20220603-元编程

1.学习内容

1.1 元编程

constexpr关键字

constexpr修饰的变量

constexpr修饰的函数

2.结果描述

1.学习内容

1.1 元编程

constexpr关键字

constexpr出现在函数的声明中，保证函数返回一个编译期常量的可能性（这个函数不是只能在编译期使用），即如果传进去的参数是编译期常量，那么这个函数就能够也返回一个编译期常量。

C++ 复制代码

```
constexpr unsigned fibonaci(unsigned i)
{
    return (i<=1u)?i:(fibonaci(i-1)+fibonaci(i-2));
}
```

多了一个constexpr，这个函数就可以在编译期和运行期同时起作用。如果函数的参数被编译器检测到为编译期常量，那么这个函数就可以自动在编译期运行。

```cpp
int main(int argc,char** argv)
{
    char int_value[fibonaci(6)]={};//正确，数组大小在编译期被强制计算
    std::cout<<sizeof(int_values)<<std::endl;//正确，sizeof函数参数在编译期被计算

    std::cout<<fibonaci(argc)<<std::endl;//在运行时计算，因为argc只有在运行时才能确定
    std::cout<<sizeof(std::array<char,fibonaci(argc)>)<<std::endl;//Error，模板参数要求在编译期确定fibonaci的值，但argc是运行时参数
}
```

constexpr修饰的变量

声明时可以加constexpr修饰的类称为literal type。所有拥有constexpr修饰的构造函数的类都是literal type，因为拥有此类构造函数的类的对象可以被constexpr函数初始化。
可以使用constexpr构造函数来创造类的编译期常量对象，鉴于它也有x的constexpr类型的getter函数，因此也可以在编译期使用这些函数来获取来获取它的成员值。

```cpp
class Point
{
    int x;
    int y;
public:
    constexpr Point(int x_,int y_):x(x_),y(y_){}
    constexpr int getX() const {return x;}
    constexpr int getY() const {return y;}
}

constexpr Point p(1,2);
constexpr int py=p.getY();
```

constexpr修饰的函数

并非所有函数都快可以被定义为编译期运算函数。函数体内不能有try块以及恶人和static和局部线程变量。并且在函数中只能条用其它constexpr函数，该函数不能有任何运行时才有的行为，比如抛出异常，使用new或delete等。

C++ 复制代码

```cpp
constexpr unsigned fibonaci(unsigned i)
{
    switch(i)
    {
        case 0: return 0;
        case 1: return 1;
        default: return fibonaci(i-1)+fibonaci(i-2);
    }
}
```

如果给函数加上constexpr关键字，并不是说我们就把这个函数绑死在编译期上。这个函数也应该能在运行期被复用。如果一次调用被认为是runtime的，那么这个函数返回的值也不再是编译期常量——它就被当作一个正常的函数来对待。
在编译期调用constexpr函数，所有运行时所做的检查，在编译期均不会处理。

C++ 复制代码

```cpp
constexpr unsigned fibonaci(unsigned i)
{
    switch(i)
    {
        case 0:return 0;
        case 1:return 1;
        default:
            {
                auto f1=fibonaci(i-1);
                auto f2=fibonaci(i-2);
                //手动进行越界检查，如果compile-time发现越界，引导函数进入throw语句，throw语句是典型的run-time语句。
                if(f1>std::numeric_limits<unsigned>::max()-f2)
                {
                    throw std::invalid_argument{"overflow detected"};
                }
            }
    }
}
```

上面的检查会始终起作用。如果在编译期传入一个过大的参数从而产生了整型溢出，那么函数就会走到抛出语句。又因为抛出异常这种行为是编译期所不允许的，所以编译时就会报错——这个函数调用并不是一个编译期运算表达式。

2.结果描述

今天算是把编译期常量、constexpr部分结束掉了，虽然还是有点似懂非懂。希望通过后面的学习能进一步加深理解。