C++ 11/14/17(Session 1)

Gong xuan Jun.2021

课程内容

- C++简介
- 作用域内枚举
- 关于命名空间
- 初始化
- 声明
- 类型推导
- 结构化绑定
- 基于范围的for循环

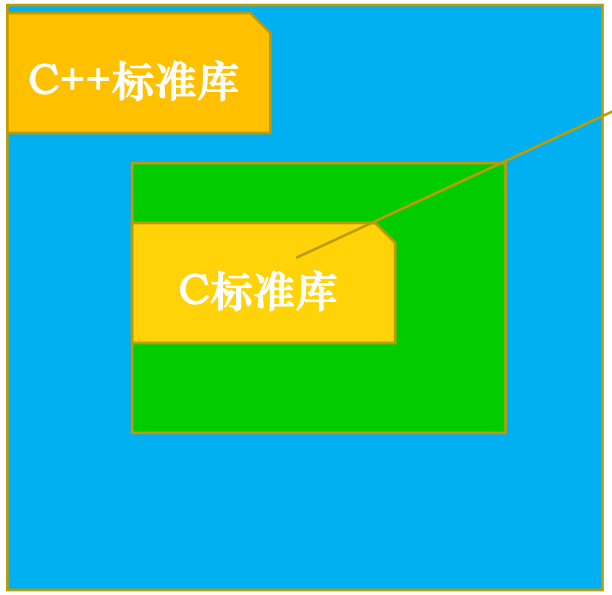
C++简介（一）

- 20世纪70年代，C语言诞生在贝尔实验室，其目的是为了解决UNIX系统跨平台问题
- C语言提供了自顶向下的结构化设计，鼓励程序员开发程序单元来表示各个任务模块
- 结构化编程在编写大型程序时，仍然面临挑战

OOP提供了一种新的方法，其强调的是数据，是试图让语言来满足问题的要求，而过程性编程试图使问题满足语言的过程性

C++简介（二）

- C++在20世纪80年代诞生于贝尔实验室，其创始人Bjarne Stroustrup在 C语言的基础上加入了OOP的特性，由此诞生了 C++
- C++并没有对C的组件做很大的改动，因此C++是C语言的超级。



1.头文件为<cname>,e.g<cstdio>
2.<cname>头文件中声明的所有名称都在std名称空间

C++融合了3种不同的编程方式，C语言代表的过程性语言、C++的类代表的面向对象语言以及C++模板支持的泛型编程。

C++简介（三）

- ANSI 和ISO致力于C++标准的制定，并于1998年获得了ISO、IEC和ANSI的批准，C++标准（ISO/IEC14882:1998） C++98诞生
- 不仅描述了已有的特性，还添加了异常、运行阶段类型识别（RTTI）、模板和STL(标准模板库)

- C++演化的特点

作用域内枚举（一）

- 传统枚举存在一些问题：
  - 两个枚举定义中的枚举量可能发生冲突：

```
enum A{E_RED,E_BLUE,E_GREEN};
enum B{E_RED,E_GRAY};
```

  - 并不是强类型的（并非类型安全的），其总被解释为整型数据,例如试图将A变量执行算数运算或将其作为整数对待，编译器可能报错 ,A eA = 0;

- C++11提供了一种新枚举，其枚举量的作用域为类. e. g.

```
enum class A{E_RED,E_BLUE,E_GREEN};
enum class B{E_RED,E_GRAY};
```

新枚举要求使用枚举名来限定枚举量：

```
A eA = A::E_RED;
B eB = B::E_RED;
```

参见实例：

```
enum class BearerType{
    SRB1,
    SRB2,
    DRB
};
```

作用域内枚举（二）

- 作用域内枚举不能隐式地转换为整型，但必要时可进行显示类型转换

```
int a = A::E_RED;//编译器将会报错
int a = int(A::E_RED) ; //显示类型转换
```

- 默认情况下，C++11作用域内枚举的底层类型为int类型,C++11还提供了一种语法，可用于做出不同的选择：

```
enum class ESHORT: short{E_RED,E_BLUE,E_GREEN};//将底层类型指定为short,参见示例
```

```
class IpAddress final{
   public:
    enum class Version : u8
    {
        VERSION_UNDEFINED = 0,
        VERSION_4 = 4,
        VERSION_6 = 6,
    };
//…
   }
```

名称空间

空间污染

- 当使用多个库时，开发人员所面临的常见问题之一就是名称冲突（也叫空间污染）。
- 利用名称空间特性，可以几乎完全解决这个问题。

例如下面的例子中类CFile被放置在名称空间Mylib中：

```
namespace Mylib{
class CFile{//…}
}
```

如果我们正在使用的库（在头文件UILib.h中声明）将它的名称包含在名称空间UILib 中,将不会有任何冲突：

```
namespace UILib{
    class CFile{ // …}
}
```

名称空间的使用（一）

- 对名称空间中的类的每个引用都必须限定，以澄清所引用的名称空间。
  例如上面的例子中如果要使用Cfile这个类，我们应该 加上名称空间：

```
UILib::CFile uifile;
Mylib::Cfile myfile;
```

将所有的声明和定义都封闭在名称空间中是一个好习惯，这样可以避免在全局名称空间中引起任何的名称冲突

- 可以创建没有命名的名称空间,下面的函数只能在声明这个未命名的名称空间的文件可见。

```
namespace {
  void TimeOfDay();
}
```

利用标准 c++中的名称空间，可以用未命名名称空间来代替静态函数。

名称空间的使用（二）

- 通过使用using 声明将名称空间的特定成员导出到范围中，例如：

```
void f(){
using MyLib::CFile; //using的声明只在f()作用域内有效
CFile objFile;
}
```

- 通过使用 using指令可以解除对整个名称空间的锁定以查找名称。

```
#include “UILib.h”
using namesapce UTLib;
CFile objFile;
```

所有C++库函数和类都在名称空间std中声明。

名称空间

嵌套命名空间(一)

- **传统的多层嵌套命名空间:**

```
namespace dynData
{
  namespace dl
  {
    class Data
    {
     //.....
     public:
      void f() {   std::cout << "f() called" <<std::endl; }
     //.....
     };
   }
 }
int main(){
 dynData::dl::Data data;
 data.f();
 return 0;
}
```

名称空间

嵌套命名空间(二)

- C++17后对多层嵌套命名空间做了简化：

```
namespace dynData::dl
{
  class Data
  {
    //.....
    public:
    void f() {   std::cout << "f() called" <<std::endl; }
    //.....
  };
}

int main(){
 dynData::dl::Data data;
 data.f();
 return 0;
}
```

初始化

统一初始化（一）

- C++11 之前，初始化类型并非总是统一的，看下面的例子：

```
struct SPoint{
      double x_;
      double y_;
    };
class Point{
       public:
           Point(double x, double y):x_(x),y_(y){}
       private:
          double x_;
          double y_;
     };
```

两种类型的初始化：

```
SPoint  p1={2.1,3.5};
Point  p2(2.1,3.5);
```

统一初始化（二）

- C++11 之后，可以使用统一的初始化方法初始化上面的例子

```
SPoint  p1={2.1,3.5};
Point  p2={2.1,3.5};
```

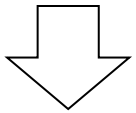
- 统一初始化可用于初始化C++中的任何内容

```
int a = 2;
int b(2);
int c = {2};
int d{3};
int e{}; // e将被初始化为0;
char strArray[]={“hello world”} //C-Style字符串初始化
string str = {“how are you”}; //string 对象初始化
short array[5] {4,5,2,76,1};   //初始化数组
int* p = new int[4] {2,4,6,7};  // 初始化动态分配的数组
```

统一初始化（三）

- 统一初始化可阻止窄化，C++隐式地执行窄化

```
void fun(int i){//…}
int main()
{
    int a= 3.14;
    fun(3.14); //调用fun之前 3.14将会被截断为3
    return 0;
}
```



```
int main()
{
    int a= {3.14};
    fun({3.14}); //编译时会发生什么？
    return 0;
}
```

统一初始化（四）

- 在类的定义中初始化成员，例如：

```
class  A
{
     public:
          A(){};
          A(short s):a(s),arr{0,1,2,3,4}{};
     private:
           short a;
           int b=10;
           int arr[5];

};
```

避免在构造函数中写重复的代码，如果构造函数在成员初始化列表中提供了相应的值，这些默认值将被覆盖

初始化列表

- 初始化列表简化了参数数量可变函数的编写，定义在头文件<initializer_list>
- 初始化列表中所有的元素都应该是同一种预定义类型

```
#include <initializer_list>
 int sum(initializer_list<int> lst)
 {
    int total  = 0;
    for(const auto& value:lst)
       total+= value;
  return total;
  }
 int a  = sum({1,2,3});
 int b = sum({10,20,30,40});
 int c = sum({1,2,3.0}); //编译会发生什么？
```

初始化

聚合初始化(C++17)

- 在初始化对象时，可用花括号对其成员进行赋值。
- 用大括号括起来的列表，可用于所有内置类型和用户定义的类型。

例如：

```
struct S {
    int x;
    struct Foo {
        int i;
        int j;
        int a[3];
    } b;
};
```

```
//执行初始化：
S s1 = { 1, { 2, 3, {4, 5, 6} } };
S s2 = { 1, 2, 3, 4, 5, 6}; // same, but with brace elision
S s3{1, {2, 3, {4, 5, 6} } }; // same, using direct-list-initialization syntax
S s4{1, 2, 3, 4, 5, 6};
```

条件分支语句初始化 (C++17)

- C++17标准中，if和switch语句有以下新形式（在if和switch中进行初始化）：

- if（初始化语句; 条件）语句
  else 语句
- switch（初始化语句; 条件）语句

例如下面的代码段：

```
set<string> strs;
if (auto [iter, success] = strs.insert("Hello"); success)
        cout << *iter << endl; // Hello
```

注：if中申明的变量，其作用域覆盖后续条件分支；
switch中声明的变量，其作用域覆盖整个switch语句。

声明

inline变量（一）

- 在c++中，类结构中静止初始化非const静态成员：

Case 1:

```
class myClass
{
  static int value=5;
};
```

Case 2:

```
//myClass.hpp, the head file was included by mulpiple Cpp files.
class myClass
{
  static int value;
};
myClass::value=5;
```

inline变量(二)

- c++17可以在头文件中定义一个内联的对象，如果这个定义被多个编译单元使用，它们都指向同一个惟一的对象，即编译器会自动链接到该变量：

```
//C++17 usage
//myClass.hpp, the head file was included by mulpiple Cpp files.
class myClass
{
  static constexpr int value = 10;
};
// For static data members, constexpr means inline since C + + 17
class myClass
{
  static inline int value = 10;
};

Inline myClass  globalObj;
```

类型推导

auto关键字

- C++11提供了多种简化声明的功能，尤其在使用模板的时候
- 关键字auto被用于实现自动类型推断 ，编译期根据初始值的类型推断变量的类型， 例如auto a=112 //a is type int
- 处理复杂类型时，自动类型推断的强大威力才能显示出来,看下面的代码

```
std::vector<int> intVec;
std::vector<int>::iterator iter = intVec.begin(); //C++98
auto iter = intVec.begin(); // C++11
```

参见下面的实例:

```
void BearerSetupProcedure::registerForBearerSetupResponse()
{
    auto successfulCallback = std::bind(&BearerSetupProcedure::handleBearerSetupResponse,
     this, std::placeholders::_1);
    auto timeoutCallback = [this]() { this->statusCallback(fsm::CompletionStatus::Timeout); };
    auto handler =
     std::make_unique<FunctorResponseHandler<messages::HiUser::BearerSetupResp>>(success
     fulCallback, timeoutCallback);
    ueDispatcher.registerForSingleResponse(std::move(handler), bearerSetupResponseTimeout);
}
```

类型推导

decltype关键字（一）

- 关键字decltype将变量的类型声明为表达式指定的类型，例如，

```
double x;
int n;
decltype(x*n) q
```

（decltype对模板很有用，因为只有等到模板被实例化时才能确定类型）

参见实例

```
  template <class T, typename I = decltype(std::begin(std::declval<T>()))>
typename std::enable_if<IsIterable<T>::value, I>::type begin(T& arr){
    return std::begin(arr);
}
template <class T>
typename std::enable_if<!IsIterable<T>::value, Iterator<T>>::type begin(T& arr){
    return {arr, 0};
}
```

类型推导

decltype关键字（二）

- 返回类型后置，在函数名和参数列表后面指定返回类型：

对于模板：templete< typename R typename T,typename U>

```
        R f(T t,U u){ …}//C++98
templete<typename T,typename U>
        auto f(T t,U u)->decltype(t*u){ …}//C++11
templete<typename T,typename U>
        auto f(T t,U u){ …} //C++14
```

参见实例：

```
auto Pools::getPools(const SubCell::PoolMapping& mapping) const -> PoolContainer
{
    PoolContainer foundPools;

    std::copy_if(pools.cbegin(), pools.cend(), std::back_inserter(foundPools), [&foundPools,
     &mapping](const auto& pool) {
        return foundPools.size() != mapping.size() && std::any_of(mapping.cbegin(),
     mapping.cend(), [&pool](const auto& poolToCompare) {
                return pool.id == poolToCompare.id && pool.type == poolToCompare.type;
            });
    });
```

结构化绑定(C++17)

多变量初始化（赋值）

- 用一对包含一个或多个变量的中括号，表示结构化绑定，但是使用结构化绑定时，须用auto关键字，即绑定时声明变量。

```
template<typename T, typename U>
struct MyStruct
{
   T key;
   U value;
};
std::list<MyStruct<int, double>> container;
container.push_back(MyStruct<int, double>{2,2.2});
container.push_back({3,3.3});
for(auto [key, value] : Container1)
{
      std::cout<<"key= "<<key<<","<<"value=" <<value<<std::endl;
}
```

结构化绑定(C++17)

返回多值

- 结构化绑定提供了类似其他语言中提供的返回多值的功能:

```
struct S
{
    double num1;
    long num2;
};

S foo(int arg1, double arg2)
{
    double result1 = arg1 * arg2;
    long result2 = arg2 / arg1;
    return {result1, result2};
}

auto [num1, num2] = foo(10, 20.2);
```

基于范围的for循环(一)

- C++11增加了基于范围的for循环,简化了传统的循环任务
- 对数组或容器类的每个元素执行相同的操作

```
double prices[5]={2.3,3.2,5.6,8.9};
for(auto x:prices)
  std::cout<<x<<std::endl;
```

如果需要修改数组中的元素，可以使用引用，改写上面的例子如下：

```
for(auto &x:prices)
  x  = x*0.9;
```

基于范围的for循环(二)

- 基于范围的for循环主要为STL而设计,参加下面的实例

```
auto createDrbTobeSetupList(const std::vector<user_management::UEBearerInformation>&
    bearerToBeSetupInformation, const std::unordered_map<types::S1AP::ErabId, types::QoSEntry>&
    bearerQoSEntries)
 {
  std::vector<messages::FsHLAP::DrbToBeSetupItem> drbTobeSetupList;
  for(auto& bearerInfo : bearerToBeSetupInformation)
  {

   drbTobeSetupList.emplace_back(messages::Multi::Builders::createDrbToBeSetup(bearerQo
   SEntries.at(*bearerInfo.erabId), bearerInfo));
  }
   return drbTobeSetupList;
  }
```

Q&A

/