Modern cpp 30 lectures — template substitution failure

Tuesday, December 22, 2020 10:21 AM

函数模板的重载决议.

替换失败非错: substitution failure is not an error. (SFINAE)

当一个函数名称和某个函数模板名称相匹配时, 重载决议过程大致如下.

- 根据名称找出所有适用的函数和函数模板.
- 对于适用的函数模板, 要根据实际情况对模板形参进行替换, 如果发生错误, 这个模板会被丢弃.
- 在上面两步生成的可行函数集中, 编译器会寻找一个最佳匹配, 产生对该函数的调用, 如果没找到, 或找到了多个匹配程度相当的, 则报错.

Eg.

```
#include <stdio.h>

struct Test {
   typedef int foo;
};

template <typename T>
void f(typename T::foo)
{
  puts("1");
}

template <typename T>
void f(T)
{
  puts("2");
}

int main()
{
  f<Test>(10);
  f<int>(10);
}
```

Output:

```
 1
 2
```

分析:

首先看 f<Test>(10);

- 有两个模板符合名字 f.
- 替换结果为 f(Test::foo) 和 f(Test),
- 使用参数 10 去匹配, 只有前者的参数可以匹配, 因而第一个模板被选择.

再看 f<int>(10):

- 还是两个模板符合名字 f
- 替换结果为 f(int::foo) 和 f(int), 显然前者不是个合法类型, 被抛弃.
- 使用参数 10 去匹配 f(int) 没有问题, 那就使用这个模板实例了,

这里体现的是 SFINAE 设计的最初用法, 如果模板实例化中发生了失败, 就此出错中止, 因为还可能有其它可用函数重载.

这里的失败仅指函数模板的原型声明, 即参数和返回值, 函数体内的失败不考虑.

编译期成员检测.

SFINAE 还可用作其它用途, E.g. 根据某个实例化的成功或失败, 在编译期检测类的特性. 下面这个模板, 就可检测一个类是否有 reserve, 参数类型为 size_t 的成员函数.

```
template <typename T>
struct has_reserve {
  struct good { char dummy; };
  struct bad { char dummy[2]; };
  template <class U,
            void (U::*)(size_t) >
   struct SFINAE {};
   template <class U>
   static good
   reserve(SFINAE<U, &U::reserve>*);
   template <class U>
   static bad reserve(...);
   static const bool value =
   sizeof(reserve<T>(nullptr))
    == sizeof(good);
};
```

- 首先定义了两个结构 good, bad,
- 然后定义了一个 SFINAE 模板, 第二个参数需要是第一个参数的成员函数指针. 并且参数类型是 size_t, 返回值是 void.
- 随后定义了一个要求 SFINAE* 类型的 reserve 成员函数模板, 返回值是 good, 再定义了一个对参数类型无要求的 reserve 成员函数模板, 返回值是 bad.
- 最后, 定义常整型布尔值 value, 结果是 true 还是 false, 取决于 nullptr 能不能和 SFINAE* 匹配成功, 而这又取决于模板参数 T 有没有返回类型是 void, 接受一个参数并且类型为 size_t 的成员函数 reserve.

SFINAE 模板技巧

enable_if

从 C++11 开始, 标准库里有了 enable_if 的模板, 可用来选择性地启用某个函数的重载.

假设一个函数, 用来往一个容器尾部追加元素, 原型是:

```
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size);
```

container 有没有 reserve 成员函数, 是对性能有影响的, 应该预留好内存空间, 以免产生不必要的对象移动和拷贝. 利用 enable_if 和上面的 has_reserve 模板:

```
template <typename C, typename T>
enable_if_t<has_reserve<C>::value,
            void>
append(C& container, T* ptr,
       size_t size)
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size; ++i) {
    container.push_back(ptr[i]);
  }
}


template <typename C, typename T>
enable_if_t<!has_reserve<C>::value,
            void>
append(C& container, T* ptr,
       size_t size)
{
  for (size_t i = 0; i < size; ++i) {
    container.push_back(ptr[i]);
  }
}
```

- 对于某个 type trait, 添加 _t 后缀等价于其 type 成员类型, 因此可用 enable_if_t 取得结果的类型.
- enable_if_t<has_reserve<C>::value, void> 可理解为: 如果类型 C 有 reserve 成员, 就启用下面的成员函数, 返回类型为 void.

decltype 返回值:

如果想限制只有具有 reserve 成员函数的类可以使用这个重载:

```
template <typename C, typename T>
auto append(C& container, T* ptr,
            size_t size)
  -> decltype(
   declval<C&>().reserve(1U),
   void())
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size; ++i) {
    container.push_back(ptr[i]);
  }
}
```

declval: 这个模板用来声明一个某个类型的参数, 但这个参数只是用来参加模板的匹配, 不允许实际使用. 可以在某类型没有默认构造函数时, 假想一个该类的对象进行类型推导.

declval<C&>().reserve(1U) 用来测试 C& 类型的对象是不是可以拿 1U 作为参数来调用 reserve 成员函数,

这种方式的主要用途是避免错误的重载.

void_t

C++17 引入的模板:

```
template <typename...>
using void_t = void;
```

这个类型模板会把任意类型映射到 void, 编译器会检查那个类型的有效性:

```
template <typename T,
          typename = void_t<>>
struct has_reserve : false_type {};


template <typename T>
struct has_reserve<
  T, void_t<decltype(
      declval<T&>().reserve(1U))>>
  : true_type {};
```

标签分发.

之前我们用了 true_type 和 false_type 来选择合适的重载, 叫标签分发 (tag dispatch).

append 也可以用标签分发实现:

```
template <typename C, typename T>
void _append(C& container, T* ptr,
             size_t size,
             true_type)
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size; ++i) {
    container.push_back(ptr[i]);
  }
}


template <typename C, typename T>
void _append(C& container, T* ptr,
              size_t size,
              false_type)
{
  for (size_t i = 0; i < size; i++) {
    container.push_back(ptr[i]);
  }
}


template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
  _append(
    container, ptr, size,
    integral_constant<
      bool,
      has_reserve<C>::value>{});
}
```

这种方法跟 enable_if 等价.

为什么不能像 python 一样, 直接写

```
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
   if (has_reserve<C>::value) {
     container.reserve(
       container.size() + size);
   }
   for (size_t i = 0; i < size; ++i) {
     container.push_back(ptr[i]);
   }
}
```

在 C 类型没有 reserve 成员函数的情况下, 编译是不能通过的, 因为 C++ 是静态类型的语言. 所有的函数, 名字必须在编译时被成功解析, 确定,

在动态类型的语言里, 只要语法没问题, 缺成员函数要执行到那一行上才会被发现.