CISC 3142
Programming Paradigms in C++

Ch28 – Metaprogramming

Abstraction Mechanisms: Elements of Metaprogramming

(*Stroustrup – The C++ Programming Language, 4th Ed*)

Introduction

- What is *metaprogramming*?
    - Programming that manipulates program entities (classes, functions, etc)
    - Writing programs that compute at compile time and generate programs
    - Variations of this idea are also called
        - *two-level programming*
        - *multilevel programming*
        - *generative programming*, and more commonly,
        - *template metaprogramming*
- Two main reasons:
    - *Improved type safety* (computes types, eliminates many explicit type conversions)
    - *Improved run-time performance* (computes values and select functions at compile time)
- Templates provide arithmetic, selection, and recursion, and they constitute a complete compiler-time functional programming language (Turing complete)

Generic Programming vs Template Metaprogramming

- Generic programming is primarily a design philosophy – a programming paradigm
- Metaprogramming is just programming with an emphasis on computation, often involving selection and iteration, which has the following 4 levels of implementation complexity
    1. No computation (just pass type/value arguments)
    2. Simple computation (on types and values) not using compile-time tests
    3. Computation using explicit compile-time test, e.g. a compile-time `if`
    4. Computation using compile-time iteration (in the form of recursion)
- Metaprogramming is both "meta" and programming
    - A metaprogram is a compile-time computation yielding types or functions to be used at run time
- Generic programming focuses on interface specification, whereas metaprogramming is programming, usually with types as the values
- Overdoing metaprogramming may lead to excessive compile time

Type Functions

- A *type function* is a function that
  - Either takes at least one type argument, or
  - Produces at least one type as a result
  - Example: `sizeof(T)`, it takes a type argument T and returns # of bytes
- Most type functions don’t look like conventional functions

```
if (is_polymorphic<int>::value) cout << "Big surprise!";
```

  - Here `is_polymorphic<T>` returns its result as a member called `value` in `bool`
- More examples:

```
enum class Axis : char { x, y, z }; // char is the underlying type for Axis (must be integral)
enum flags { off, x=1, y=x<<1, z=x<<2, t=x<<3 }; // unscoped enum
typename std::underlying_type<Axis>::type x; // x is a char
typename std::underlying_type<flags>::type y; // y is probably an int (§8.4.2)
```

Type Functions (cont')

- Type functions are compile-time functions
- They can only take arguments (types/values) that are known at compile time and produce results (types/values) usable at compile time

```
template<typename T, int N>
struct Array_type {
        using type = T;
        static constexpr int dim = N; // this is compile-time determined, enum is also commonly used
        // enum { dim = N}; // also resolved during compile-time
};
```

- While not a useful type function, this returns multiple types/values

```
using Array = Array_type<int,3>; // Array is an alias type
Array::type x; // x is an int (type returned by Array)
constexpr int s = Array::dim; // s is 3 (value returned by Array)
```

Type Function – A More Elaborate Example

- Type function can perform very general computations using types and values – the backbone of metaprogramming
- Example: allocate an object on the stack or heap depending on its size (notice different code is used based on type)

```
constexpr int on_stack_max = sizeof(std::string); // max size for stack (8~32)
template<typename T>
struct Obj_holder {
    using type = typename std::conditional<
        (sizeof(T)<=on_stack_max),
        Scoped<T>,   // first alternative (definition on right)
        On_heap<T> // second alternative
    >::type; // here the type is either Scoped<T> or On_heap<T>
};
void f() {
    typename Obj_holder<double>::type v1; // on the stack
    typename Obj_holder<array<double ,200>>::type v2; // the array goes
                                                        on the free store
    ∗v1 = 7.7; // Scoped provides pointer-like access (* and [])
    (*v2)[77] = 9.9; // On_heap provides pointer-like access (* and [])
}
```

```
template<typename T>
struct On_heap {
    On_heap() : p(new T) { }  // allocate
    ˜On_heap() { delete p; } // deallocate
    T& operator∗() { return ∗p; }
    T∗ operator−>() { return p; }
private:
    T∗ p; // pointer to object on the free store
};
template<typename T>
struct Scoped {
    T& operator∗() { return x; }
    T∗ operator−>() { return &x; }
private:
    T x; // the object
};
// Note: The statement p->m is interpreted as
        (p.operator->())->m
```

Type Predicates

- A type function that returns a Boolean value

```
template<typename T>
void copy(T∗ p, const T∗ q, int n) {
    if (std::is_pod<T>::value) // is_pod: is plain old data (contiguous sequence of bytes)
        memcpy(p, q, n); // use optimized memory copy
    else
        for (int i=0; i!=n; ++i)
            p[i] = q[i]; // copy individual values using copy assignment
}
```

- For all standard library type predicates, a function returning a bool is also defined:

```
template<typename T>
void copy(T∗ p, const T∗ q, int n) {
    if (is_pod<T>())
    // ...
}
```

Selecting a Function

- A type function can also be used to select a function (functor)

```
struct X { // write X
    void operator()(int x) { cout <<"X" << x << "!"; }
    // ...
};
struct Y { // write Y
    void operator()(int y) { cout <<"Y" << y << "!"; }
    // ...
};
void f() {
    Conditional<(sizeof(int)>4),X,Y>{}(7); // make an X or a Y with default ctor, then call it with argument 7
    using Z = Conditional<(Is_polymorphic<X>()),X,Y>;
    Z zz; //make an X (if X has defined a virtual function) or a Y (otherwise)
    zz(7); // call an X or a Y
}
```

- Note the outer parentheses of **(sizeof(int)>4)**, without them, the **>** before 4 will be interpreted as end of template argument list, which leads to compile error

Traits

- A *trait* is used to associate properties with a type
- For example, the properties of an iterator are defined by iterator_traits (from <iterator>)

```
template<typename Iterator>
struct iterator_traits {
    using difference_type = typename Iterator::difference_type; // generally int (difference between pointers)
    using value_type = typename Iterator::value_type; // sometimes we can't do Iterator::value_type directly
    using pointer = typename Iterator::pointer;       // e.g. we can't do (int*)::value_type, though int* is an iterator
    using reference = typename Iterator::reference;
    using iterator_category = typename Iterator::iterator_category; // forward, bidirectional, random, i/o etc
};
```

- This can be interpreted as
  - A type function with many results (can return many values)
  - A bundle of type functions

```
template<typename Iter>
Iter search(Iter p, Iter q, typename iterator_traits<Iter>::value_type val) { // search for val between [p, q)
    typename iterator_traits<Iter>::difference_type m = q−p; // get the range of the search
    // ...
}
```

Control Structures - Selection

- To do general computation at compile time, we need selection and recursion
- Selection (for selecting types, not values)
  - Conditional: a way to choose between **two** types (an alias for std::conditional)
  - Select: a way of choosing among **several** types
- Conditional (std::conditional is from <type_traits>)

```
template<bool C, typename T, typename F> // general template, simply choosing T
struct conditional {
        using type = T;
};
template<typename T, typename F> // partial specialization for false, which resolves ahead of the general version
struct conditional<false, T, F> {
        using type = F;
};
```

- Usage:

```
typename conditional<(std::is_polymorphic<T>::value), X, Y>::type z; // z is of type X, if T is polymorphic, Y otherwise
```

- Type alias and usage:

```
template<bool B, typename T, typename F>
using Conditional = typename std::conditional<B,T,F>::type; // this alias is just for the purpose of being able to omit "::type"
Conditional<(Is_polymorphic<T>()),X,Y> z;
```

Selecting Among Several Types

```
class Nil {}; // class version of NULL
template<int N, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
struct select; // return a type based on N's value, definition not given on purpose (resolved on specializations)
template<int N, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
using Select = typename select<N,T1,T2,T3,T4>::type; // type alias for easier use (omitting ::type)
// Specializations for 0-3: (they are partial specializations)
template<typename T1, typename T2, typename T3, typename T4>
struct select<0,T1,T2,T3,T4> { using type = T1; }; // specialize for N==0
template<typename T1, typename T2, typename T3, typename T4>
struct select<1,T1,T2,T3,T4> { using type = T2; }; // specialize for N==1
template<typename T1, typename T2, typename T3, typename T4>
struct select<2,T1,T2,T3,T4> { using type = T3; }; // specialize for N==2
template<typename T1, typename T2, typename T3, typename T4>
struct select<3,T1,T2,T3,T4> { using type = T4; }; // specialize for N==3
```

- Using out of bound value for int N will lead to compile error

```
Select<5,int,double ,char> x; // Error: general form of select (i.e., with arbitrary value of N) is not defined
// note you may want to rename "select" as it may clash with system header files on certain platforms
```

Iteration and Recursion

- Using a factorial function template (calculating a value at compile time)

```
template<int N>
constexpr int fac() { return N∗fac<N−1>(); }
template<> constexpr int fac<1>() { return 1; } // specialization
constexpr int x5 = fac<5>(); // x5 evaluated to be 120 at compile time
```

- Since we don't have true variables at compile time, recursion is used
- There is no checking condition of N==1, a specialization for N==1 is given (could use N==0 as well)
- The alternative way of achieving this without using templates

```
constexpr int fac(int i) {
        return (i<2)?1:i*fac(i–1);
}
constexpr int x6 = fac(6);
```

- Note: the non-template constexpr functions can be evaluated at compile-time (in the above example) or run-time (if you pass a variable unresolved by the compiler). The template (metaprogramming) version is for compile-time use only

Recursion Using Classes

- Iteration involving more complicated state or more elaborate parameterization can be handled using classes
- The same example of factorial program:

```
template<int N>
struct Fac {
    static constexpr int value = N∗Fac<N−1>::value; // static constexpr and enum all evaluated
};                                                 // at compile time, and available without
                                                   // an object being created (static)

template<>
struct Fac<1> {
    static constexpr int value = 1;
};

constexpr int x7 = Fac<7>::value;
```

When to Use Metaprogramming

- Why would you want to compute something at compile time?
- Only if when they yield code that’s
  - cleaner
  - better-performing
  - easier-to-maintain
- Yet code depending on complicated use of templates can be hard to read
- Nontrivial uses of templates can also impact compile times
- If one finds the need to write macros to hide “details” of template instantiation, metaprogramming might have gone a bit too far

Chapter-end Advice

[1] Use metaprogramming to improve type safety; §28.1.

[2] Use metaprogramming to improve performance by moving computation to compile time; §28.1.

[3] Avoid using metaprogramming to an extent where it significantly slows down compilation; §28.1.

[4] Think in terms of compile-time evaluation and type functions; §28.2.

[5] Use template aliases as the interfaces to type functions returning types; §28.2.1.

[6] Use **constexpr** functions as the interfaces to type functions returning (non-type) values; §28.2.2.

[7] Use traits to nonintrusively associate properties with types; §28.2.4.

[8] Use **Conditional** to choose between two types; §28.3.1.1.

[9] Use **Select** to choose among several alternative types; §28.3.1.3.

[10] Use recursion to express compile-time iteration; §28.3.2.

[11] Use metaprogramming for tasks that cannot be done well at run time; §28.3.3.

[12] Use **Enable_if** to selectively declare function templates; §28.4.

[13] Concepts are among the most useful predicates to use with **Enable_if**; §28.4.3.

[14] Use variadic templates when you need a function that takes a variable number of arguments of a variety of types; §28.6.

[15] Don’t use variadic templates for homogeneous argument lists (prefer initializer lists for that); §28.6.

[16] Use variadic templates and **std::move()** where forwarding is needed; §28.6.3.

[17] Use simple metaprogramming to implement efficient and elegant unit systems (for finegrained type checking); §28.7.

[18] Use user-defined literals to simplify the use of units; §28.7.