

C++ 11/14/17(Session 2)

Gong xuan Jun.2021

课程内容

- 右值引用
- 移动语义
- 移动赋值
- 智能指针
- 关于默认函数
- 禁止特定的方法
- 委托构造
- 继承构造
- 管理虚方法

右值引用

- C++11引入右值引用，使得标示符关联到右值，用&&表示。
- 出现在赋值表达式右边，但不能对其应用地址运算符的值。
- 右值包括字面常量，如x+y等表达式以及返回值的函数（条件是该函数返回的不是引用）。

- 右值引用的例子

```
int x=10;  
int y = 23;  
int && r1=13;  
int && r2=x+y;  
double && r3 = std::sqrt(2.0)
```

- 右值引用的目的是提供在涉及临时对象时可以选用的特定方法
- 右值引用带来的主要好处是能够利用右值引用实现移动语义的库代码，例如stl类现在都有复制构造函数、移动构造函数、复制赋值运算符和移动赋值运算符

移动语义 (move semantics)

- 复制构造函数在对象的复制操作过程中花费了很大的工作量。

下面的例子中，对temp对象由创建到删除，这个过程做了大量的无用功。

```
vector<string> allcaps(const vector<string> & vs){  
    vector<string> temp;  
    //...  
    return temp;  
}
```

```
vector<string> vstr; // 假设建立了一个20000字符串的vector, 每个字符串1000个字符长度  
vector<string> vstr_copy1(vstr);  
vector<string> vstr_copy2(allcaps(vstr));
```

如果将字符保留在原来的地方，并将vstr_copy2与其关联，将会避免上面的无用操作问题。

- 移动语义避免了移动原始数据，而只是修改了记录。

移动赋值

- 适用于构造函数的移动语义考虑也适用赋值运算符。
- 移动赋值的格式： `ClassName & operator=(ClassName && parameters)`.
- 移动赋值运算符的参数也不能是 `const` 引用，因其修改了源对象。
- 示例代码：

```
Useless &Useless::operator=(Useless &&f)  
{  
    if(this == &f)  
        return *this;  
    delete [] ps;  
    n = f.n;  
    pc = f.pc;  
    f.n = 0;  
    f.pc = nullptr;  
    return *this  
}
```

智能指针

简介

- 对于传统的动态内存分配，堆内存的释放必须由程序员保证，否则将会造成内存泄露
- 智能指针可帮助管理动态分配的内存，避免内存泄露
- 智能指针是行为类似指针的类对象，并使用模板实现
- 无论是正常离开还是因为异常离开，智能指针总调用delete来析构在堆栈上动态分配的对象
- C++提供了4种智能指针用于对从堆中分配的内存进行自动释放
 - *auto_ptr* : C++98引入，在 C++11被摒弃
 - *unique_ptr*:
 - *shared_ptr*:
 - *weak_ptr*

均包含在头文件<memory>中，且分别属于下面的命名空间

- *std::auto_ptr*
- *std::shared_ptr*
- *std::unique_ptr*
- *std::weak_ptr*

智能指针

unique_ptr(1)

- unique_ptr指针对内存有唯一的所有权，指针离开作用域时，释放所引用的内存

```
{  
    unique_ptr<CReport> ps (new CReport("Using unique_ptr"));  
    ps->comment(string("ps:"));  
  
    // the below code is ok?  
    unique_ptr<CReport> p1;  
    p1 = ps;  
    p1->comment(string("p1:"));  
}
```

智能指针

unique_ptr(2)

- 尽可能使用make_unique()创建unique_ptr (make_unique在C++14中引入)
- make_unique的声明如下:

```
template< class T, class... Args >  
    unique_ptr<T> make_unique( Args&&... args )  
{  
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));  
}
```

参见: 5G, *L2-L0/src/u1/PduDemux.cpp*

```
template< class T >  
    unique_ptr<T> make_unique( std::size_t size );  
e.g auto intArray = make_unique<int[]>(10);
```


智能指针

shared_ptr(一)

- shared_ptr是引用计数的智能指针，多个shared_ptr实例可指向同一块动态分配的内存：

```
shared_ptr<CReport> ps (new CReport("Using shared_ptr.));  
ps->comment(string("ps:"));  
shared_ptr<CReport> p1;  
p1 = ps;  
p1->comment(string("p1:"));  
ps->comment(string("after p1=ps:"));
```

参见 *CplaneServices.hpp*

智能指针

shared_ptr(二)

- *make_shared*声明:

```
template< class T, class... Args >  
shared_ptr<T> make_shared( Args&&... args );
```

- 应总是使用*make_shared*创建*shared_ptr*

```
auto reportSharedPtr = make_shared<CReport>("only one test");
```

- 不要使用*shared_ptr*管理指向C-Style数组的指针:

```
auto intArray = make_shared<int[]>(10);
```

智能指针

移动语义

- `shared_ptr`和`unique_ptr`支持移动语义

```
class A{};  
shared_ptr<A> fun()  
{  
    auto ptr = make_shared<A>();  
    //...  
    return ptr; // std::move() will be called  
}  
int main()  
{  
    shared_ptr<A> sharedPtr = fun();  
    return 0;  
}
```

新的类功能

default关键字

- 假定要使用某个默认的函数，而这个函数由于某种原因不会自动创建，在这种情况下，可使用关键字default显示的声明这些方法的默认版本

```
class A{  
    public:  
        A()=default;  
        // A ( ) {}  
        A(int a){cout<<"a="<<a<<endl;}  
        void print(){cout<<"print class A"<<endl;}  
};  
A objA;
```

参见下面的示例：

```
class ConcreteMME : public MME{  
    public:  
        ConcreteMME(Logger&, CplaneUESender&, services::TimerService&, MMEInformation::Configuration,  
            CentralQueue&,  
                std::unique_ptr<procedures::MultiBearerActivation>);  
        ConcreteMME(const ConcreteMME&) = default;  
        ConcreteMME(ConcreteMME&&) = default;  
        ConcreteMME& operator=(const ConcreteMME&) = default;  
        ConcreteMME& operator=(ConcreteMME&&) = default;  
        ~ConcreteMME() = default;
```

新的类功能

delete关键字(一)

- 如果禁止编译器使用特定的方法，可使用关键字delete, 如禁止复制对象，可禁用复制构造函数。

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

示例:

```
class IPAddress final{  
    public:  
        IPAddress();  
        explicit IPAddress(Version sourceVersion);  
        explicit IPAddress(const IPAddress& source);  
        template<typename T> explicit IPAddress(const T& value);  
        IPAddress(IPAddress&& source) = delete;  
  
        IPAddress& operator = (const IPAddress& source);  
        IPAddress& operator = (IPAddress&& source) = delete;  
    //...  
}
```

新的类功能

delete关键字(二)

- delete可用于任何函数， 一种可能的用法是禁止特定的转换：

```
class A{  
    public:  
    ...  
    void fun(double a){ cout<<“value=”<<a<<endl;}  
    void fun(float a)=delete;  
};
```

```
A objA;  
objA.fun(2.0);
```

新的类功能

委托构造

- 给一个类编写多个构造函数，可能会编写重复的代码
- C++11允许在一个构造函数的定义中使用另一个构造，从而达到简化代码的目的，这被称为“委托”，其使用成员初始化列表语法的变种，例如：

```
class A{  
    public:  
        A();  
        A(int);  
        A(int,double);  
        A(int,double,string);  
    private:  
        int a_;  
        double b_;  
        string str_;  
};  
A::A(int a,doulbe b,string str):a_(a),b_(b),str_(str){//....}  
A::A(int a):A(a,1.0,"hello"){//....}
```

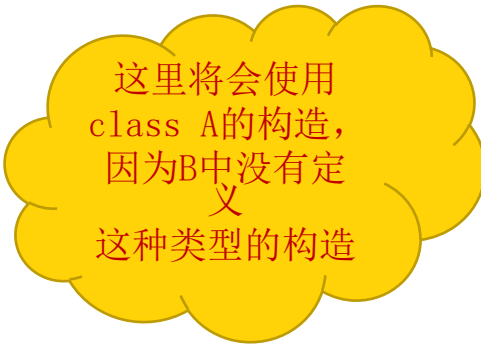
新的类功能

继承构造

- 传统C++中，构造函数如果需要继承，需要将参数传递，导致效率低下；
- C++11提供了一种让派生类能够继承基类构造函数的机制(默认构造函数，复制构造函数和移动构造函数除外)，但不会使用与派生类构造函数的特征标匹配的构造函数

```
class A{  
    public:  
        A(int a, double b){ //...}  
};  
class B:public A{  
    public:  
        using A::A;  
        B(int a){ //...}  
};
```

```
B objB (10) ;  
B objB(10,0.2);
```



这里将会使用
class A的构造，
因为B中没有定
义
这种类型的构造

新的类功能

管理虚方法 (override)

- 虚方法给编程带来一些陷阱，例如，如果基类声明了一个虚方法，而在派生类中提供不同的版本，这将隐藏旧版本。

```
class A
{
    public:
        virtual void f(char ch) const { std::cout<<ch<<"\n";}
}
class B : public A
{
    public:
        virtual void f(char *ch) const { std::cout<<ch<<"\n";}
}
B b;
b.f("c");
```

- C++11中可使用override指出要覆盖的一个虚函数, 如果声明与基类方法不匹配，编译器将视为错误

```
virtual void f(char ch) const override { std::cout<<ch<<"\n";}
```

新的类功能

管理虚方法 (override)

➤ 参见实例

```
class BearerManager
{
public:
    virtual BearerInformation allocate(BearerType) = 0;
    virtual void free(const BearerInformation&, BearerType) = 0;
    virtual ~BearerManager() = default;
};

class ConcreteBearerManager : public BearerManager
{
public:
    ConcreteBearerManager(BearerManagerConfig config);
    ConcreteBearerManager(std::vector<BearerGroup> bearerGroups, BearersPerBearerGroupCount
        maxNumberOfBearersPerBearerGroup, L2NRTEventQueueIDConfig l2HiQueueIDConfig);
    BearerInformation allocate(BearerType bearerType) override;
    void free(const BearerInformation& identifiers, BearerType bearerType) override;
    //...
};
```

新的类功能

管理虚方法 (final)

- 标识符**final** 用来禁止派生类覆盖特定的虚方法或者禁止基类被派生

```
class A
{
    public:
        virtual void f(char ch) const final { std::cout<<ch<<“\n”;}
}
class B : public A
{
    public:
        virtual void f(char ch) const {std::cout<<ch<<“\n”;}
}
```

新的类功能

管理虚方法 (final)

➤ 参加下面的实例

```
namespace endpoints{  
class S1APEndpoint final : public Endpoint, public endpoints::PayloadListener{  
public:  
    S1APEndpoint(Logger&, const Configuration::SCTP&, std::string address, unsigned port, CentralQueue&,  
        const std::string&);  
    S1APEndpoint(const S1APEndpoint&) = delete;  
    S1APEndpoint(S1APEndpoint&&) = delete;  
  
    S1APEndpoint& operator=(const S1APEndpoint&) = delete;  
    S1APEndpoint& operator=(S1APEndpoint&&) = delete;  
  
    void send(const Endpoint::MessageType&) final;  
    void onNewMessage(const messages::Envelope&) final;  
  
private:  
    void sendErrorIndication();  
    void sctpLinkFailureHandler();  
    CentralQueue& queue;  
    SctpEndpoint sctpEndpoint;  
};  
}
```

Q&A