CS100 Recitation 14

GKxx

Contents

- 模板（总结、补充）
  - 基础知识
  - 可变参数模板
  - 一些由模板编译引发的问题
  - 实现一个 `std::distance`
  - 认识模板元编程

基础知识

模板

有模板函数、类模板、变量模板、别名模板。

```
template <typename T>
const T &max(const T &a, const T &b) {
  return a < b ? b : a;
}

template <typename T> class vector { /* ... */ };

template <typename T>
inline constexpr bool is_abstract_v = /* ... */;

template <typename T>
using MyPair = std::pair<T, T>; // MyPair<T> is an alias of std::pair<T, T>,
                                // for any type T.
```

模板不是 ... ，而是 ...

模板函数不是真正的函数，它只是提供给编译器的一份指导方案，编译器将根据该"方案"根据需要合成真正的函数。

当编译器看到一个模板函数时：

- 它会对这个函数做基本的语法检查，缺少分号、自己发明运算符等错误会被发现。
- 它会对这个函数中**与模板参数无关**的部分做一些细致的语法、语义分析。
- 除此之外，什么都不做，也不会生成任何的代码。

当这个模板函数被**实例化**时（伴随着相应的模板参数被提供）：

- 编译器将模板实参代入模板函数中，得到一个真正的函数。
- 编译器对这个真正的函数做彻底的检查和分析，并生成相应的代码。

https://godbolt.org/z/b7TP5v8ca

模板不是 … ，而是 …

模板函数不是真正的函数，它只是提供给编译器的一份指导方案，编译器将根据该“方案”根据需要合成真正的函数。

类模板、变量模板、别名模板也是类似。

对于类模板，未被用到的成员函数也不会被实例化，哪怕这个类被实例化了。

```
template <typename T> struct A {
  T x;
  void foo() { ++x; } // 不是什么类型都支持 ++ 的
};
A<std::string> a; // T = std::string 显然不支持 ++ ,
                  // 但是没关系，我们并未调用 a.foo()
```

模板实例化 (instantiation)

比较常见的是**隐式实例化**：当某个实例被需要时，它会被自动实例化出来。

- 比如，模板函数的某个实例被调用时，创建了类模板的某个实例的对象时。

```
template <typename T> void f(T x);
f(0); // 隐式实例化 f<int>(int)
```

也可以**显式实例化**：可能是我预判到这个实例将被用到，所以先要求编译器把它实例化出来。

```
template void f<double>(double); // 显式实例化 f<double>(double)
template void f<>(char);         // 显式实例化 f<char>(char) ，模板参数被推断
template void f(int);            // 显式实例化 f<int>(int) ，模板参数被推断
```

模板是编译时的游戏

模板实例化完全发生在**编译时**。左边这个 C++ 函数与右边的 Python （动态类型）函数有**根本的区别**。

```cpp
template <typename T>
T add(T a, T b) {
  return a + b;
}
```

```python
def add(a, b):
  return a + b
```

所有模板参数（在尖括号 `<>` 之间的内容）必须在**编译时已知**。

可变参数模板

可变参数模板

```
template <typename First, typename... Rest> // Rest 是一个模板参数包
void read(First &first, Rest &...rest) {    // rest 是一个函数参数包
  std::cin >> first;
  if (/* rest 不是空的 */)
    read(rest...); // 包展开
}
int i; double d; std::string s;
read(i); // First = int, Rest 和 rest 都是空的
```

`read(i, d, s)` 实例化出以下函数：

- `void read(int &first, double &rest_1, std::string &rest_2)`
- `read(rest...)` 又会调用 `read(rest_1, rest_2)`，导致 `void read(double &first, std::string &rest_1)` 被实例化，而它的 `read(rest_1)` 会导致 `void read(std::string &first)` 被实例化。

`sizeof...(pack)`

一个参数包里有几个参数？用 `sizeof...` 运算符。这个运算符在编译时求值。

*** 务必区分声明参数包时的 `...`、包展开时的 `...` 和 `sizeof...` 中的 `...`！**

```
template <typename First, typename... Rest> // Rest 是一个模板参数包
void read(First &first, Rest &...rest) {    // rest 是一个函数参数包
  std::cin >> first;
  if (sizeof...(Rest) > 0)
    read(rest...); // 包展开
}
```

报了个编译错误？它说我试图调用 `read()`

```
a.cpp: In instantiation of ‘void read(First&, Rest& ...) [with First = int; Rest = {}]’:
a.cpp:12:7:   required from here
a.cpp:7:9: error: no matching function for call to ‘read()’
    7 |     read(rest...); // 包展开
```

`if constexpr`

不妨试着写出当 `Rest = {}` 时的函数长什么样：

```
template <typename First>
void read(First &first) {
  std::cin >> first;
  if (false) // sizeof...(Rest) == 0
    read();  // Ooops! read 接受至少一个参数！
}
```

问题出在**这个 `if` 是运行时的控制流**，哪怕这个条件 100% 是 `false`，这个部分也必须能编译才行！

`if constexpr`

`if constexpr (condition)`：**编译时的** `if` (since C++17)

- `condition` 必须能在编译时求值
- 只有在 `condition` 为 `true` 时，`statements` 才会被编译。

```
if constexpr (condition)
  statements
```

- 根据 `condition` 的值来决定编译 `statementsTrue` 还是 `statementsFalse`。

```
if constexpr (condition)
  statementsTrue
else
  statementsFalse
```

`if constexpr`

```
template <typename First, typename... Rest>
void read(First &first, Rest &...rest) {
  std::cin >> first;
  if constexpr (sizeof...(Rest) > 0)
    read(rest...);
}
```

如果没有 `if constexpr`，我们就需要通过重载来完成：（这里的重载决议不用搞清楚）

```
template <typename T> // 为一个参数的情况单独定义
void read(T &x) { std::cin >> x; }
template <typename First, typename... Rest>
void read(First &first, Rest &...rest) {
  read(first); read(rest...);
}
```

一些由模板编译引发的问题

实现 `Dynarray<T>` 的 `operator<`

不就是给各个地方都加上 `<T>` 么，我会！

```
template <typename T>
class Dynarray {
  friend bool operator<(const Dynarray<T> &, const Dynarray<T> &);
};
template <typename T> // 别忘了模板声明
bool operator<(const Dynarray<T> &, const Dynarray<T> &) {
  // 实现这个函数...
}

Dynarray<int> a, b;
if (a < b) // ld 报错 undefined reference to operator< ???
  // ...
```

非模板的情形

```
class Dynarray {
  friend bool operator<(const Dynarray &, const Dynarray &); // (1)
};
bool operator<(const Dynarray &, const Dynarray &) { // (2)
  // 实现这个函数...
}
```

毫无疑问，(1) 处的声明和 (2) 处的定义是同一个函数。

模板情形

```
template <typename T> class Dynarray {
  friend bool operator<(const Dynarray<T> &, const Dynarray<T> &); // (1)
};
template <typename T>
bool operator<(const Dynarray<T> &, const Dynarray<T> &) { /* ... */ } // (2)
```

- 当我们使用 `Dynarray<int>` 时，这个类被实例化出来。
- 这时它顺带声明了 (1) `friend bool operator<(const Dynarray<int> &, const Dynarray<int> &)`，这个函数**不是模板**，而 (2) 是一个函数模板，**编译器不认为 (1) 和 (2) 是同一个函数**。
- 表达式 `a < b` 对于 `operator<` 做重载决议时，既能找到 (1) 又能找到 (2)，但是其它条件相同的情况下**非模板优于模板**，所以它选择了 (1) 而不是 (2)。
- 接下来压力给到链接器：(1) 只有声明而没有定义，遂报错。

解决方案 1：不用 friend

```
template <typename T>
class Dynarray {
  // 不声明 operator< 为 friend
};
template <typename T>
bool operator<(const Dynarray<T> &, const Dynarray<T> &) {
  // 不访问 Dynarray<T> 的私有成员
}
```

如果真的可以不借助 `friend` 实现它（而且不产生额外的代价），这当然也是个办法。

这时 `a < b` 的 `operator<` 正对应了这个模板函数，能够正确编译和链接。

这里只有一个 `operator<`，不存在两个 `operator<` 争宠的情况。

解决方案 2：在声明 `friend` 的同时定义它

```
template <typename T>
class Dynarray {
  friend bool operator<(const Dynarray<T> &, const Dynarray<T> &) {
    // 直接在这里实现它
  }
};
```

这里也只有一个 `operator<` 。

解决方案 3：告诉编译器“真相”

```
template <typename T> class Dynarray {
  friend bool operator<(const Dynarray<T> &, const Dynarray<T> &); // (1)
};
template <typename U> // 为了不引起混淆，这里用 U
bool operator<(const Dynarray<U> &, const Dynarray<U> &) { /* ... */ } // (2)
```

编译器认为：(1) 不是模板，(2) 是模板，(1) 和 (2) 不是同一个函数。

真相：(1) 和 (2) 应当是同一个函数。在 `T` 给定的情况下，(1) 其实是 (2) 的 `U = T` 情形的实例。

解决方案 3：告诉编译器“真相”

```
// 1. 在 `friend` 声明之前声明这个模板函数
//    为了声明 operator< 的参数，还得再为 class Dynarray 补充一个声明
template <typename T> class Dynarray;
template <typename T>
bool operator<(const Dynarray<T> &, const Dynarray<T> &);
template <typename T> class Dynarray {
  // 2. 声明 friend 时在函数名后面加上 <>（或 <T>）
  //    说明它是先前声明过的一个模板函数的一个实例
  friend bool operator< <>(const Dynarray<T> &, const Dynarray<T> &);
};
// 3. 正常给出 `operator<` 的定义
template <typename T>
bool operator<(const Dynarray<T> &, const Dynarray<T> &) {
  // ...
}
```

实现一个 `std::distance`

`std::distance`

定义于 `<iterator>` 中。相关的函数还有 `std::advance`, `std::next` 等。

```
template <typename Iterator>
auto distance(Iterator first, Iterator last);
```

计算从 `first` 到 `last` 的"距离"：

- `Iterator` 至少得是 InputIterator。
- 对于 RandomAccessIterators，返回 `last - first`。
- 对于一般的 InputIterator，从 `first` 开始不断 `++`，直到碰到 `last` 为止。

返回值类型是什么？

通常情况下，一个迭代器应当具有一个类型别名成员 `difference_type`，表示两个迭代器的“距离”的类型。

这个类型通常是 `std::ptrdiff_t`（和指针相减的类型相同），但这并不一定。

为了写出最通用的 `distance`，我们应该使用这个 `difference_type`：

```
template <typename Iterator>
typename Iterator::difference_type distance(Iterator first, Iterator last);
```

开头的这个 `typename` 是啥？如果 `Iterator` 是个指针怎么办？一会儿再说…

如何知道迭代器的型别？

通常情况下，一个迭代器应当具有一个类型别名成员 `iterator_category`，它是以下五个类型之一的别名：

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag : input_iterator_tag {};
  struct bidirectional_iterator_tag : forward_iterator_tag {};
  struct random_access_iterator_tag : bidirectional_iterator_tag {};
}
```

Tag dispatch

将两种不同的实现写在两个函数里，**分别加上一个 tag 参数**

```
template <typename Iterator>
auto distance_impl(Iterator first, Iterator last,
                   std::random_access_iterator_tag); // (1)
template <typename Iterator>
auto distance_impl(Iterator first, Iterator last,
                   std::input_iterator_tag); // (2)

template <typename Iterator>
auto distance(Iterator first, Iterator last) {
  using category = typename Iterator::iterator_category;
  // 传一个 category 类型的对象作为第三个参数
  // 如果 category 是 std::random_access_iterator_tag，就会匹配 (1)，否则匹配 (2)
  return distance_impl(first, last, category{});
}
```

Tag dispatch

```
template <typename Iterator>
auto distance_impl(Iterator first, Iterator last,
                   std::random_access_iterator_tag) { // (1)
  return last - first;
}
template <typename Iterator>
auto distance_impl(Iterator first, Iterator last,
                   std::input_iterator_tag); { // (2)
  typename Iterator::difference_type result = 0;
  while (first != last) { ++first; ++result; }
  return result;
}
template <typename Iterator>
auto distance(Iterator first, Iterator last) {
  using category = typename Iterator::iterator_category;
  return distance_impl(first, last, category{});
}
```

指针怎么办？

以上实现依赖于 `Iterator::difference_type` 和 `Iterator::iterator_category`，如果 `Iterator` 根本不是类类型怎么办？

当然可以直接为指针做一个重载：

```
template <typename T>
auto distance(T *first, T *last) {
  return last - first;
}
```

但事实上有很多函数都面临这个问题，全都多加一份重载也太麻烦了。

Traits 技术

```cpp
template <typename Iterator> // 一般情况：Iterator 是一个类类型
struct Traits {
  using difference_type = typename Iterator::difference_type;
  using iterator_category = typename Iterator::iterator_category;
};
template <typename T> // 为指针做特化
struct Traits<T *> {
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::random_access_iterator_tag;
};
```

使用 `Traits<Iterator>::difference_type` 和 `Traits<Iterator>::iterator_category`，即可处理所有情况。

iterator_traits

上面的这个 `Traits` 正对应了标准库 `std::iterator_traits`。

```
namespace std {
  template <typename Iterator> // 一般情况：Iterator 是一个类类型
  struct iterator_traits {
    using value_type        = typename Iterator::value_type;
    using pointer           = typename Iterator::pointer;
    using reference         = typename Iterator::reference;
    using difference_type   = typename Iterator::difference_type;
    using iterator_category = typename Iterator::iterator_category;
  };
}
```

`iterator_traits`

上面的这个 `Traits` 正对应了标准库 `std::iterator_traits`。

```
namespace std {
  template <typename T> // 为指针做特化
  struct iterator_traits<T *> {
    using value_type        = std::remove_cv_t<T>; // 这是啥？
    using pointer           = T *;
    using reference         = T &;
    using difference_type   = std::ptrdiff_t;
    using iterator_category = std::random_access_iterator_tag;
  };
}
```

`std::remove_cv_t<T>`：是 `T` 去除可能的顶层 `const` 或 `volatile` 后的类型，定义于 `<type_traits>`。

用 `if constexpr` 实现

能不能直接这样写？

```
template <typename Iterator>
auto distance(Iterator first, Iterator last) {
  using category = typename std::iterator_traits<Iterator>::iterator_category;
  if constexpr (/* category == std::random_access_iterator_tag */)
    return last - first;
  else {
    typename std::iterator_traits<Iterator>::difference_type result = 0;
    while (first != last) {
      ++first; ++result;
    }
    return result;
  }
}
```

如何判断两个类型相同？

用 `if constexpr` 实现

`std::is_same_v<T, U>`：`bool` 类型的编译期常量，当 `T` 和 `U` 是同一个类型时为 `true`，否则为 `false`。定义于 `<type_traits>`。

```
template <typename Iterator>
auto distance(Iterator first, Iterator last) {
  using category = typename std::iterator_traits<Iterator>::iterator_category;
  if constexpr (std::is_same_v<category, std::random_access_iterator_tag>)
    return last - first;
  else {
    typename std::iterator_traits<Iterator>::difference_type result = 0;
    while (first != last) {
      ++first; ++result;
    }
    return result;
  }
}
```

认识模板元编程 (Template Metaprogramming)

Hello world

```cpp
template <unsigned N>
struct Factorial {
  static const auto result = N * Factorial<N - 1>::result;
};
template <>
struct Factorial<0u> {
  static const auto result = 1u;
};
int main() {
  const auto n = Factorial<10>::result; // 3628800
  int a[n]; // 正确。n 是一个编译时常量，可以用来开数组。
}
```

虽然在 modern C++ 它可以完全被 `constexpr` 函数替代，但是这仍是 TMP 的最经典的 hello world。

`std::tuple`：一个编译期容器

可能的实现：

```
template <typename First>
class tuple<First>;
template <typename First, typename... Rest>
class tuple<First, Rest...> : public tuple<Rest...>;
// 例：tuple<A, B, C> 继承自 tuple<B, C>
```

```
std::tuple<int, double, int> t{0, 4.0, 42};
```

`std::ratio`：编译期有理数类

标准库 `<chrono>` 利用 `std::ratio` 来表示各种时间单位，并且保证了量纲的正确性

- 例如，微秒和毫秒不能在数值上直接相加。

一个经典的例子：将七大基本物理单位对应于七个模板参数

```
template <int mass, int length, int time, int charge,
          int temperature, int intensity, int amount_of_substance>
struct quantity;
```

- 例如，力（牛顿）就是 `quantity<1, 1, -2, 0, 0, 0, 0>`，即 $\mathrm{kg} \cdot \mathrm{m}/\mathrm{s}^2$。
- 正确定义 `quantity` 之间的运算，就可以在**编译时**杜绝量纲错误。

更疯狂的例子（不止局限于模板元编程）

C++ 的编译时计算能力非常强大 (turing-complete) 。

在 `r14` 文件夹下有

- `merge_sort`: 编译时归并排序
- `ctjson`: 编译时 json parser （需要 C++20）

更疯狂的：

编译时正则表达式

编译时 raytracer

其它可能的应用

- 表达式模板 expression templates
- 序列化 serialization
- Embedded Domain Specific Language
- ...

事实上很多使用模板的程序都需要一些 TMP 技术，哪怕只是非常简单的 specialization 或 SFINAE。