does not
override
     void f2(void) override {}
          ^~
```

结果证明`f2`不是`A`类的虚函数，导致编译到`B`类`f2`时报错。同理若`A`类**没有**`f2`函数，编译也会报错。

---

explicit

`explicit`用于修饰构造函数，表示只能显式构造，不能隐式转换。

例子

代码：

```
#include<iostream>
#include<vector>

class A
{
    public:
```

```
    explicit A(int i) { std::cout << "!!" << std::endl;}
};

int main(void)
{
    A a = 1;
    A aa(2);

    return 0;
}
```

编译结果：

```
,,test.cpp: In constructor 'A::A(int)':
test.cpp:7:20: warning: unused parameter 'i' [-Wunused-parameter]
     explicit A(int i) { std::cout << "!!" << std::endl;}
                    ~~~~^
test.cpp: In function 'int main()':
test.cpp:12:11: error: conversion from 'int' to non-scalar type 'A'
requested
     A a = 1;
           ^
```

若将`explicit`去除，程序则编译通过，证明`explicit`阻止了编译器将`A a = 1`隐式转换成`A a(1)`。

---

enum class

有作用域的枚举。能用这个就不要用传统枚举类型。原因请看例子。

例子

代码：

```
#include<iostream>
#include<vector>

enum A { a };
enum B { b };
enum class C { c };
enum class D { d };

int main(void)
{
    if( a == b )
    {
        std::cout << "a = b" << std::endl;
    }
```

```
    if( C::c == D::d )
    {
        std::cout << "c = d" << std::endl;
    }

    return 0;
}
```

编译结果：

```
,,test.cpp: In function 'int main()':
test.cpp:11:14: warning: comparison between 'enum A' and 'enum B' [-Wenum-
compare]
     if( a == b )
              ^
test.cpp:16:14: error: no match for 'operator==' (operand types are 'C' and
'D')
     if( C::c == D::d )
         ~~~~~^~~~
test.cpp:16:14: note: candidate: 'operator==(D, D)' <built-in>
test.cpp:16:14: note:   no known conversion for argument 1 from 'C' to 'D'
test.cpp:16:14: note: candidate: 'operator==(C, C)' <built-in>
test.cpp:16:14: note:   no known conversion for argument 2 from 'D' to 'C'
```

若把`if( C::c == D::d )`的代码块注释掉，程序不单单编译通过，并且判断`a == b`为真，这显然是错的。这就是不应该使用传统枚举类型的原因，因为能用不同作用的枚举参数进行对比。而不同类型的新枚举元素之间的比较甚至不能编译通过。而且新枚举类型甚至能选定底层类型。

---

for循环的新格式

以前的`for`循环格式为`for( ; ; )`，现在可以用基于范围循环的格式`for( : )`。

例子

代码：

```
#include<iostream>
#include<vector>

int main(void)
{
    std::vector<int> arr(10);
    int num = 0;

    for(auto it = arr.begin(); it != arr.end(); ++it)
    {
        *it = num++;
    }
```

```
    for(int n : arr)
    {
        std::cout << n << ' ';
    }

    return 0;
}
```

运行结果：

```
0 1 2 3 4 5 6 7 8 9
```

委托构造函数

委托构造函数支持在类的构造函数中调用自己的另一个构造函数，这能简化变量初始化的操作。

例子

代码：
如果是这样就很麻烦。

```
class A
{
    public:
    A(int a) { m_a = a; }
    A(int a, int b) { m_a = a; m_b = b; }
    A(int a, int b, int c) { m_a = a; m_b = b; m_c = c; }

    private:
    int m_a;
    int m_b;
    int m_c;
};
```

使用委托构造函数就轻松多了。

```
class A
{
    public:
    A(int a) { m_a = a; }
    A(int a, int b) : A(a) { m_b = b; }
    A(int a, int b, int c) : A(a, b) { m_c = c; }

    private:
    int m_a;
```

```
    int m_b;
    int m_c;
};
```

继承构造函数

当一个派生类的构造方法与基类完全一样，则使用继承构造函数能轻松很多。

例子

代码：
现在有基类A如下：

```
class A
{
    public:
    A(int a) { m_a = a; }
    A(int a, int b) : A(a) { m_b = b; }
    A(int a, int b, int c) : A(a, b) { m_c = c; }

    private:
    int m_a;
    int m_b;
    int m_c;
};
```

有个这样的B类，但是太麻烦了。

```
class B : public A
{
    public:
    B(int a) : A(a) {}
    B(int a, int b) : A(a, b) {}
    B(int a, int b, int c) : A(a, b, c) {}
};
```

若使用**继承构造函数**，简单多了。

```
class B : public A
{
    public:
    using A::A;
};
```

---

nullptr

在C++11之前空指针值均为`NULL`，而`NULL`并不是什么都没有的意思，它有时是`int`类型的`0`，有时是其他。而`nullptr`是真真正正的空指针。

---

std::function

`std::function`是一个函数包装器，可以包装可调用对象：**函数、函数指针、成员函数、静态函数、lamda表达式**和**函数对象**。当`std::function`对象未包装任何可调用实体又被调用该`std::function`对象时，抛出`std::bad_function_call`异常。

例子

代码：

```
#include<iostream>
#include<functional>

void f(int a)
{
    std::cout << a << std::endl;
}

int main(void)
{
    std::function<void(int)> fc = f;
    fc(100);

    return 0;
}
```

运行结果：

```
100
```

---

std::bind

实质上是适配器。将可调用对象与参数绑定到一起。

作用

1. 将可调用对象与参数一起绑定为另一个std::function供调用。
2. 将n元可调用对象转成m(m < n)元可调用对象，绑定一部分参数，这里需要使用std::placeholders。

例子

代码：

```
#include<iostream>
#include<functional>

void f(int a, int b)
{
    std::cout << a + b << std::endl;
}

void f2(int a)
{
    std::function<void(int)> fc = std::bind(f, std::placeholders::_1, 2);
    fc(a);
}

int main(void)
{
    f2(10);

    return 0;
}
```

运行结果：

```
12
```

---

lambda表达式

匿名函数，可以捕获一定范围的变量在函数内部使用，语法：

```
auto func = [capture] (params) opt -> ret { func_body; };
```

其中func是可以当作lambda表达式的名字，作为一个函数使用，capture是捕获列表，params是参数表，opt是函数选项(mutable之类)， ret是返回值类型，func_body是函数体。

一个完整的lambda表达式：

```
auto func1 = [](int a) -> int { return a + 1; };
auto func2 = [](int a) { return a + 2; };
cout << func1(1) << " " << func2(2) << endl;
```

如上代码，很多时候lambda表达式返回值是很明显的，c++11允许省略表达式的返回值定义。

lambda表达式允许捕获一定范围内的变量：

- []不捕获任何变量

- [&]引用捕获，捕获外部作用域所有变量，在函数体内当作引用使用
- [=]值捕获，捕获外部作用域所有变量，在函数内内有个副本使用
- [=, &a]值捕获外部作用域所有变量，按引用捕获a变量
- [a]只值捕获a变量，不捕获其它变量
- [this]捕获当前类中的this指针

例子

代码：

```
int a = 0;
auto f1 = [=](){ return a; }; // 值捕获a
cout << f1() << endl;

auto f2 = [=]() { return a++; }; // 修改按值捕获的外部变量，error
auto f3 = [=]() mutable { return a++; };
```

代码中的f2是编译不过的，因为我们修改了按值捕获的外部变量，其实lambda表达式就相当于是一个仿函数，仿函数是一个有operator()成员函数的类对象，这个operator()默认是const的，所以不能修改成员变量，而加了mutable，就是去掉const属性。

还可以使用lambda表达式自定义stl的规则，例如自定义sort排序规则：

```
struct A {
    int a;
    int b;
};

int main() {
    vector<A> vec;
    std::sort(vec.begin(), vec.end(), [](const A &left, const A &right) {
return left.a < right.a; });
}
```

新的数据结构

- std::forward_list：单向链表，只可以前进，在特定场景下使用，相比于std::list节省了内存，提高了性能
- std::unordered_set：基于hash表实现的set，内部不会排序，使用方法和set类似
- std::unordered_map：基于hash表实现的map，内部不会排序，使用方法和set类似
- std::array：数组，在越界访问时抛出异常，建议使用std::array替代普通的数组
- std::tuple：元组类型，类似pair，但比pair扩展性好

智能指针

C++11新增的智能指针只有三个：

1. `std::unique_ptr<T>`：删除了拷贝构造和赋值构造，只留下了移动构造和移动赋值运算符，独占资源所有权。
2. `std::shared_ptr<T>`：利用引用计数，共享资源所有权。
3. `std::weak_ptr<T>`：扮演`std::shared_ptr<T>`的观察者角色。

> **注意**
>
> 1. `std::unique_ptr<T>`与`std::shared_ptr<T>`都可以自定义删除器（deleter）。
> 2. 能用`make_xxx`就别用`new`。
> 3. 不要用同一个裸指针初始化多个`std::shared_ptr<T>`，不然会多次销毁同一个地方的资源导致出现`double_free`造成程序崩溃。
> 4. 成员函数返回类本身的`std::shared<T>`不能直接返回this，因为this是裸指针，应该使用`shared_from_this()`。
> 5. `std::shared<T>::get`可以获取裸指针，但是不要`delete`它，你自己销毁了，智能指针就不会干了。
> 6. 避免`std::shared<T>::get`循环引用的办法就是使用`std::weak_ptr<T>`。定义对象时，用强智能指针`std::shared<T>::get`，在其它地方引用对象时，使用弱智能指针`std::weak_ptr<T>`。

右值引用

C++11新增右值引用，我们一次性把左右值等概念描述一次：

- **左值**：可以取地址并且有名字的东西就是左值。
- **右值**：不能取地址的没有名字的东西就是右值。
- **纯右值**：运算表达式产生的临时变量、不和对象关联的原始字面量、非引用返回的临时变量、lambda表达式等都是纯右值。
- **将亡值**：可以理解为即将要销毁的值。
- **左值引用**：对左值进行引用的类型。
- **右值引用**：对右值进行引用的类型。
- **移动语义**：转移资源所有权，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用。
- **完美转发**：可以写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参。
- **返回值优化**：当函数需要返回一个对象实例时候，就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，这里有复制构造函数和析构函数会被多余的调用到，有代价，而通过返回值优化，C++标准允许省略调用这些复制构造函数。

详细请看

并发

`std::thread`

定义`std::thread`的三种情况：

1. 定义一个空的对象
2. 输入子线程执行函数及参数
3. 接管另一个子线程（移动语义）

`join()`与`detach()`的区别：

1. `join()`：阻塞主线程，直到子线程结束，说白了就是让主线程在任何情况下都要等到子线程结束自己才能结束。
2. `detach()`：主线程与子线程分离，就是说主线程和子线程的生命周期不再关联。

> 注：这两个函数开始执行意味着线程开始工作。

例子

代码：

```
#include <iostream>
#include <thread>

void function_one(int & x)
{
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<5; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
        ++x;
        std::cout << "x = " << x << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
}

void function_two(void)
{
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<10; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
}

int main(void)
{
    int x = 0;

    std::thread t1(function_one, std::ref(x));
    std::thread t2(function_two);
```

```
    if(t1.joinable())
    {
        t1.join();
    }

    if(t2.joinable())
    {
        t2.detach();
    }

    return 0;
}
```

执行结果：

```
id = id = 3 is running2 is running
id =
id = 23

id = x = 1
id = 3
2id =
x = 3
2id = 3

id = id = 2
3x =
3
id = 3
id = id = 3
2id =
3x = 4

id = id = 2
3x =
5id =
id = 3
2id =  is end
3 is end
```

std::mutex

互斥锁。
可以看到上一个用例的输出结果非常混乱，而且经过好几次实验，每次的输出都不同，原因就是两个子线程同时工作，自顾自地输出结果导致输出混乱，若要两个子线程按顺序输出就需要上锁。两个子线程公用同一个锁，当这个锁在一个线程中上锁之后，直到解锁前下一个线程都不能执行任务。

例子

代码：

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex locker;

void function_one(int & x)
{
    locker.lock();
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<5; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
        ++x;
        std::cout << "x = " << x << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
    locker.unlock();
}

void function_two(void)
{
    locker.lock();
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<10; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
    locker.unlock();
}

int main(void)
{
    int x = 0;

    std::thread t1(function_one, std::ref(x));
    std::thread t2(function_two);

    if(t1.joinable())
    {
        t1.join();
    }

    if(t2.joinable())
    {
        t2.join();
```

```
    }

    return 0;
}
```

> 注：为了让两个线程的输出结果都能显示完整，`t1`改用`join()`。

执行结果：

```
id = 2 is running
id = 2
x = 1
id = 2
x = 2
id = 2
x = 3
id = 2
x = 4
id = 2
x = 5
id = 2 is end
id = 3 is running
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3 is end
```

输出结果不乱了。

其他互斥锁

1. **递归式互斥量**：`std::recursive_mutex`。递归式互斥量是在同一个线程内互斥量没有解锁的情况下可以再次对其加锁，但其加解锁的次数需要保持一致。这种互斥量平时用得比较少。
2. **允许超时的独占式互斥量**：`std::timed_mutex`
3. **允许超时的递归式互斥量**：`std::recursive_timed_mutex`

锁封装

锁封装对于锁等于智能指针对于指针，目的就是为了减少mutex的操作降低风险。

锁封装有两个：

1. `std::lock_guard`：轻量级的

2. `std::unique_lock`：多了几个成员函数，面对条件变量需要用这个。

例子

代码：

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex locker;

void function_one(int & x)
{
    std::lock_guard<std::mutex> lock(locker);
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<5; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
        ++x;
        std::cout << "x = " << x << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
}

void function_two(void)
{
    std::lock_guard<std::mutex> lock(locker);
    std::cout << "id = " << std::this_thread::get_id() << " is running" <<
std::endl;
    for(int i=0; i<10; ++i)
    {
        std::cout << "id = " << std::this_thread::get_id() << std::endl;
    }
    std::cout << "id = " << std::this_thread::get_id() << " is end" <<
std::endl;
}

int main(void)
{
    int x = 0;

    std::thread t1(function_one, std::ref(x));
    std::thread t2(function_two);

    if(t1.joinable())
    {
        t1.join();
    }
```

```
    if(t2.joinable())
    {
        t2.join();
    }

    return 0;
}
```

执行结果：

```
id = 2 is running
id = 2
x = 1
id = 2
x = 2
id = 2
x = 3
id = 2
x = 4
id = 2
x = 5
id = 2 is end
id = 3 is running
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3
id = 3 is end
```

std::future

由于线程无视执行程序返回值，当异步操作中需要获取返回值有两个方法：

1. 使用全局变量
2. 使用std::future等工具

std::future分别与两个工具结合使用：

1. std::promise：绑定一个值，获取该值做异步处理。
2. std::packaged_task：绑定函数，获取函数返回值做异步处理。

例子

使用'std::promise'，代码：

```
#include <iostream>
#include <thread>
#include <future>

void fun(std::promise<int> &pro)
{
    pro.set_value(22);
}

int main(void)
{
    std::promise<int> pro;
    std::shared_future<int> fut = pro.get_future();
    std::thread thr(fun, std::ref(pro));

    thr.detach();

    fut.wait();//阻塞至结果可用
    std::cout << fut.get() <<std::endl;//应该输出22

    return 0;
}
```

执行结果：

```
22
```

使用'std::packaged_task'，代码：

```
#include <iostream>
#include <thread>
#include <future>

int fun(int x)
{
    return 3*x;
}

int main(void)
{
    std::packaged_task<int(int)> pt(fun);
    std::shared_future<int> fut = pt.get_future();
    std::thread thr(std::move(pt), 11);

    thr.detach();

    fut.wait();//阻塞至结果可用
    std::cout << fut.get() <<std::endl;//应该输出22
```

```
    return 0;
}
```

执行结果：

```
33
```

原子操作std::atomic

原子类型不会发生数据争夺，相当于自己加了锁。

例子

代码：

```
#include <iostream>
#include <thread>
#include <atomic>
#include <list>

using namespace std;

atomic_int iCount(0);

void threadfun1()
{
    for(int i =0; i< 20; i++)
    {
        printf("iCount:%d\r\n",  iCount++);
    }
}
void threadfun2()
{
    for(int i =0; i< 20; i++)
    {
        printf("iCount:%d\r\n",  iCount--);
    }
}

int main()
{
    std::list<thread> lstThread;
    for (int i=0; i< 10; i++)
    {
        lstThread.push_back(thread(threadfun1));
    }
    for (int i=0; i< 10; i++)
    {
```

```
        lstThread.push_back(thread(threadfun2));
    }

    for (auto& th: lstThread)
    {
        th.join();
    }

    int x = iCount.load(memory_order_relaxed);
    printf("finally iCount:%d\r\n",  x);
}
```

执行结果：

```
...
iCount:-7
iCount:-6
iCount:-5
iCount:-4
iCount:-3
iCount:-2
iCount:-1
finally iCount:0
```

RAII思想

说白了就是一个让资源自动销毁的“方法”，利用C++局部对象自动销毁的特性，创建类利用类的构造函数与析构函数管理资源，以此达到对资源的自动销毁。

智能指针不就是最好的例子了吗？