第37章 模板参数优化

《现代C++语言核心特性解析》 谢丙堃

允许常量求值作为所有非类型模板实参

- 严格的非类型模板参数的实例化规则：
  1. 对于整型作为模板实参，必须是模板形参类型的经转换常量表达式。所谓经转换的常量表达式是指隐式转换到某类型的常量表达式，特点是隐式转换和常量表达式。
  2. 对于对象指针作为模板实参，必须是静态或者是有内部或者外部链接的完整对象。
  3. 对于函数指针作为模板实参，必须是有链接的函数指针。
  4. 对于左值引用的形参作为模板实参，必须也是有内部或者外部链接的。
  5. 而对于成员指针作为模板实参的情况，必须是静态成员。

允许常量求值作为所有非类型模板实参

- 满足内部和外部链接

```
template<const char *> struct Y {};
extern const char str1[] = "hello world"; // 外部链接
const char str2[] = "hello world"; // 内部链接

int main()
{
    Y<str1> y1;
    Y<str2> y2;
}
```

允许常量求值作为所有非类型模板实参

- 反面例子——C++17之前会导致编译错误

```
int v = 42;
constexpr int* foo() { return &v; }
template<const int*> struct X {};

int main()
{
    X<foo()> x;
}
```

允许常量求值作为所有非类型模板实参

- C++17新规则：非类型模板形参使用的实参可以是该模板形参类型的任何经转换常量表达式

```
template<const char *> struct Y {};
int main()
{
    static const char str[] = "hello world";
    Y<str> y;
}
```

允许常量求值作为所有非类型模板实参

- 以下对象作为非类型模板实参依旧会造成编译器报错：
  - 对象的非静态成员对象
  - 临时对象
  - 字符串字面量
  - typeid的结果
  - 预定义变量

允许局部和匿名类型作为模板实参

- 例子：

```
template <class T> class X { };
template <class T> void f(T t) { }
struct {} unnamed_obj;
int main()
{
    struct A { };
    enum { e1 };
    typedef struct {} B;
    B b;
    X<A>  x1;         // C++11编译成功，C++03编译失败
    X<A*> x2;         // C++11编译成功，C++03编译失败
    X<B>  x3;         // C++11编译成功，C++03编译失败
    f(e1);            // C++11编译成功，C++03编译失败
    f(unnamed_obj); // C++11编译成功，C++03编译失败
    f(b);             // C++11编译成功，C++03编译失败
}
```

允许函数模板的默认模板参数

- C++11标准允许在函数模板中使用默认的模板参数，且不会影响模板参数的推导，例如：

```
template<class T = double>
void foo(T t) {}

int main()
{
    foo(5);
}
```

允许函数模板的默认模板参数

- 无论是函数的默认参数还是类模板的默认模板参数，都必须保证是从右往左的定义默认值。
- 反例：

```
template<class T = double, class U, class R = double>
struct X {};

void foo(int a = 0, int b, double c = 1.0) {}
```

允许函数模板的默认模板参数

- 函数模板没有此限制

```
template<class T = double, class U, class R = double>
void foo(U u) {}

int main()
{
    foo(5);
}
```

函数模板添加到ADL查找规则

- 在C++20标准之前，ADL的查找规则是无法查找到带显式指定模板实参的函数模板的。

```
namespace N {
    struct A {};
    template <class T> int f(T) { return 1; }
}

int x = f<N::A>(N::A());
```

允许非类型模板形参中的字面量类类型

- 在C++20标准之前，类类型无法作为非类型模板形参。

```
struct A {};

template <A a>
struct B {};

A a;
B<a> b; // 编译失败
```

扩展的模板模板参数匹配规则

- C++17标准之前，模板模板形参只能精确匹配实参列表。

```
template <template <typename> class T, class U> void foo()
{
    T<U> n;
}
template <class, class = int> struct bar {};

int main()
{
    foo<bar, double>(); // C++17以前编译错误
}
```

扩展的模板模板参数匹配规则

- C++17中非类型模板形参可以使用auto，这也是必须扩展匹配规则的一个理由。

```
template <template <int> class T, int N> void foo()
{
    T<N> n;
}
template <auto> struct bar {};

int main()
{
    foo<bar, 5>();
}
```

扩展的模板模板参数匹配规则

- 在C++17标准中放宽了对模板模板参数的匹配规则，它要求模板模板形参至少和实参列表一样特化。
  - 例如：函数模板foo的模板形参template <typename> class T相较于实参template <class, class = int> struct bar更加特化。
  - 再例如：对于模板形参template <int> class T相较于template <auto> struct bar也更加特化。

感谢您的观看

欢迎关注