C++ Essentials

Kaiyuan Xu

May 17, 2021

Useful Materials

- cppreference.com
- *C++ Primer, 5th Edition* by Stanley Lippman, Josée Lajoie, Barbara Moo.
- *The C++ Programming Language, 4th Edition* by Bjarne Stroustrup.
- *C++ Templates: The Complete Guide, 2nd Edition* by David Vandevoorde, Nicolai Josuttis, Douglas Gregor.

From C `struct` to C++ `class`

C struct in C++:

plain old data = trivial type + standard layout type

```
1 // type_traits
2
3 template<class T>
4 struct is_trivial;
5
6 template<class T>
7 struct is_standard_layout;
```

POD is deprecated in c++20.

None Standard Layout Type

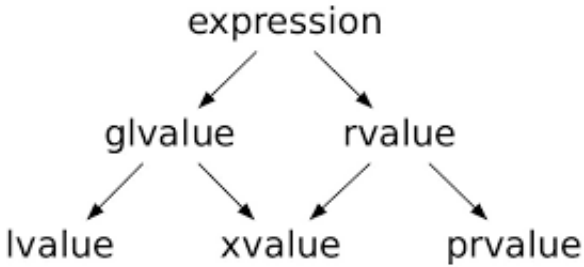
```
1  class Base{}; class BaseA: Base {}; class BaseB: Base{};
2
3  class Class :                    // none empty base class
4          virtual Base,            // virtual base class
5          BaseA, BaseB             // two base class with same type
6  {
7  private: protected: public:      // access control
8      Base &base;                  // reference as member
9
10     virtual void foo();          // virtual function
11 };
```

None Trivial Type

```
1 class Class {
2     Class();                              // constructor
3     Class(const Class &);                 // copy constructor
4     Class(Class &&);                      // move constructor
5     Class &operator=(const Class &);      // copy assignment operator
6     Class &operator=(Class &&);           // move assignment operator
7     ~Class();                             // deconstructor
8 };
```

Reference and Value Categories

```
1 int fn_a(); int a;
2 int &lvalue_ref = a;
3 int &&rvalue_ref = fn_a();
```



Move Sematics

```
1  template<class T>
2  T &&move(T &&t) { // notice: simplified implementation
3      return static_cast<T &&>(t);
4  }
5
6  class Class {
7  public:
8      Class(const Class &);                // copy constructor
9      Class(Class &&);                     // move constructor
10     Class &operator=(const Class &);     // copy assignment operator
11     Class &operator=(Class &&);          // move assignment operator
12 }
13
14 Class a{};
15 Class b{a};                          // copy constructor
16 Class c{move(a)};                   // move constructor
17 Class d = a;                        // copy assignment operator
18 Class e = move(a);                  // move assignment operator
```

Reference Collapsing and Perfect Forwarding

```
1  #include <utility>
2  #include <type_traits>
3
4  using lvalue = int &;
5  using rvalue = int &&;
6  using lvalue_lvalue = lvalue &;     // lvalue reference
7  using lvalue_rvalue = rvalue &;     // lvalue reference
8  using rvalue_lvalue = lvalue &&;    // lvalue reference
9  using rvalue_rvalue = rvalue &&;    // rvalue reference
10
11 template<class T>           // T can be lvalue reference or rvalue reference
12 void fn_a(T &&a) {          // what type is a?
13     fn_a(std::forward<T>(a));  // use perfect forwarding!
14 }
```

Operator overloading

```
1  operator op                      // + - * / % ^ & | ~ ! = < > += -= *= /= %= ^=
2                                   // &= |= << >> >>= <<= == != <= >=
3                                   // <=> (since C++20) && || ++ -- , ->* -> () []
4  operator type                    // user defined conversion function
5  operator new                     // allocation function
6  operator new []
7  operator delete                  // deallocation function
8  operator delete []
9  operator "" suffix-identifier    // user defined literal (since C++11)
10 operator co_await                // (since C++20)
```

Lambda Expression and Function Operator

```cpp
#include <iostream>
#include <functional>

struct FnA {
    int &a; FnA(int &a) : a{a} {}
    int operator()(int value) { return a += value; }
};
struct FnB {
    int b; FnB(int &b) : b{b} {}
    int operator()(int value) { return b += value; }
};

int main() {
    int a = 0, b = 0;
    std::function<int(int)> fn_a = [&](int value) { return a += value; };
    // std::function<int(int)> fn_a = FnA{a};
    std::function<int(int)> fn_b = [=](int value) mutable { return b += value; };
    // std::function<int(int)> fn_b = FnB{b};
    std::cout << fn_a(1) << ' ' << a << std::endl;  // 1 1
    std::cout << fn_b(1) << ' ' << b << std::endl;  // 1 0
}
```

Template

```
1  // template class
2  template<class T> class ClassA {}
3  // template function
4  template<class T> void fn(T) {}
5  // template type alias
6  template<class T> using Type = T;                    // since c++11
7  // template variables
8  template<int value> constexpr int value = value;     // since c++14
9  // template concept
10 template <class T, class U>                          // since c++20
11 concept Derived = std::is_base_of<U, T>::value;
12
13 // variable as template parameter
14 template<int value> class ClassB;
15 // type as template parameter
16 template<class T> class ClassC;
17 // template as template parameter
18 template<template<class> class T> class ClassD;
```

Template Instantiation and Argument Deduction

- The One Definition Rule
- On-Demand Instantiation
- Lazy Instantiation
- SFINAE: Substitution Failure Is Not An Error

Explicit Spetialization

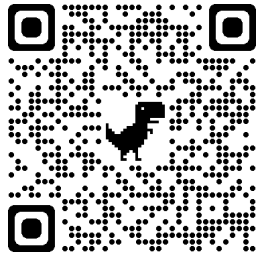
```
1  #include<iostream>
2
3  template<int a> struct A {
4      static constexpr int val = A<a - 1>::val + A<a - 2>::val;
5  };
6
7  template<> struct A<1> {
8      static constexpr int val = 1;
9  };
10
11 template<> struct A<2> {
12     static constexpr int val = 1;
13 };
14
15 int main() {
16     std::cout << A<10>::val << std::endl;
17 }
```

Member Function Pointer to Function Pointer

```cpp
#include <utility> // std::forward

template<class F, F f>
struct MemFnWrapper;

template<class Class, class Ret, class ...Args, Ret (Class::*f)(Args...) const>
struct MemFnWrapper<Ret (Class::*)(Args ...) const, f> {
    static Ret inner(const Class *self, Args...args) {
        return (self->*f)(std::forward<Args>(args)...);
    }
};

template<class Class, class Ret, class ...Args, Ret (Class::*f)(Args...)>
struct MemFnWrapper<Ret (Class::*)(Args ...), f> {
    static Ret inner(Class *self, Args...args) {
        return (self->*f)(std::forward<Args>(args)...);
    }
};
```

C++20: It compiles!

```cpp
#include<iostream>

auto fn_a(auto a, auto b) {
    return a + b;
}

int main() {
    std::cout << fn_a(1, 1) << std::endl;
    std::cout << fn_a(0.1, 0.1) << std::endl;
}
```



Q. & A.