Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 12 Templates

- C++ **template（模板）** is based upon **the concept of type variable**, i.e. a variable that takes a **type** as its value **（以类型作为变量）**
  - General form:

    ```
    template <class/typename T1, class/typename T2, int i, ...>
    return_type function_name(T1 value1, T2 value2, int i, ...){
        ...
    }
    ```

    - Can also use normal function-style parameters (like `int i`) to specify nontype parameters.
  - Example:

    ```
    template <class T> class vector {
      T *v;
      int size;

    public:
      vector(int m) {
        v = new T[size = m];
        for (int i = 0; i < size; i++)
          v[i] = 0;
      }
      T operator*(vector &y) {
        T sum = 0;
        for (int i = 0; i < size; i++)
          sum += this->v[i] * y.v[i];
        return sum;
      }
    };

    int main(){
      vector<int> v1(10);
      vector<double> v2(5);
      return 0;
    }
    ```

- Kinds of Templates
  - **Class template（类模板）**: Permit the development of **generic objects（通用对象）**
  - **Function template（函数模板）**: Permit the development of **generic algorithms（通用算法）**
  - Variable template: C++ new standard
- **Class Template（类模板）**
  - Consists of a template header followed by a normal class definition
  - Format:

```
template <class T>
class Class_name{
    ...
}
```

- Need not use "`T`", any identifier will work **（这个`T`不唯一，想把它换成任何不是关键字作为类型变量名都可以）**
- To create an object of the class, type:

```
Class_name<type> my_object;
```

- Class **template member functions（模板成员函数）**
  - If the member functions are defined **inside the class**, the member functions is defined normally
  - If the member functions are defined **outside the class**, the member functions must be defined by the **function templates（如果成员函数定义在类外，那么必须以函数模板的形式进行定义）**：

    ```
    template <class T>
    return_type Class_name<T>::function_name(arglist) {
      ...
    }
    ```

  - Example:

    ```
    template <class T>
    class Vector{
      T *v;
      int size;
    public:
      Vector(int m);

      T operator*(Vector &y){ // inside the class
          T sum = 0;
          for (int i = 0; i < size; i++){
            sum += this->v[i]*y.v[i];
          }
          return sum;
      }
    };

    template <class T> // outside the class
    Vector<T>::Vector(int m){
      v = new T[size=m];
      for (int i = 0; i < size; i++){
          v[i] = 0;
      }
    }
    ```

- **Class Template Instantiation（类模板实例化）**
  - A **template class（模板类）** is a class built from a **class template（类模板）**
  - The process of creating a template class from a class template is called a **instantiation（实例化）**
  - The **compiler** will perform the error analysis **only when an instantiation takes place（编译器仅在实例化时进行错误分析）**
  - Can use **non-type parameters（非类型参数）** in templates
    - Default argument
    - Treated as `const`
    - Example:

```
template <class T, int size>
class array {
  T a[size];
  ...
};

array<int, 10> a1;
```

Creates array **at compiling time（在编译期间就已经创建了数组）**, rather than dynamic allocation at execution time

- **Function Template（函数模板）**
  - General form:

    ```
    template <class T1, class T2, ...>
    return_type function_name(T1 value1, T2 value2, ...) {
      ...
    }
    ```

  - Example:

    ```
    template <typename T>
    T min(const T &a, const T &b) {
      // operator `<` needs to be defined for the actual template parameter type
      // if not, compile-time error occurs
      if (a < b)
        return a;
      else
        return b;
    }
    ```

  - All matches for **formal parameters involving type parameters** must be **consistent（在模板函数调用时，包括类型参数在内的形参匹配必须一致）**
    - Only **trivial promotions（只有琐碎地匹配开销是被允许的）** to produce a match are allowed, for example, `int&` to `const int&` is allowed
    - Formal parameters **not involving type parameters must also be matched（非类型参数也需要被匹配）** without nontrivial conversion/promotion.
    - Example:

```cpp
#include <iostream>

using namespace std;

template <class T>
T min(const T &a, const T &b) {
  if (a < b)
    return a;
  else
    return b;
}

int main(){
  int value1 = 100;
  char value2 = 'a';

  cout << min(value1, 97) << endl;
  cout << min(value1, value2) << endl; // error
}
```

- **Overloading** of template functions:

```cpp
#include <iostream>

using namespace std;

template <class T> void display(T x) {
  cout << "template display:" << x << endl;
}

void display(int x) { cout << "Explicit display:" << x << endl; }

int main() {
  display(100); // Explicit display:100
  display(12.34); // template display:12.34
  display('c'); // template display:c
  return 0;
}
```

- **Function call resolution（函数调用解析）**
  - To resolve a function call, compiler follows **3 steps**:
    - Examine **all non-template** versions of the function, if any, for an exact match. **（先找非模板函数）**
      - error if there are more than one exact match.
    - Examine **all template** functions, if any, for an exact match. **（再找模板函数）**
      - error if there are more than one exact match.
    - If steps 1 and 2 do not resolve the call or produce an error, then re-examine all non-template versions of the function using call-resolution rules for regular **overloaded functions**. **（最后以重载函数的常规解析规则来匹配）**
  - Example:

```cpp
#include <iostream>

using namespace std;

template <class T> T max(T a, T b) {
  cout << "Call the template function" << endl;
  return (a > b) ? a : b;
}

int max(int a, int b) {
  cout << "Call the non-template function" << endl;
  return (a > b) ? a : b;
}

void f(int num, char ch) {
  max(num, num); // step 1 match
  max(num, ch); // step 3 match
}

int main() {
  f(65, 'a');
  return 0;
}
```

Output:

```
Call the non-template function
Call the non-template function
```

- **Non-Type Template Arguments（非类型模板参数）**：
  - Treated as `const`：

```cpp
template <typename T, int n>
T max1(T arr[n]) {
  T ans = arr[0];
  for (int i = 1; i < n; i++) {
    ans = (ans > arr[i]) ? ans : arr[i];
  }
  return ans;
}

int main() {
  int k = 5;
  int a[] = {1, 2, 3, 4, 5, 6, 7};
  cout << max1<int, 5>(a) << endl; // ok
  cout << max1<int, k>(a) << endl; // error
  return 0;
}
```

  - Difference between **non-type template arguments** and **common arguments**:

```cpp
template <typename T>
T max2(T arr[], int n) {
  T ans = arr[0];
  for (int i = 0; i < n; i++) {
    ans = (ans > arr[i]) ? ans : arr[i];
  }
  return ans;
}

int main() {
  int k = 5;
  int a[] = {1, 2, 3, 4, 5, 6, 7};
  cout << max2(a, 5) << endl; // ok
  cout << max2(a, k) << endl; // ok
  return 0;
}
```

- **Templates and friends（模板与友元）**
  - Friendships allowed between a **class template** and
    - **Global function**
    - **Member function of another class**
    - **Entire another class**
  - `friend` functions
    - Inside definition of class template `X`:
      - ```cpp
        friend void f1();
        ```
        - `f1()` is `friend` of all template class
      - ```cpp
        friend void f2(X<T> &);
        ```
        - `f2(X<int> &)` is a `friend` of `X<int>` only. The same applies for `float`, `double`, etc.
      - ```cpp
        friend void A::f3();
        ```
        - Member function `f3` of class `A` is a `friend` of all template classes
      - ```cpp
        friend void C<T>::f4(X<T> &);
        ```
        - `C<float>::f4(X<float> & )` is a `friend` of class `X<float>` only
  - `friend` classes
    - `friend` class `Y`, declared in template class `X`:
      - Class `Y` is a `friend` of every template class made from `X`.
      - Every member function of `Y` is a `friend` of every template class made from `X`.
    - `friend` class `Z<T>`
      - Class `Z<float>` is a `friend` of class `X<float>`, etc.
- **Templates and static Members（模板与静态成员）**
  - Non-template class
    - `static` data members **shared between all objects**
  - Template classes
    - Each class (`int`, `float`, etc.) **has its own copy** of `static` data members
    - `static` variables **initialized at file scope（在文件中已初始化）**
    - Each template class gets its **own copy of** `static` **member functions（各有各的静态成员函数）**