第34章 基础特性的其他优化

《现代C++语言核心特性解析》 谢丙堃

显式自定义类型转换运算符

- 自定义类型转换运算符的弊病

```
template<class T>
class SomeStorage {
public:
    SomeStorage() = default;
    SomeStorage(std::initializer_list<T> l) : data_(l) {};
    operator bool() const { return !data_.empty(); }
private:
    std::vector<T> data_;
};
int main() {
    SomeStorage<float> s1{ 1., 2., 3. };
    SomeStorage<int> s2{ 1, 2, 3 };
    std::cout << "s1 == s2 : " << (s1 == s2) << std::endl;
    std::cout << "s1 + s2  : " << (s1 + s2) << std::endl;
}
```

显式自定义类型转换运算符

- 使用explicit

```
template<class T>
class SomeStorage {
public:
    SomeStorage() = default;
    SomeStorage(std::initializer_list<T> l) : data_(l) {};
    explicit operator bool() const { return !data_.empty(); }
private:
    std::vector<T> data_;
};
SomeStorage<float> s1{ 1., 2., 3. };
SomeStorage<int> s2{ 1, 2, 3 };
std::cout << "s1 == s2 : " << (s1 == s2) << std::endl;   // 编译失败
std::cout << "s1 + s2  : " << (s1 + s2) << std::endl;    // 编译失败
std::cout << "s1 : " << static_cast<bool>(s1) << std::endl;
if (s1) { std::cout << "s1 is not empty" << std::endl; }
```

关于std::launder()

- C++中存在的一个问题

```
struct X { const int n; };
union U { X x; float f; };

U u{{ 1 }};
X *p = new (&u.x) X {2};
```

关于std::launder()

- 解决方案

```
assert(*std::launder(&u.x.n) == 2);
```

```
template<typename T>
constexpr T* launder(T* p) noexcept
{
    return p;
}
```

返回值优化

- 返回值优化是C++中的一种编译优化技术，它允许编译器将函数返回的对象直接构造到它们本来要存储的变量空间中而不产生临时对象。
  - RVO（Return Value Optimization）
  - NRVO（Named Return Value Optimization）

返回值优化

- 例子

```
class X {
public:
    X() { std::cout << "X ctor" << std::endl; }
    X(const X&x) {
    std::cout << "X copy ctor" << std::endl; }
    ~X() { std::cout << "X dtor" << std::endl; }
};
X make_x() {
    X x1;
    return x1;
}
int main() {
    X x2 = make_x();
}
```

输出结果:

```
X ctor
X dtor
```

返回值优化

- 使用-fno-elide-constructors关闭拷贝消除

```
class X {
public:
    X() { std::cout << "X ctor" << std::endl; }
    X(const X&x) {
    std::cout << "X copy ctor" << std::endl; }
    ~X() { std::cout << "X dtor" << std::endl; }
};
X make_x() {
    X x1;
    return x1;
}
int main() {
    X x2 = make_x();
}
```

输出结果:

```
X ctor
X copy ctor
X dtor
X copy ctor
X dtor
X dtor
```

返回值优化

- 优化失效的情况

```
X make_x()
{
    X x1, x2;
    if (std::time(nullptr) % 50 == 0) {
        return x1;
    }
    else {
        return x2;
    }
}
int main() {
    X x3 = make_x();
}
```

输出结果:

```
X ctor
X ctor
X copy ctor
X dtor
X dtor
X dtor
```

返回值优化

- C++14标准明确了对于常量表达式和常量初始化而言，编译器应该保证RVO，但是禁止NRVO。
- C++17标准中提到了确保拷贝消除的新特性：
  - 在传递临时对象或者从函数返回临时对象的情况下，编译器应该省略对象的拷贝和移动构造函数，即使这些拷贝和移动构造还是有一些额外的作用，最终直接将对象构造到目标的存储变量上，从而避免临时对象的产生。另外标准还强调，这里的拷贝和移动构造函数甚至可以是不存在或者不可访问的。

允许按值进行默认比较

- C++20允许按值比较运算符函数
  - 默认比较运算符函数可以是一个参数为const C&的非静态成员函数，或则是两个参数为const C&或C的友元函数。

```
struct C {
    int i;
    friend bool operator==(C, C) = default;
};
```

支持new表达式推导数组长度

- 数组声明时根据初始化元素个数推导数组长度的特性应该是一致的。

```
// 自动推导数组长度
int x[]{ 1, 2, 3 };
char s[]{ "hello world" };

// C++20之前new无法推导数组长度，会编译错误
int *x = new int[]{ 1, 2, 3 };
char *s = new char[]{ "hello world" };
```

允许数组转换为未知范围的数组

- C++20中确定长度数组可以转换为长度未知的数组。

```
void f(int(&)[]) {}
int arr[1];

int main()
{
    f(arr);
    int(&r)[] = arr;
}
```

在delete运算符函数中析构对象

- 对象的析构和内存销毁有固定的执行顺序

```
struct X {
    X() {}
    ~X() {
        std::cout << "call dtor" << std::endl;
    }
    void operator delete(void* ptr) {
        std::cout << "call delete" << std::endl;
        ::operator delete(ptr);
    }
};

X* x = new X;
delete x;
```

在delete运算符函数中析构对象

- C++20允许由程序员来控制析构的执行

```
void operator delete(X* ptr, std::destroying_delete_t)
{
    ptr->~X();
    std::cout << "call delete" << std::endl;
    ::operator delete(ptr);
}
```

输出结果:

```
call dtor
call delete
```

调用伪析构函数结束对象生命周期

- C++20标准完善了调用伪析构函数结束对象生命周期规则。
  - 伪析构函数的调用总是会结束对象的生命周期，即使对象是一个平凡类型。

修复const和默认拷贝构造函数不匹配造成无法编译的问题

- 没有调用就不会报错

```
struct MyType {
    MyType() = default;
    MyType(MyType&) {};
};

template <typename T>
struct Wrapper {
    Wrapper() = default;
    Wrapper(const Wrapper&) = default;
    T t;
};

Wrapper<MyType> var;
```

不推荐使用volatile的情况

- 因为volatile限定符现实意义的减少以及容易理解偏差，于是C++20标准在部分情况中不推荐volatile的使用，这些情况包括：
  - 不推荐算术类型的后缀++和--表达式以及前缀++和--表达式使用volatile限定符
  - 不推荐非类类型左操作数的赋值使用volatile限定符
  - 不推荐函数形参和返回类型使用volatile限定符
  - 不推荐结构化绑定使用volatile限定符

不推荐在下标表达式中使用逗号运算符

- 标准希望将array[x,y] 这种表达方式能用在矩阵，视图，几何实体，图形API中。（C++23标准已经支持多参数的数组下标运算符）

模块

- 主要用途是将大型工程中的代码拆分成独立的逻辑单元，以方便大型工程的代码管理。
  - VS2019的代码，使用了experimental implementation

```
// helloworld.ixx
export module helloworld;
import std.core;
export void hello() {
    std::cout << "Hello world!\n";
}

// modules_test.cpp
import helloworld;
int main()
{
    hello();
}
```

感谢您的观看

欢迎关注