12. Templates I

19. Generic Programming

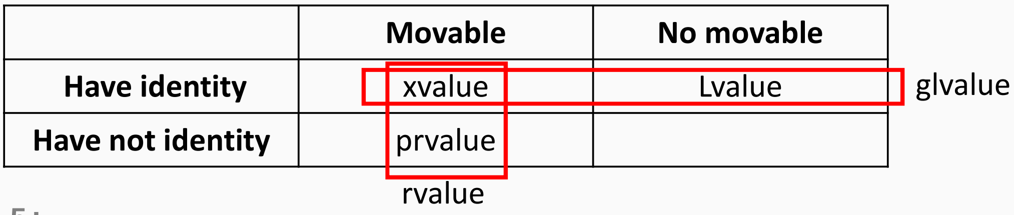
Rvalue References (1)

```
x + 2 = y; // Illegal!
-----------------------------------------------------
int x = 5;
int& r = x + 3; // Illegal!
-----------------------------------------------------
int x = 5;
const int& cr = x + 3; // Legal
-----------------------------------------------------
int g(int& n) {
   return 10 * n;
}
std::cout << g(x + 2) << '\n'; // Illegal!
-----------------------------------------------------
int h(const int& n) {
   return 10 * n;
}
std::cout << h(x + 2) << '\n'; // Legal
```

Rvalue References (2)

```
datatype &&Name;

lvalue,
rvalue,
glvalue, prvalue, xvalue
```



	Movable	No movable
Have identity	xvalue	Lvalue
Have not identity	prvalue	

```
int x = 5;
int&& r = x + 3; // Legal, note the two ampersands
std::cout << "x = " << x << " r = " << r << '\n';
```

Rvalue References (3)

```
#include <iostream>
class Ex {
public:
    int x;
    Ex(int x = 0) : x(x)     {     std::cout << "Constr." << std::endl;       }
    Ex(const Ex& e) : x(e.x) {     std::cout << "Copy constr." << std::endl;  }
    void Set(int xx)         {     x = xx;                                    }
    Ex operator +(int n)     {     return { x + n };                          }
};
void Fn1(Ex e1) {
    e1.Set(1);
}
void Fn2(Ex&& e1) {
    e1.Set(2);
}
int main() {
    Ex e;                    // Constr.: e
    Fn1(e);                  // Copy constr.: e1
    Fn2(e + 1);              // Constr.: e+1

    std::cout << e.x << std::endl;
}
```

Rvalue References (4)

```
// Move constructor
// Move assignment operator
// A move constructor enables the resources owned by an rvalue object
// to be moved into an lvalue without copying
class X {
   X(X&& other);
   X& operator=(X&& other);
};

className(className &&)
className(className &&) = default;
className(className &&) = delete;

className &operator=(className &&)
className &operator=(className &&) = default;
className &operator=(className &&) = delete;
```

Smart Pointers (1)

```
// A smart pointer is a wrapper class over a pointer with operator
// overloaded.
// shared_ptr, unique_ptr, weak_ptr (for circular references)

#include <iostream>
#include <memory>
struct Widget {
   int data;
   Widget(int n) : data(n) {}
   ~Widget() { std::cout << "Destroying: " << data << std::endl;  }
};
int main() {
   std::shared_ptr<Widget> p11(new Widget(11));
   std::shared_ptr<Widget> p12 = std::make_shared<Widget>(12);
   auto p13 = std::make_shared<Widget>(13);
   std::shared_ptr<Widget> p14 = p12;
   std::cout << p11.use_count() << std::endl;       // 1
   std::cout << p12.use_count() << std::endl;       // 2
```

Smart Pointers (2)

```
   p11.reset(); // p11 = nullptr;            // Destroying: 11
   p12.reset();
   p14.reset();                              // Destroying: 12
   {
      std::shared_ptr<Widget> p15 = std::make_shared<Widget>(15);
   }
   std::cout << (bool)p11 << std::endl;      // 0
   std::cout << (bool)p12 << std::endl;      // 0
   std::cout << (bool)p13 << std::endl;      // 1
   std::cout << (bool)p14 << std::endl;      // 0
}
```

Smart Pointers (3)

```
int main() {
   std::unique_ptr<Widget> p21(new Widget(21));
   std::unique_ptr<Widget> p22(new Widget(22));
   std::unique_ptr<Widget> p23 = std::make_unique<Widget>(23);
   ~~//std::shared_ptr<Widget> p29 = p21;~~
   p21.reset();                              // Destroying: 21
   Widget *p02 = p22.release();
   std::cout << (bool)p21 << std::endl;      // 0
   std::cout << (bool)p22 << std::endl;      // 0
   std::cout << (bool)p23 << std::endl;      // 1
   delete p02;                               // Destroying: 22
   std::unique_ptr<Widget> p24 = std::move(p23);
   std::cout << (bool)p23 << std::endl;      // 0
   std::cout << (bool)p24 << std::endl;      // 1

   std::cout << "Pointers" << std::endl;
   Widget *p = new Widget(0);
}       // Destroying: 23
```

Template

```
// Template: generic programming, function and class
// Template parameters: type template parameters, non-type template
//     parameters, and template template parameters.


std::array<int, 10> a{10};
Ex<int, std::vector> v;
```

Function Templates (1)

```
#include <iostream>
#include <string>

bool equal(int a, int b) {
   return a == b;
}
bool equal(std::string a, std::string b) {
   return a == b;
}

int main() {
   std::cout << equal(2, 3) << '\n';                // 0
   std::cout << equal(2.2, 2.7) << '\n';            // 1
   std::cout << equal("abc", "abcd") << '\n';       // 0
}
```

Function Templates (2)

```
// In C++, a template is a model of a function or a class that can
// be used to generate functions or classes.

template <class T>   // template <typename T>

#include <iostream>
#include <string>

template <class T>
bool equal(T a, T b) {
   return a == b;
}
int main() {
   std::cout << equal(2, 3) << '\n';                    // 0
   std::cout << equal(2.2, 2.7) << '\n';                // 0
   std::cout << equal("abc", "abcd") << '\n';           // 0
}
```

Function Templates (3)

```
template <typename T>
T sum(const std::vector<T>& v) {
   T result = 0;
   for (T elem : v)
      result += elem;
   return result;
}

template <typename ElemType>
void swap(ElemType& a, ElemType& b) {
   ElemType temp = a;
   a = b;
   b = temp;
}
```

Function Templates (4)

```
#include <iostream>

template <class T>
double average(T a, T b) {
   return (a+b)/2.;
}
int main() {
   std::cout << average(2, 3) << '\n';
   ~~std::cout << average(2, 2.7) << '\n';~~
   std::cout << average(2.2, 2.7) << '\n';
}

template <class T1, class T2>
double average(T1 a, T2 b) {
   return (a+b)/2.;
}
```

Function Templates (5)

```
#include <iostream>

template <class T>
T *new_var(int size) {
   return new T[size];
}

int main() {
   int *p1 = new_var<int>(10);
   double *p2 = new_var<double>(10);
}
```

Function Templates (6)

```
#include <iostream>
template <int N>
int scale(int value) {
   return value * N;
}

template <typename T, int N>
T scale(const T& value) {
   return value * N;
}

int main() {
   std::cout << scale<3>(5) << '\n';
   std::cout << scale<4>(10) << '\n';
   std::cout << scale<double, 3>(5.3) << '\n';
   std::cout << scale<int, 4>(10) << '\n';
}
```

Class Templates (1)

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

int main() {
   Point<int> p1(10, 10);
   Point<double> p2(10.5, 20.2);
   std::cout << p1.x << "," << p1.y << std::endl;
   std::cout << p2.x << "," << p2.y << std::endl;
}
```

Class Templates (2)

```
#include <iostream>
template <typename T>
class Point {
public:
   T x;   T y;
   Point(T x, T y) : x(x), y(y) {}
   void Print();
};
template <typename T>
void Point<T>::Print(){
   std::cout << x << "," << y << std::endl;
}
int main() {
   Point<int> p1(10, 10);
   Point<double> p2(10.5, 20.2);
   p1.Print();        p2.Print();
}
```

Class Templates (3)

```
#include <iostream>
template <typename T>
class Point {
public:
   T x;   T y;
   Point(T x, T y) : x(x), y(y) {}
   void Print();
};
template <typename T>
void Point<T>::Print() {
   std::cout << x << "," << y << std::endl;
}
template <> // Explicit specialization of template
void Point<int>::Print() {
   std::cout << x << ":" << y << std::endl;
}
int main() {
   Point<int> p1(10, 10);
   Point<double> p2(10.5, 20.2);
   p1.Print();p2.Print();
}
```