现代C++特性介绍

张翔

技术工程基础研发组

- 类型推断*auto & decltype*
- 新特性概览
- 智能指针与垃圾回收
- 右值引用与移动语义
- *Lambda*表达式

类型推断

- *ParamType*是指针或者引用，并且不是通用引用(T&&)
- *ParamType*是一个通用引用(T&&)
- *ParamType*既不是指针也不是引用
- *expr*是数组和函数时为特例

```
template<typename T>
void f(ParamType param);

f(expr);
```

ParamType类型是啥?

类型推断——case1

- 指针的处理与引用的处理相似，以引用为例
- 如果*expr*是引用，剔除引用部分
- 然后根据*expr*与*ParamType*推断*T*类型

```
template<typename T>
void f(const T& param);

int x = 27;          //x是int
const int cx = x;  //cx是const int
const int& crx = x;//crx是const int&

f(x);     //T是int, Param是const int&
f(cx);    //同上
f(crx);  //同上
```

类型推断——case2

- 如果*expr*是左值，*T*和*ParamType*都被推断为左值引用
- 如果*expr*是右值，则对照*case*1的匹配方式

```
template<typename T>
void f(T&& param);

int x = 27;
const int cx = x;
const int& crx = x;

f(x);          //x是左值，故T是int&
               //param类型是int&

f(cx);         //cx是左值，故T是const int&
               //param类型是const int&

f(crx);        //crx是左值，故T是const int&
               //param类型是const int&

f(27);         //27是右值，故T是int
               //param类型是int&&
```

类型推断——case3

- 如果*expr*是引用类型，忽略引用
- 如果*expr*是*const*，*volatile*也忽略
- 最后进行匹配，与类型推断

```
template<typename T>
void f(T param);

int x = 27;
const int cx = x;
const int& crx = x;

f(x);       //T和param类型都是int

f(cx);      //T和param类型都是int

f(crx);     //T和param类型都是int
```

类型推断——auto

- 一般情况*auto*类型推断与函数模版相似分为：*auto&*、*auto&&*、 *auto*
- 大括号初始化推断是特例，*auto*可以，函数模版无法进行推断

```
auto x = {1, 4, 5, 6}; //initializer_list<int>

template<typename T>
void f(T param);

f({1, 4, 5, 6});        //error 🤢
```

- *auto*在函数返回值或者*lambda*中时，采用模版推断

auto优点

- 避免冗长的定义前缀

```
auto derefLess =
    [] (const auto& p_lh, const auto& p_rh)
    {return *p_lh < Ip_rh;}

传参: std::unique_ptr<MyClass>
```

- 应对*lamdba*和模版这种难以推断类型的场景

- 语法糖，避免错误使用类型

auto潜在的问题

- *auto*是类型推断，结果也许和预想不同
- *bool*为临时返回值，隐式转换出一个*bool*变量
- *auto*推断变量是一个*proxy*类型，带来悬挂问题

```
processWidget(Widget&, bool); //definition
std::vector<bool> features(const Widget& w);
//return a tmp

Widget w;

bool highPriority = features(w)[5];
//隐式转换，bool是一个新的val

processWidget(w, highPriority);
```

```
//operator[] -> vector<bool>::reference
auto highPriority = features(w)[5];

//tmp销毁，highPriority悬挂，5个偏移结果未定义
processWidget(w, highPriority);
```

有问题吗?

类型推断——decltype

- 推断类型时，结果最“符合”预期
- 对于左值表达式，推断为引用类型

```
int x = 0;
decltype(x);   //int
decltype((x)); //int&
```

- 函数返回值，*lamdba*，变量初始化中使用

```
const Widget& cw = w;
auto autotype = cw;  //Widget
decltype(auto) detype = cw; //const Widget&
```

新特性——通用

- using别名机制代替typedefs，支持模版类型
- 使用nullptr避免NULL导致的重载函数误用
- 引入强类型*Enum*，限制作用域范围，可配合*auto*
- 使用noexcept代替throw，性能优化，移动语义
- 引入constexpr，优化性能

```
constexpr int pow(int base, int exp) noexcept;
constexpr auto numExp = 6;
std::array<int, pow(2, numExp)> ret_arr;
```

新特性——类构造

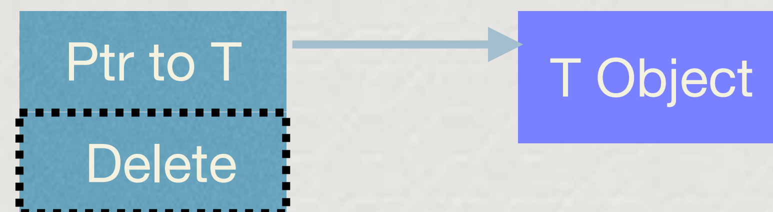
- 引入{}构造形式，拒绝表达式隐式转换构造；可能的情况下{}优先匹配列表构造函数 `initializer_list<T>`
- 引入deleted函数，代替private禁止访问技巧
- 引入override关键字，避免继承中误定义重载函数

新特性——线程并发

- 语言层面的标准并发库，引入了基于任务(task_based)的编程模式
- std::async由标准库负责调度，降低开销
- std::async不承诺异步、并发；对于thread_locals的编程模式有影响。

智能指针——unique_ptr

- 开销小基本等于裸指针，性能高，只有移动所有权语义(*move-only*)

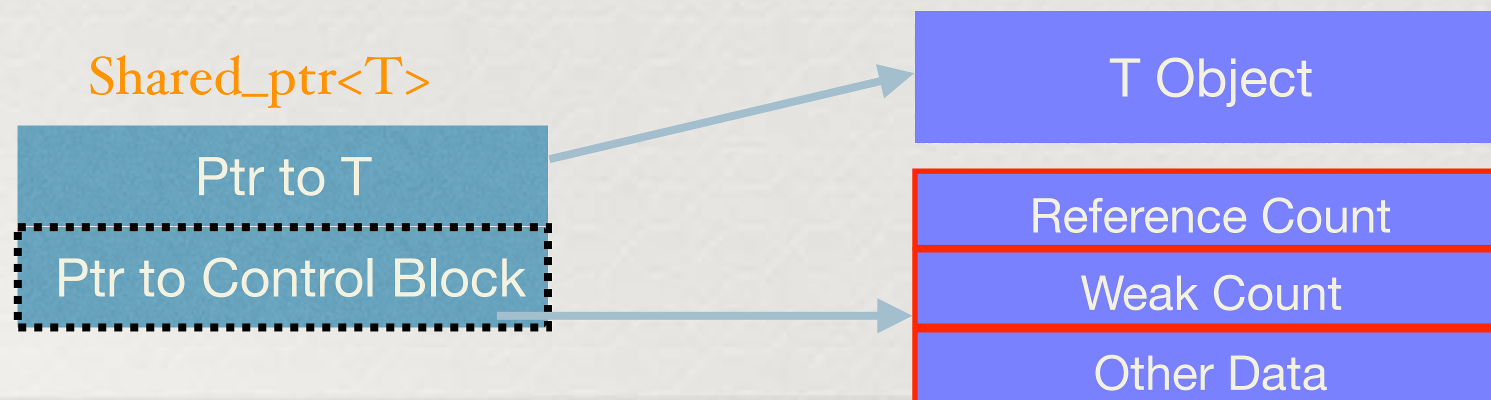


- 可自定义销毁函数，*delete*也是*unique_ptr*类型一部分

```
auto delInvemt = [] (Investment* pInvestmnet){
                            LOG();
                            delete pInvestmnet;
                        };
std::unique_ptr<Investment, decltype(delInvmt)> pInv(nullptr, delInvmt);
```

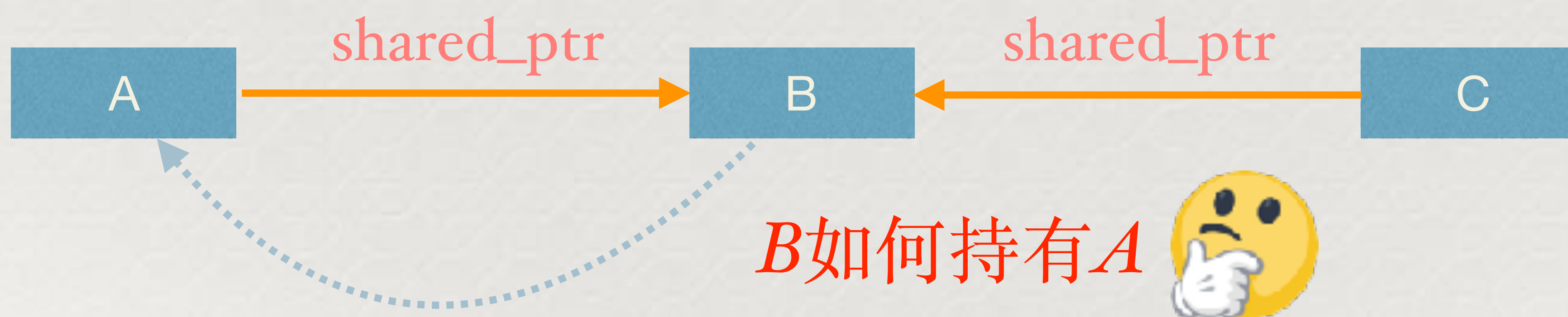
智能指针——shared_ptr

- 灵活的垃圾回收机制，简单的生命周期管理
- 可自定义销毁函数，*delete*不是shared_ptr类型部分
- 大小一般为裸指针两倍，引用计数操作需要确保原子性，保存计数的内存必须动态分配



智能指针——weak_ptr

- 观察者模式的弱引用机制
- 用例：*muduo*中的timewheel；类依赖中避免循环依赖



智能指针——make

- 与new相比make语法更简洁，避免安全问题（裸指针销毁、异常内存泄漏），空间占用少，性能高

```
int getPriority(); //may throw a exception

processWidget(shared_ptr<Widget>(new Widget),
              getPriority());
```

- 无法使用自定义的销毁函数。
- 可能会引起*GC*延迟（weak计数导致），造成变相的“内存泄漏”，对象内存大时影响显著

右值引用与移动语义

- Q：什么是右值？什么是右值引用？变量？

```
void f(Widget&& w);
```

w是什么

- Q：move函数机理是什么？实现什么功能？
- Q：移动语义和右值机制是不是性能的代名词？

移动语义-move&forward

- move函数除了类型转换(cast)，什么事儿也不做

```
template<typename T>
decltype(auto) move(T&& param)
{
  using ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(param);
}
```

- move的作用是准备好被移动的数据；不作移动操作，不承诺一定能调用移动构造函数
- 不要试图对一个const 对象执行move&构造，南辕北辙

T&&

- auto&&和模版中的T&&可不是右值引用
- 不精确的类型推断，type&&是右值引用
- 通用引用在类型推断中，将右值推断为右值引用，将左值推断为左值引用

```
void f(Widget&& param);

Widget&& var1 = Widget();

auto&& var2 = var1;

template<typename T>
void f(T&& param);


template<typename T>
void f(verctor<T>&& param);
```

移动语义总结

- move不是银弹，不一定带来巨大的性能提升

- 移动无效：没有移动构造函数；对象移动性能无提升；对象不支持移动操作；操作对象是左值

- 不要臆想，不要假定对象可移动；不假定性能

lambda表达式

- 默认的引用捕获会带来“悬挂”引用问题；默认的拷贝捕获可能会带来指针(this)悬挂问题

```
void Widget::addFilter() const
{
  filters.emplace_back(
   [divisor = divisor](int value)
   {return value % divisor == 0;}
  )
}
```

- *decltype+auto*&&+forward可以在*lambda*实现完美转发

```
auto func = [](auto&& x)
            {return normalize(std::forward<decltype(x)>(x));};
```

- 使用*lambda*代替*std::bind*，提高可读性和性能

reference

- *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf*
- *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf*
- *Effective Modern C++, Scott Meyers*
- c++ primer 5th
- 深入理解C++1

Thanks