C++ 11 新特性

（部分）

潘建瑜@MATH.ECNU

目录页

Contents

- 数据类型相关：auto, decltype, typeid, size_t, …
- 初始化 { }
- 指针相关：nullptr，智能指针
- 基于范围的 for 循环
- 匿名函数：lambda 表达式
- 更多库：random（随机数），chrono（时间），cstdint (int8_t, uint8_t, …)，complex（复数），…

编译时需要加上选项： -std=c++11

数据类型相关

- auto, decltype, typeid
- size_t, wchar_t
- constexpr

auto

auto 类型推导

- 根据初始化表达式对变量类型进行推导

```
auto x ; // ERROR
auto x = 2; // OK
for (auto i = 0; i < n; i++)  // OK
{... ...}
```

ex17_auto.cpp

- 不能用在形参中，如: int add(auto x, auto y) // ERROR
- 不能用于数组，如: auto x[2]={1,2}; // ERROR

decltype

decltype 类型推导

- 在编译时期进行类型推导，分析表达式获取其类型

ex17_decltype.cpp

```
int a = 2;
auto c = 0.0;
decltype(a+c) d;  // double d;
```

- 与 auto 类似，但用于不同的场景：decltype 的主要用途是推导表达式的类型，而不是像 auto 那样从变量的初始化表达式中推导类型。

typeid

typeid 获取表达式的类型信息

```
#include <typeinfo>
```

- 结果保存在一个 type_info (标准库类型) 的对象中，并返回该对象的常引用
- 需要通过 type_info 的成员函数查看相关信息

```
typeid(DataType)
typeid(expression)
```

ex17_typeid.cpp

```
typeid(double).name()
typeid(a+b).name()
```

size_t、wchar_t

size_t　无符号整型

- 通常用于数组索引和循环计数，以及动态内存分配等
- sizeof() 的返回结果是 size_t 类型的

ex17_datatype.cpp

wchar_t　宽字符型

- 用来记录一个宽字符的数据类型

除了基本数据类型外，为了方便记忆，通常会定义一些易于理解的基本数据类型变体（由 typedef 定义的别名）

constexpr

constexpr 常量表达式

- 值不会变（与 const 类似）
- 在编译时就能得到计算结果（const 可能会在运行时才得到结果）

- 编译器保证 constexpr 对象是常量表达式，即在编译时就能得到结果
- constexpr 函数（能用于常量表达式的函数）可以把运行期的计算迁移至编译期，从而使得程序运行更快（但会增加编译时间）

初始化 { }

- C++ 11 中，可以使用大括号进行初始化

```
int a = 3;
int b(4);
int c{5};
```

ex17_initialization.cpp

指针相关

- 空指针：nullptr，替代给空指针赋值时的 0 和 NULL
- 智能指针：自动管理内存（释放内存）

#include <memory>

智能指针

- 智能指针是 C++ 中用于自动管理内存的一种工具，它通过封装原始指针，确保在适当的时候释放内存，从而避免内存泄漏，主要用于持久动态内存分配。
- 智能指针包括：unique_ptr，shared_ptr，weak_ptr，auto_ptr (已弃用)

unique_ptr

- 独占资源所有权，同一时间只能有一个 unique_ptr 指向特定内存
- 离开 unique_ptr 对象的作用域时，会自动释放资源

智能指针

shared_ptr

- 共享资源所有权，多个 shared_ptr 可以指向同一内存，内存在最后一个 shared_ptr 被销毁时释放

week_ptr

- 与 shared_ptr 一起使用， 一个 weak_ptr 对象看做是 shared_ptr 管理的资源的观察者，它不影响共享资源的生命周期。
- 需要使用 weak_ptr 正在观察的资源，可以将 weak_ptr 提升为 shared_ptr。
- 当 shared_ptr 管理的资源被释放时，weak_ptr 会自动变成 nullptr

基于范围的 FOR 循环

```
int x[] = {1, 5, 2, 6, 8};
for (auto i : x)
    cout << i << endl;
```

ex17_for_range.cpp

Lambda 表达式

Lambda 表达式

Lambda 表达式/函数

- lambda 表达式可以看作是匿名函数。
- Lambda 表达式可以像对象一样使用，比如将其赋给变量或作为参数传递，也可以像函数一样对其求值。

Lambda 表达式一般形式

```
[capture](parameters) ->type { 函数体 };   // 注意：最后以分号结束
```

- `capture`：捕获当前作用域内的变量，以便在函数中直接引用，`[]` 不能省
- `parameters`：参数列表，如果没有参数，则小括号可以省略
- `->type`：返回值类型

Lambda 表达式

```
[capture](parameters) ->type { 函数体 };
```

- 捕获方式
  - 值捕获：只能访问变量的值，不能修改变量的值
  - 引用捕获：可以访问变量的值，也可以修改变量的值
  - 混合捕获

[] 不能省略，即使捕获列表为空。

不能捕获全局变量和静态变量。

ex17_lambda.cpp

```
int main()
{
   int a = 3, b = 4;
   auto f1 = [a] () -> int {return 2*a;};
   auto f3 = [=] { return a+b; };  // 值捕获所有变量
   atuo f3 = [a] int {a = 5; return 2*a;};  // ERROR，值捕获，不能修改
   auto f4 = [&a] {a = 5; return 2*a;};  // OK
   auto f5 = [&] {a = 5; return 2*a;}; // 引用捕获所有变量
   auto f6 = [=,&a] {a = 5; ...}; // 引用捕获a，值捕获其他变量
   . . . . . .
}
```

Lambda 表达式

```
[capture](parameters) ->type { 函数体 };
```

- 返回类型
  - 指定返回值的类型
  - 如果不指定，则编译器会根据代码实现推导一个返回类型
  - 如果没有返回值，则忽略此部分

```
int main()
{
   int a = 3, b = 4;
   auto f1 = [a] {return 2*a;};
   atuo f2 = [a] {a = 5; return 2*a;};  // ERROR
   auto f3 = [=] { ... };  // 捕获所有变量
   auto f4 = [&a] {a = 5; return 2*a;};  // OK
   auto f5 = [&] {a = 5; return 2*a;}; // 捕获所有变量
   . . . . . .
   return 0;
}
```

Lambda 表达式

- Lambda 表达式可以添加限定符 mutable

```
[capture](parameters) mutable ->type { 函数体 };
```

  - 如果加上 mutable，则表示允许修改通过值捕获的外部变量的**拷贝**（外部变量的值保持不变）
  - 如果需要加限定符 mutable，则参数列表中的小括号不能省，没有参数也不能省

```
int main()
{
   int a = 3, b = 4;
   auto h1 = [a] () mutable { a = 8; return a;};
   cout << “h1 : ” << h1() << endl; // 8
   cout << “a = ” << a << endl;  // 3

}
```

Lambda 表达式

优点

- 简洁：可以在一个表达式中定义，不需要单独声明定义（特别是可以在函数内定义）
- 方便：可以捕获外部变量，使得函数更加灵活和易用
- 通用：可以用在任何需要函数的地方，如算法、线程等

缺点

- 可读性不如普通函数，可能过于简洁，增加理解难度
- 复杂性：如果包含复杂的逻辑和控制流，可能增加代码调试和维护成本
- 性能：可能会产生额外的开销，如捕获外部变量

更多库

- chrono：时间管理
- random：随机数
- cstdint：int8_t, uint8_t, int16_t, uint16_t, …
- complex：复数

#include <chrono>

- 提供了一套丰富的工具来测量时间间隔（可达纳秒级别）、执行时间点的计算以及处理日期和时间，是 C++ 标准库中处理时间相关操作的核心部分。
- 三个基本概念：时间点，持续时间，时钟(Clocks)

```
auto T0 = chrono::system_clock::now();   // 获取时间点，与系统时间同步
// auto T0 = chrono::steady_clock::now();   // 不受系统时间影响
... ...
auto T1 = chrono::system_clock::now();
auto TT = chrono::duration_cast<chrono::seconds>(T1 - T0); // 计算持续时间
```

#include <chrono>

```
S0 = chrono::high_resolution_clock::now();   // 更高精度的时间
... ...
S1 = chrono:: high_resolution_clock ::now();
auto SS1 = chrono::duration_cast<chrono::seconds>(S1 - S0); // 秒
auto SS2 = chrono::duration_cast<chrono::milliseconds>(S1 - S0); // 毫秒
auto SS3 = chrono::duration_cast<chrono::microseconds>(S1 - S0); // 微秒
auto SS4 = chrono::duration_cast<chrono::nanoseconds>(S1 - S0); // 纳秒
```

```
auto now = chrono::system_clock::now();
time_t now_c = chrono::system_clock::to_time_t(now);
cout << "Current date and time: " << ctime(&now_c);  // 当前时间
```

ex17_chrono.cpp

更多功能参见库文件说明。

#include <random>

- 提供了满足各种分布的随机数生成工具，为需要随机数的程序提供广泛的选择
- 三个主要组件：随机数引擎，随机数分布，随机数适配器

```
random_device rd;   // 生成随机种子
default_random_engine gen(rd()); // 指定随机数引擎
uniform_int_distribution<> dist_int(1, 100); // 指定随机数分布
cout << "Uniform integer: " << dist_int(gen) << std::endl;
```

ex17_random.cpp

#include <cstdint>

- 定义了一组固定宽度的整数类型，在不同的平台上具有相同的大小和表示范围

```
int8_t, uint8_t
int16_t, uint16_t
int32_t, uint32_t
int64_t, uint64_t
intmax_t, uintmax_t
```

ex17_cstdint.cpp

#include <complex>

- 提供对复数运算的支持
- 基本运算：加 +、减 -、乘 *、除 /、共轭 conj、模 abs、幅角 arg

```
complex<float> z1(1,2);
cout << “z1 = ” << z1 << endl;  // 输出
cout << “real part of z1 = ” << z1.real() << endl;  // 实部
cout << “|z1| = ” << abs(z1) << endl;  // 模
```

ex17_complex.cpp

谢 谢

THANK YOU