CISC 3142
Programming Paradigms in C++

Ch23-25 – Selected Topics

Abstraction Mechanisms: Elements of Generic Programming

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

Introduction

- Templates provide
  - direct support for generic programming
  - A way to represent a wide range of general concepts
- The mechanism allows a type or a value to be a parameter
  - In defining a class, a function, or a type alias
  - And they can match less general code in run-time and space efficiency
  - The argument types need not be part of an inheritance hierarchy (built-in types are acceptable and common)
  - Templates are type-safe, but unfortunately
  - A template’s requirements on arguments can’t be stated in code (updated in C++20)

A Simple String Template

- Making the character type a parameter

```
template<typename C>   // a template is being declared, a type argument C will be used in the declaration
class String {         // C is a type name, it doesn't have to be a class name, but String<C> is a class name
public:
        String();                                  // default constructor
        explicit String(const C∗);                 // parametric constructor
        String(const String&);                     // copy constructor
        String& operator=(const String&);          // copy assignment
        // ...
        C& operator[](int n) { return ptr[n]; }    // unchecked element access
        String& operator+=(C c);                   // add c at end - concatenation
        // ...
private:
        static const int short_max = 15;           // for the short string optimization
        int sz;
        C∗ ptr;                                    // ptr points to sz Cs
};
```

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;
```

The standard library, string is:

using string = std::basic_string<char>

A type alias

Defining a Template

- A class generated from a class template is a perfectly ordinary class
  - No run-time overhead associated with it
- The rules for templates apply equally to class and function templates
- When designing a template, it's best to have debugged a particular class, such as String, before turning it into a template, String<C>
  - Handles the more concrete case first before dealing with the more abstract one
  - Templates should be viewed as generalization of concrete examples, rather than being designed from first principles
- Members of class template can be defined outside the class

```
template<typename C>
String& String<C>::operator+=(C c) {
    // ... add c to the end of this string ...
    return ∗this;
}
```

Template Instantiation

```
String<char> cs; // cs is like a normal class after template instantiation
void f() {
    cs = "It's the implementation's job to figure out what code needs to be generated";
}
```

- *Instantiation*: generating a class/function from a template plus a template argument list
- A version of a template defined for a specific template argument list is called a *specialization* (note, it could have different behaviors, thus - specialization)
- Only **used** member functions of a template class will be generated by the compiler, so for the above example, it only includes
    - Default constructor, destructor, and copy assignment: `String<char>::operator=(const char*)`
- Generated classes and functions are ordinary ones that obey all usual rules

Type Checking

- Type checking is done on the code generated by template instantiation
  - The generated code may contain a lot of details the user (programmer) may not expect
- Would like to do this (Note: available since C++20)

```
template<Container Cont, typename Elem>
    requires Equal_comparable<Cont::value_type, Elem>() // requirements for types Cont and Elem
int find_index(Cont& c, Elem e); // find the index of e in c
```

- This kind of requirement needs predicate on template arguments – such as "C must be a container", where
  - Container<vector<int>>() returns true, but Container<int>() returns false
  - **Such a predicate is called a *concept***
- For the time-being, concepts are specified via comments like
  - Container<T>() will be true if
  - T must have a subscript operator [] (but what is the type of the index, and what does it return? Think map)
  - T must have a size() member function
  - T must have a member type value_type which is the type of its elements

Type Equivalence

```
using uchar = unsigned char;
String<uchar> s5; // the same as String<unsigned char>
```

- Types generated from a single template by different template arguments are different types
- Generated types from related arguments are not automatically related
- Assume Circle derives from Shape

```
Shape* p {new Circle({0, 0},100)}; // Circle* (center 0, 0, radius 100) converts to Shape*
vector<Shape>* q {new vector<Circle>{}}; // error : no vector<Circle>* to
                                                    vector<Shape>* conversion
vector<Shape> vs {vector<Circle>{}}; // error : no vector<Circle> to vector<Shape>
                                              conversion
vector<Shape*> vs {vector<Circle*>{}}; // error : no vector<Circle*> to vector<Shape*>
                                                conversion
```

Class Template Members Can Have

- Data Members

```
template<typename T>
struct X {
    int m1 = 7;
    T m2;
    X(const T& x) :m2{x} { }
};
X<int> xi {9};
X<string> xs {"Rapperswil"};
```

- Member Functions

```
template<typename T>
struct X {
   void mf1() { /* ... */ } // defined in-class
   void mf2();
};
template<typename T>
void X<T>::mf2() { /* ... */ } // defined outside
```

- Member Type Aliases

```
template<typename T>
class Vector {
public:
    using value_type = T;
    using iterator = Vector_iter<T>; // Vector_iter is defined elsewhere
    // ...
};
```

- Member Types

```
template<typename T>
struct X {
    enum E1 { a, b };
    enum class E3;
};
template<typename T>
enum class X<T>::E3 { a, b }; // defined outside
```

Function Templates

- While container classes are mostly class templates, function templates are commonly used for manipulating such containers

```
template<typename T> void sort(vector<T>&); // declaration, note it's different from std::sort
```

- They are essential for writing generic algorithms
- The function's arguments determine which version of template is used
- The above declaration implies operator < is used for comparison, but not every type has a < operator. A better version:

```
template<typename T, typename Compare = std::less<T>>
void sort(vector<T>& v) { … } // definition skipped
```

- Then

```
vector<int> vi = {1, 3, 2, 4, 5};
sort(vi); // sort using default "<" operator, sort(vector<int>&)
sort<int, std::greater<int>()>(vi); // sort(vector<int>&) using greater (in descending order)
```

Function Template Arguments Deduction

- Compilers can deduce type and non-type arguments from a call

```
template<typename T1, typename T2>
pair<T1,T2> makePair(T1 a, T2 b) {
        return {a,b};
}
auto x = makePair(1, 2); // x is a pair<int, int>
auto y = makePair(string("New York"), 7.7); // y is a pair<string, double>
```

- Note that class template parameters are never deduced
  - For example, you can't declare **vector<23> v**; when you meant to say **vector<int> v**;
- If a template argument can't be deduced from function arguments, it can be specified explicitly

```
template<typename T>
T∗ create();    //make a T and return a pointer to it
void f() {
        int∗ p = create<int>(); // function template argument T is explicitly specified as int
        int∗ q = create(); // error : can't deduce template argument
}
```

Function Template Overloading

- Function templates and ordinary functions with the same name, can be called correctly via overload resolution

```
template<typename T>
    T sqrt(T);                          // 1) general
template<typename T>
    complex<T> sqrt(complex<T>);  // 2) specific
double sqrt(double);                    // 3) specific
void f(complex<double> z) {
    sqrt(2);    // sqrt<int>(int) – 1)
    sqrt(2.0); // sqrt(double) – 3)
    sqrt(z);    // sqrt<double>(complex<double>) – 2)
}
```

- Another example (compiler doesn't prefer one resolution over the other, concerning conversions)

```
template<typename T>
    T max(T,T);
max('a',1); // error : ambiguous: max<char,char>() or max<int,int>()?
max(2.7,4); // error : ambiguous: max<double,double>() or max<int,int>()?
```

Generic Programming

- The most common use of templates is to support *generic programming*
  - Programming focused on the design/implementation/use of general algorithms
  - Template is C++'s main support for generic programming
  - Templates provide (compile-time) parametric polymorphism
- The two aspects of generic programming
  - *Lifting*: generalizing an algorithm to allow the greatest (reasonable) range of argument types
  - *Concepts*: precisely specifying the requirements of an algorithm (or a class) on its arguments

Algorithms and Lifting

- An *algorithm* is a procedure for solving a problem
  - A finite series of computation steps to produce a result
  - A function template is often called an algorithm
- The most effective way of getting a good algorithm is to generalize from concrete examples – such generalization is called *lifting*
- It's important that going from concrete to general maintains performance and readability
- Consider the following two concrete functions:

```
double add_all(double∗ array, int n) {
        double s {0};
        for (int i = 0; i<n; ++i) // check for end, next item
            s = s + array[i]; // accessing current value
        return s;
}
```

```
struct Node {
        Node∗ next;
        int data; };
int sum_elements(Node∗ first, Node∗ last) {
        int s = 0;
        while (first!=last) {  // check for end
            s += first−>data; // current value
            first = first−>next; // next item
        }
        return s;
}
```

Algorithms and Lifting (cont')

- Example of lifting

```
template<typename Iter, typename Val>
Val sum(Iter first, Iter last) {
        Val s = 0;
        while (first!=last) {
             s = s + *first;
             ++first;
        }
        return s;
}
```

- This is an algorithm that can be used for both arrays and linked lists, and for both ints and doubles:

```
double ad[] = {1, 2, 3, 4};
double s = sum<double*, double>(ad, ad+4);
list<int> lst = {10, 20, 30, 40};
int s2 = sum<list<int>::iterator, int>(lst.begin(),
        lst.end());
```

- Note: sum() is as efficient as the handcrafted ones we start from
- It can be generalized even further
  - In the above example, the typename Val can't be deduced from calling (not passed as function's arguments), so it has to be explicitly specified, which in turn requires the first argument to be entered
  - To avoid explicitly specifying type argument Val, it can be passed as a function parameter, which would also allow it to serve as the initial sum
  - Even the summation could be generalized away so any operations (+, ×, etc) could be supported
  - The improved version is shown on next slide

A more generalized sum()

```
template<typename Iter, typename Val, typename Oper>
Val accumulate(Iter first, Iter last, Val s, Oper op) {
	while (first!=last) {
		s = op(s,∗first);
		++first;
	}
	return s;
}
```

- We can now use argument op to combine element values

```
double ad[] = {1,2,3,4};
double s1 = accumulate(ad, ad+4, 0.0, std::plus<double>());          // as before, gets 10
double s2 = accumulate(ad, ad+4, 1.0, std::multiplies<double>()); // gets 24
```

- Notice that we are not using accumulate<double*, double, std::plus<double>>, because all of them can be deduced now

Type and value as arguments

- Use short names with initial uppercase letters as names of template type argument, e.g. T, C, Cont, and Ptr
  - Long names with all caps may easily clash with macros
- A template parameter is defined to be a *type parameter* by prefixing it with typename or class
- A template parameter that is not a type (or a template) is called a *value parameter* and an argument passed to it a *value argument* (**can be int or pointer, can't be a floating-point value**)

```
template<typename T, int max>
class Buffer {
        T v[max];
public:
        Buffer() { }
        // ...
};
Buffer<int, 5000> ibuf;
Buffer<Record, 8> rbuf;
```

```
template<typename T, char∗ label>
class X {
    // ... };
char lx2[] = "BMW323Ci";
X<int,lx2> x2;              // OK
int i=100;
Buffer<int, i> bx;          // error : constant expression expected for integer
constexpr int max = 200;
Buffer<int, max> bm; // OK: constant expression
```

Reusing a type parameter

- A type template parameter can be used as a type later in a template parameter list
- This becomes particularly useful when combined with a default template argument

```
template<typename T, T default_value = T{}>
class Vec {
    // ... Make use of default_value
};
Vec<int,42> c1; // c1 is a Vector with integer elements, their default_value is 42
Vec<int> c11; // default_value is int{}, that is, 0
Vec<string,"fortytwo"> c2;
Vec<string> c22; // default_value is string{}, that is, ""
```

Operations as arguments

```
template<typename Key, Class V, typename Compare = std::less<Key>>
class map {
public:
    map() { /* ... */ } // use the default comparison, i.e. std::less<Key>
    map(Compare c) : cmp{c} { /* ... */ } // pass a Compare to override the default
    // ...
    Compare cmp; // comparison functor (as a data member)
};
map<string, int> m1; // use the default comparison (less<string>)
map<string, int, std::greater<string>()> m2; // compare using greater<string>()
```

Specialization

- A template gives a single definition to be used for every template argument
- What if I want to have a special (different) behavior when a specific template argument is passed
- This can be addressed by providing alternative definitions of the template
- The compiler can choose between them based on the arguments provided where they are used
- Such alternative definitions of a template are termed *user-defined specializations*, or simply *user specializations*

Example of Specialization

- No specialization

```
template<typename T>
class Vector { // general vector type
    T* v;
    int sz;
public:
    Vector();
    explicit Vector(int);
    T& elem(int i) { return v[i]; }
    T& operator[](int i);
    void swap(Vector&);
// ...
};
Vector<int> vi;          // code replication
Vector<Shape*> vps; // for all types
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

- Specialization for pointer parameters (for polymorphism)

```
template<> // this is a specialization (complete)
class Vector<void*>{ // i.e. T replaced by a special case: void*
    void** p;
    // You can have alternative implementation of interface here
    void*& operator[](int i);
};
```

- This specialization can then be used as the common implementation for all Vectors of pointers, thus save the bloat of code replication
- Specialization is a way of specifying **alternative** implementations for different uses of a **common** interface

Order of Specialization

```
template<typename T>
    class Vector; // most general; the primary template, must be declared first
template<typename T>
    class Vector<T*>; //specialized for any pointer (partial specialization)
// ////////////////////////////////////////////////////////////////////////////////
// note: in this case Vector<int*>, not Vector<int>, is matched with Vector<T*>.    //
// A complete implementation of this partial specialization would derive from      //
// the complete specialization version below, and simply cast all void* to T*.     //
// Doing so still saves code replication. See book for details                     //
// ////////////////////////////////////////////////////////////////////////////////
template<>
    class Vector<void*>; // specialized for void* (complete specialization)
```

- The most specialized version will be preferred over the others in declarations of objects, pointers
- Partial specialization only applies to class templates.
- For function templates, specialization must be complete (no partial), that is, all type parameters must be instantiated. However, function templates allow overloading, i.e., different versions of the function with the *same* name but with different number and types of parameters

Function Template Specialization

(from Prata - C++ Primer Plus, 6th Ed)

```
// non template function prototype
1) void swap(Job & a, Job & b) {
       Job temp = a;
       a = b;
       b = temp;
   }

// template prototype
3) template <typename T>
   void swap(T & a, T & b) {
       T temp = a;
       a = b;
       b = temp;
   }
```

```
struct Job {
    char name[40];
    double salary;
    int floor;
};
```

```
// explicit specialization for the Job type
// we may want to swap only salary/floor
2) template <> void swap<Job>(Job &j1, Job &j2) {
       double t1 = j1.salary; // swap salary
       j1.salary = j2.salary;
       j2.salary = t1;
       int t2 = j1.floor; // swap floor
       j1.floor = j2.floor;
       j2.floor = t2;
   }
```

- Order of specialization
  1) Non-template function (most special)
  2) Explicit specialization
  3) Regular template (most general)

Function Template Specialization (cont')

(from Prata - C++ Primer Plus, 6th Ed)

- Function 1 will take precedence over function 2 if both are present

```
Job sue = {"Susan Yaffee", 73000.60, 7};
Job sidney = {"Sidney Taffee", 68060.72, 9};
swap(sue, sidney);
```

- If Function 1 (non-template) is used, and we print out sue and sidney:

  Sidney Taffee: $68060.72 on floor 9 // this is sue (not likely what you wanted)

  Susan Yaffee: $73000.60 on floor 7 // this is sidney

- If Function 2 (specialization) is used:

  Susan Yaffee: $68060.72 on floor 9 // this is sue (this is likely what you wanted)

  Sidney Taffee: $73000.60 on floor 7 // this is sidney