深蓝学院
shenlanxueyuan.com

C++ 基础

第 14 章：元编程

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

元编程的引入

- 从泛型编程到元编程
  - 泛型编程——使用一套代码处理不同类型
  - 对于一些特殊的类型需要引入额外的处理逻辑——引入操纵程序的程序
  - 元编程与编译期计算

我们平时写的程序都是在运行期引入计算。元编程可以理解为是在编译期的计算
老师更喜欢用编译期计算这个名字来代替元编程.但是对于c++来说，元编程代表
编译期计算。对于别的语言来说，不是这样的。

- 第一个元程序示例（Erwin Unruh）
  - 在编译错误中产生质数
- 使用编译期运算辅助运行期计算
  - 不是简单地将整个运算一分为二
  - 详细分析哪些内容可以放到编译期，哪些需要放到运行期
    - 如果某种信息需要在运行期确定，那么通常无法利用编译期计算

元编程的引入——续

- 元程序的形式
  - 模板， constexpr 函数，其它编译期可使用的函数（如 sizeof ）
  - 通常以函数为单位，也被称为函数式编程
- 元数据
  - 基本元数据：数值、类型、模板
  - 数组 编译期数组我们是采用边长模板的方式来定义。
- 元程序的性质
  - 输入输出均为“常量” `constexpr int x = 3;`编译期常量
  - 函数无副作用
    编译期可被调用
- type_traits元编程库
  - C++11 引入到标准中，用于元编程的基本组件

//元程序：模板

```
template <int x>
struct M {
  constexpr static int val = x + 1;
//因为我们标注这个是编译期常量constexpr
, 所以这个是一个编译期完成的计算
};

int main() {
  return M<3>::val;
}
```

元程序：constexpt

```
constexpr int fun(int x) {
  return x + 1;
}
constexpr int val = fun(3);
int main() {
  return val;
}
```

```
//元数据：数组
template<unsigned... T> struct Cont; //类
模板声明，这个就相当于我们定义了一个数组
template<auto... T> struct Cont1;//另外一个
数值的数组
template<typename... T> struct Cont3;//类型
数组
template<template <typename> class... T> struct Cont4;
//模板数组,这个是一个可变长度的，这里面的每个元素都是
一个模板，这个模板可以接受一个类型作为模板形参。
```

```
//函数无副作用
int fun(int input) {
  static int x = 0;
  return input + (x++);
} //这个函数就是有副作用的，因此不能在编译期被
调用
//无副作用的概念就是对于相同的输入，能产生
相同的输出。
```

顺序、分支、循环代码的编写方式

- 顺序代码的编写方式
  - 类型转换示例：为输入类型去掉引用并添加 const
  - 代码无需至于函数中
    - 通常置于模板中，以头文件的形式提供
  - 更复杂的示例：
    - 以数值、类型、模板作为输入
    - 以数值、类型、模板作为输出
  - 引入限定符防止误用 `public, private`
  - 通过别名模板简化调用方式

去掉引用并添加const

```
#include <type_traits>

template <typename T> // T是输入
struct Fun {
  using RemRef = typename std::remove_reference<T>::type;
  //这个相当于程序执行的第一步，我们给定输入，来把
它的引用去掉。RemRef是等于type的，type是一个依赖形
的名称，这个依赖的东西，间接依赖我们的模板名称。
  using type = typename std::add_const<RemRef>::type;
//这个就是一个元函数，输入是T,返回是type
};//你得习惯，这个东西就是一个元函数，
int main() {
  Fun<int&>::type x = 3;  // const int x = 3;
}


以数值、类型、模板作为输入
template <typename T, unsigned S>
struct Fun {
  using RemRef = typename std::remove_reference<T>::type;
  constexpr static bool value = (sizeof(T) == S);
};//我以数值和类型作为输入，返回一个数值。去掉输入类型的引用，
并判断输入的类型的size是不是和输入的数值相一致。

//引入别名来简化调用方式
template <typename T, int S>
constexpr auto Fun_ = Fun<T,S>::value;

int main() {
  constexpr bool res = Fun<int&, 4>::value;// 等号右边就是
一个函数
  Fun<double&, 4>;// 这个就是上句的简化版本。

  Fun<int&, 4>::RemRef x; //你也可以直接用RemRef来作为一个type的声明
}
```

顺序、分支、循环代码的编写方式

- 分支代码的编写方式
  - 基于 if constexpr 的分支：便于理解只能处理数值，同时要小心引入运行期计算
  - 基于（偏）特化引入分支：常见分支引入方式但书写麻烦
  - 基于 std::conditional 引入分支：语法简单但应用场景受限

    通常来说只能应用于类型。它很像运行期的一个三元表达式。但是很多程序不会使用三元表达式，因为应用场景受限，不能写入复杂的逻辑
  - 基于 SFINAE 引入分支
    - 基于 std::enable_if 引入分支：语法不易懂但功能强大
      - 注意用做缺省模板实参不能引入分支!

      匹配失败并非错误，不仅可以使用在函数模板（支持函数重载）中，还可以使用在类模板中（类模板不支持函数重载，但支持偏特化）。
    - 基于 std::void_t 引入分支： C++17 中的新方法，通过“无效语句”触发分支
  - 基于 concept 引入分支： C++20 中的方法
    - 可用于替换 enable_if
  - 基于三元运算符引入分支： std::conditional 的数值版本

```
constexpr int fun(int x) {//可以在编译期执行，
也可以在运行期执行
  if (x > 3) {
    return x * 2 ;
  }
  else {
    return x - 100;
  }
}

constexpr int x = fun(100);
//这种方法引入的编译器分支应用范围
非常小。


基于if constexpr 分支，(由于函数没有定义constexptr)因此
这个函数只能在运行期执行，参数是以模板参数的形式提供，
而不是函数参数的形式。
虽然这个函数是在运行期执行的，但是参数是在编译期获得的
因为这个参数是个模板的形参，模板的形参必须在编译期获得
template <int x>//x作为了模板参数
int fun() {
  if constexpr (x>3) {//编译期分支，我们在运行期的函数中
引入了一个编译期分支，
    return x * 2;
  }
  else {
    return x - 100;
  }
}
int main() {
  int y = fun<100>();//这个调用的过程是在运行期执行的，

}
```

```
//引入特化
template<int x>
struct Imp {
  constexpr static int value = x * 2;
  using type = int;
};

//引入特化
template <>
struct Imp<100>
{
  constexpr static int value = 100 - 3;
  using type = double;
};
constexpr int x = Imp<97>::value;//选择普通分支
constexpr int x = Imp<100>::value;//选择100分支
using x = Imp<100>::type; // x是double

//实现分支功能
template <int x>
struct Imp;
template <int x>
  requires (x < 100)
struct Imp<x> //因为是特化，所以这里要有<x> {
  constexpr static int value = x * 2;
};
template <int x>
  requires (x >= 100)
struct Imp<x> {
  constexpr static int value = x-3;
};
constexpr auto x = Imp<100>::value;
int main() {
  std::cout << x << std::endl;
}
```

```
基于 std::enable_if 引入分支
template<int x, std::enable_if_t<
(x<100)>* = nullptr>//*void 我们
为什么要在这里定义void*呢，我们在
这里作为了一个模板形参类型，模板
形参类型是void*,模板形参的缺省值
是nullptr.
constexpr auto fun() {
  return x * 2;
}
template <int x, std::
enable_if_t<(x >=100)* = nullptr>
constexpr auto fun() {
  return x - 3;
}
constexpr auto x = fun<99>();

template<int x, typename = void*>
struct Imp;
template <int x>
struct Imp<x, std::enable_if_t<
(x<100)>*> {
  constexpr static int value = x
* 2;
  using type = int;
};
template <int x>
struct Imp<x, std::enable_if_t<
( x>100)>*> {
  constexpr static int value = x
3;
  using type = int;
};
```

```
template<int x>
constexpr auto fun = (x<1000) ? x * 2 : x - 3;

constexpr auto x = fun<102>;

int main() {
  x;
}
```

顺序、分支、循环代码的编写方式

- 循环代码的编写方式 通常来说会使用分支的方法来完成循环
  - 简单示例：计算二进制中包含 1 的个数
  - 使用递归实现循环
  - 任何一种分支代码的编写方式都对应相应的循环代码编写方式
  - 使用循环处理数组：获取数组中 id=0,2,4,6... 的元素
  - 相对复杂的示例：获取数组中最后三个元素 有个改进版本的例子没有记下来

```
//接左下
template <typename... Processed, typename T1>//如果待处理的数
组只有一个元素，选择这个逻辑
struct Imp<Cont<Processed...>, Cont<T1>> {
  using type = Cont<Processed..., T1>;
};
template <typename... Processed>
struct Imp<Cont<Processed...>, Cont<>> {//待处理数组为0
  using type = Cont<Processed...>;
};
using outptu = Imp<Cont<>, Input>::type;//表示最开始的时候，
没有任何一个元素在结果列表里，待处理所有的元素都在Input中
int main () {
  cout << std::is_same_v<Output, Cont<int, double, void>>;
```

```
//使用循环处理数组,获取数组中id=0,2,4,6的元素
template <typename...> class Cont;//数组，这个东西更像是一个容器，
其中可以包含很多很多类型。
using Input = Cont<int, char, double, bool. void>;
template <typename Res, typename Rem>//可变长度的序列
struct Imp;
};//定义了一个类模板，这里面接受了一个type,就是返回Res,这个
类模板接受的参数是不定长的。
//引入特化
template <typename... Processed, typename T1, typename T2, typename... TRemain>//待处理
数组包含大于等于2个数组
struct Imp<Cont<Processed...>, Cont<T1, T2, TRemain...>> {
  using type1 = Cont<Processed..., T1>;
  using type = typename Imp<type1, Cont<TRemain...>>::type; //为了表示type是一个
类型，因此需要加入typename
};//循环的主体逻辑，1.把t1扔到已经处理的类型中(Proccessed),2. 递归调用Imp,但是把type1
放到前面，把T1,T2扔掉
```

//循环代码

```
template <int x>//普通版本
constexpr auto fun =  (x%2) + fun<x / 2>;

template <>//特化版本,通过特化版本来结束
constexpr auto fun<0> = 0;

constexpr auto x = fun<99>;
int main() {
  x;
}
```

```
fun<(1100011b)>
1 + fun<(110001b)>
1 + 1 + fun<(11000b)>
1 + 1 + 0 + fun<(1100b)>
1 + 1 + 0 + 0 + fun<(110b)>
1 + 1 + 0 + 0 + 0 + fun<(11b)>
1 + 1 + 0 + 0 + 0 + 1 + fun<(1b)>
1 + 1 + 0 + 0 + 0 + 1 + 1 + fun<0>
1 + 1 + 0 + 0 + 0 + 1 + 1 + 0
```

//获取最后三个元素

```
template <typename...> class Cont;
using Input = Cont<int, char, double, bool, void>;

template <typename Res, typename Rem>//这是一个类模板的声明，这个声明表示这是个元函数，可以接受两个参数，第一个参数表示已经处理完的部分，第二个参数表示待处理的部分。
struct Imp;//接下来5个特化来实现了整个逻辑。

template <typenam U1, typename U2, typename U3, typename T, typename... TRemain>
struct Imp<Cont<U1,U2,U3>, Cont<T, TRemain...>>{//已经处理完的特化里已经包含了3个元素了，同时待处理的部分不为空。4. 前面有三个元素的时候，他会把U1扔掉，把U2,U3,T拿出来，然后这个函数会被反复调用，直到剩余部分为空。
  using type1 = Cont<U2, U3, T>;
  using type = typename Imp<type1, Cont<TRemain...>>::tyoe;
};
template <typenam U1, typename U2, typename T, typename... TRemain>
struct Imp<Cont<U1,U2>, Cont<T, TRemain...>>{//已处理的部分已经包含了2个元素，待处理的部分不为空
  using type1 = Cont<U1, U2,T>;
  using type = typename Imp<type1, Cont<TRemain...>>::tyoe;
};
template <typenam U1, typename T, typename... TRemain>
struct Imp<Cont<U1>, Cont<T, TRemain...>>{//2.匹配这个版本，把第2个元素拿出来，和第一个元素放到一起
  using type1 = Cont<U1,T>;
  using type = typename Imp<type1, Cont<TRemain...>>::tyoe;
};
template <typename T, typename... TRemain>
struct Imp<Cont<>, Cont<T, TRemain...>>{//1.这个版本会先被匹配,我把第一个元素放到type1中
  using type1 = Cont<U,T>;
  using type = typename Imp<type1, Cont<TRemain...>>::tyoe;
};
template <typename... TProcessed>
struct Imp<Cont<TProcessed...>, Cont<>> {//待处理部分为空，如果剩余部分为空，系统会选择这个版本。
  using type = Cont<TProcessed...>;
};
using Output = Imp<Cont<>, Input>::type;
int main() {
  std::cout << std::is_same_v<Output, Cont<double, bool, void>> << std::endl;
```

减少实例化技巧

- 为什么要减少实例化
  - 提升编译速度，减少编译所需内存
- 相关技巧
  - 提取重复逻辑以减少实例个数
  - conditional 使用时避免实例化
  - 使用 std::conjunction / std::disjunction 引入短路逻辑
- 其它技巧介绍
  - 减少分摊复杂度的数组元素访问操作

```
template <typename T>
constexpr bool intORdouble = std::is_same_v<T, int> || std::is_same_v<T, doule>;
//这里我们是不能保证使用短路逻辑的

template <typename T>
constexpr bool intORdouble = std::disjunction<std::is_same_v<T,int>, std::
is_same_v<T,double>>;
```

右边的代码会产生的实例：

```
using res = At<Cont<double, int, char, bool>, 2>::type;

At<Cont<double, int, char, bool>, 2>
At<Cont<int, char, bool>, 1>
At<Cont<char, bool>, 0>

using res2 = At<Cont<double, int, char, bool>, 3>::type;
At<Cont<double, int, char, bool>, 3>
At<Cont<int, char, bool>, 2>
At<Cont<char, bool>, 1>
At<Cont<bool>, 0>
```

```
template <typename... T> class Cont;

template<typename T, unsigned id>
struct At;
template <typename TCur, typename...T, unsigned id>
struct At<Cont<TCur, T...>, id> {
  using type = typename At<Cont<T...>,id-1>::type;
};
template<typename TCur, typename...T>
struct At<Cont<TCur, T...>, 0> {
  using type = TCur;
};
using res = At<Cont<double, int, char, bool>, 2>::type; //我希望编写一个程
序，想获得数组的第二个元素
//这是一个非常经典的循环模式。
会产生的实例: At<Cont<double, int, char, bool>,2>  At<Cont<int,char, bool>,1>
At<Cont<char, bool>,0>
```

```
conditional 使用时避免实例化
using Res = std::conditional_t<false,
            std::remove_reference_t<int&>,
            std::remove_reference_t<double&>>;
//上面这个语句对应的是
std::conditional<false,
            std::remove_reference_t<int&>, //对应int
            std::remove_reference_t<double&>>::type;//对应double
//最后的结果就是double
//但是这个程序会引入更多的实例化。他会生成remove_reference_t<int&>,和
remove_reference_t<double>两个实例化。
但是，因为我们输入的是false,我们其实不需要对int做实例化
std::remove_reference_t<int&>等价于 std::remove_reference_t<int&>::type

using Res = std::conditional_t<false,
                               std::remove_reference<int&>,
                               std::remove_reference<double&>>::type;
//这样就不会对中间那条语句进行实例化了
```

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening