سلسلة تعلم البرمجة بلغة ++C الحديثة

Learn Modern C++ Programming Series

إعداد المهندس أحمد الديب

#39: Templates

Templates

Template Argument List

```
template <typename T, std::size_t N>
class array {
public:
  T elem[N];
  void print() {
    for(auto i : elem) std::cout << i << '\n';
  }
};


int main() {
  array<int, 3> arr{1, 2, 3};
  arr.print();
}
```

- Allows a type or a value to be a parameter in the definition of a class, a function, or a type alias.
- Support for generic programming.
- Type-safe.
- Does not imply any run-time mechanisms.
- Decreases the code generated because code for a member function of a class template is only generated if that member is used.
- Type checking is done (late) on the code generated by template instantiation.
- No requirements on argument list. We will take later about concepts.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Templates

```
template <typename T, std::size_t N>
class array {
 public:
  T elem[N];
  void print();
};
```

```
template <typename T, std::size_t N>
void array<T, N>::print() {
  for (auto i : elem) std::cout << i << '\n';
}
```

Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Type Aliases

```
template <std::size_t N>
using ArrayOfIntegers = array<int, N>;


int main() {
  array<int, 3> arr1{1, 2, 3};

  ArrayOfIntegers<3> arr2{1, 2, 3};
  arr2.print();
}
```

Useful for shortening the long names of classes generated from templates. Also, allows us to hide the fact that a type is generated from a template.

Aliases do not introduce new types, we always refer to the same generated type.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Static Members

```
template<typename T>
struct Shape {
  static const int hight = 10; // ok
  static int width = 20; // error: not const
  static int depth;
};

template<typename T>
int Shape<T>::hight = 30; // error: redefinition
```

A static data or function member that is not defined in-class must have a unique definition in a program.

As for non-template classes, a const or conexpr static data member of literal type can be initialized in-class and need not be defined outside the class.

A static member need only be defined if it is used.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Virtual Members

```
template <typename T>
struct Shape {
  static const int hight = 10;
  static const int width = 20;
  virtual void print() = 0;  // ok


  template <typename U>
  virtual void combine(U&) = 0;  // error: virtual on member function templates
};
```

The linker would have to add a new entry to the virtual table for class Shape each time someone called combine() with a new argument type. Complicating the implementation of the linker in this way was considered unacceptable.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Template Argument Deduction

```
template <typename T>
class Point {
 public:
  Point(T x, T y) : _x(x), _y(y) {}


 private:
  T _x, _y;
};


int main() {
  Point p1{1, 2};
  Point p2{1.1, 2.2};
}
```

Function Object

```cpp
// function
template <typename T>
bool Check_f(T value, T max, T min) {
  return value <= max && value >= min;
}

// function object
template <typename T>
class Check {
 public:
  Check(T max, T min) : _max(max), _min(min) {}
  bool operator()(T value) const { return value <= _max && value >= _min; }

 private:
  T _max, _min;
};


int main() {
  bool res1 = Check_f(50, 80, 10);

  Check check(80, 10);
  bool res2 = check(50);
}
```

Thank you