ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

RVALUE-REFERENCE

- **lvalue-expression** – correspond to objects you can refer to, either by name or by following a pointer or lvalue reference. You can always take the address of an lvalue expression.
- **rvalue-expression** – correspond to temporary objects. You can't take the address of an rvalue expression.

- **Type &** – lvalue-reference. Can only be associated with an lvalue object.
- **Type &&** – rvalue-reference. Conceptually, it is considered a reference to a temporary object. But it can be associated with both an rvalue object and an lvalue object (by type casting).

```cpp
int obj = 0;
int& obj_lref = obj;    // lvalue-reference
int& obj_lref2 = 1;     // Error!

int&& obj_rref = 1;     // rvalue-reference
int&& obj_rref2 = obj; // Error!
int&& obj_rref3 = static_cast<int&&>(obj); // rvalue-reference
int&& obj_rref4 = std::move(obj); // rvalue-reference

struct Point {
public:
    ...
    Point(Point&& rhs);
    ...
};
```

rhs is an lvalue, though it has an rvalue reference type

TEMPLATE TYPE DEDUCTION

```
// Pseudocode of a function template
template <typename T>
void func(ParamType param);

func(expr); // Deduce T and ParamType from expr.
```

```
template <typename T>
void func(const T& param); // ParamType - const T&

int obj = 0;
func(obj);  // Call func with an int
            // T - int.
            // ParamType - const int&.
```

THREE CASES

- **ParamType** is a Reference or Pointer, but not a Universal Reference.
- **ParamType** is Universal Reference.
- **ParamType** is neither a pointer nor a reference.

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(ParamType param);

func(expr);
```

Type deduction rules for **T**:

1. If **expr**'s type is a reference, ignore the reference part.
2. Then pattern-match **expr**'s type against **ParamType** to determine **T**.

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(T& param);

int x = 27;          // x is an int
const int cx = x;   // cx is a const int
const int& rx = x;  // rx is a const int&

func(x);     // T - int,              type of param - int&
func(cx);    // T - ???,              type of param - ???
func(rx);    // T - ???,              type of param - ???
```

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(T& param);

int x = 27;           // x is an int
const int cx = x;     // cx is a const int
const int& rx = x;    // rx is a const int&

func(x);     // T - int,           type of param - int&
func(cx);    // T - const int,     type of param - const int&
func(rx);    // T - const int,     type of param - const int&
```

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(const T& param); // param is now a ref-to-const

int x = 27;          // x is an int
const int cx = x;    // cx is a const int
const int& rx = x;   // rx is a const int&

func(x);     // T - ???,    type of param - ???
func(cx);    // T - ???,    type of param - ???
func(rx);    // T - ???,    type of param - ???
```

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(const T& param); // param is now a ref-to-const

int x = 27;           // x is an int
const int cx = x;     // cx is a const int
const int& rx = x;    // rx is a const int&

func(x);    // T - int,    type of param - const int&
func(cx);   // T - int,    type of param - const int&
func(rx);   // T - int,    type of param - const int&
```

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(T* param); // param is now a pointer

int x = 27;             // x is an int
const int* px = &x; // px is a const int*

func(&x);   // T - ???,          type of param - ???
func(px);   // T - ???,          type of param - ???
```

1. ParamType is a Reference or Pointer, but not a Universal Reference

```
template <typename T>
void func(T* param); // param is now a pointer

int x = 27;          // x is an int
const int* px = &x; // px is a const int*

func(&x);   // T - int,         type of param - int*
func(px);   // T - const int,   type of param - const int*
```

2. ParamType is Universal Reference

```
template <typename T>
void func(ParamType param);

func(expr);
```

Universal Reference:
ParamType declared as T&&

Type deduction rules for **T**:

- If **expr** is an *lvalue*, both **T** and **ParamType** are deduced to be *lvalue* references.
- If **expr** is an *rvalue*, the "normal" (i.e., **Case 1**) rules apply.

2. ParamType is Universal Reference

```
template <typename T>
void func(T&& param);

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a const int&

func(x);      // x  - lvalue, T - int&,        param - int&
func(cx);     // cx - ???,    T - ???,         param - ???
func(rx);     // rx - ???,    T - ???,         param - ???
func(27);     // 27 - ???,    T - ???,         param - ???

func(std::move(x)); // std::move(x) - ???, T - ???
                    // param - ???
```

2. ParamType is Universal Reference

```
template <typename T>
void func(T&& param);

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a const int&

func(x);     // x  - lvalue, T - int&,        param - int&
func(cx);    // cx - lvalue, T - const int&, param - const int&
func(rx);    // rx - lvalue, T - const int&, param - const int&
func(27);    // 27 - rvalue, T - int,         param - int&&

func(std::move(x)); // std::move(x) - rvalue, T - int
                    // param - int&&
```

3. ParamType is Neither a Pointer nor a Reference

```
template <typename T>
void func(ParamType param);

func(expr);
```

Pass-by-value:
ParamType declared as T

Type deduction rules for **T**:

1. If **expr**'s type is a reference, ignore the reference part.
2. If, after ignoring **expr**'s reference-ness, **expr** is **const**, ignore that, too. If it's **volatile**, also ignore that.

3. ParamType is Neither a Pointer nor a Reference

```
template <typename T>
void func(T param);

int x = 27;          // x is an int
const int cx = x;   // cx is a const int
const int& rx = x;  // rx is a const int&

func(x);     // T - ???,    param - ???
func(cx);    // T - ???,    param - ???
func(rx);    // T - ???,    param - ???
```

3. ParamType is Neither a Pointer nor a Reference

```
template <typename T>
void func(T param);

int x = 27;          // x is an int
const int cx = x;  // cx is a const int
const int& rx = x; // rx is a const int&

func(x);    // T - int,   param - int
func(cx);   // T - int,   param - int
func(rx);   // T - int,   param - int
```

3. ParamType is Neither a Pointer nor a Reference

```
template <typename T>
void func(T param);

const char* const ptr =    // ptr is const pointer to const object
     "Fun with pointers";

func(ptr);     // param - ???
```

3. ParamType is Neither a Pointer nor a Reference

```
template <typename T>
void func(T param);

const char* const ptr =   // ptr is const pointer to const object
    "Fun with pointers";

func(ptr);    // param - const char*
```

Array Arguments

```
template <typename T>
void func1(T param);

template <typename T>
void func2(T& param);

const char ptr[] = "Hello world"; // ptr - const char[12]

func1(ptr);   // T - const char*,      param - const char*
func2(ptr);   // T - const char[12],  param - const char (&)[12]
```

Function Arguments

```
template <typename T>
void func1(T param);

template <typename T>
void func2(T& param);

void someFunc(int, double); // type is void(int, double)

func1(someFunc);    // param - void(*)(int, double)
func2(someFunc);    // param - void(&)(int, double)
```

AUTO TYPE DEDUCTION

```
auto x = 27;

const auto cx = x;

const auto& rx = x;
```

Applying template type deduction rules

```
template <typename T>
void func_for_x(T param);

func_for_x(27);
```

```
template <typename T>
void func_for_cx(const T param);

func_for_cx(x);
```

```
template <typename T>
void func_for_rx(const T& param);

func_for_rx(x);
```

AUTO TYPE DEDUCTION

```
auto x = 27;           // x is an int
const auto cx = x;  // cx is a const int
const auto& rx = x; // rx is a const int&

auto&& uref1 = x;  // x  - int and lvalue, uref1 - int&
auto&& uref2 = cx; // cx - const int and lvalue,
                   //      uref2 - const int&

auto&& uref3 = 27; // 27 - int and rvalue, uref3 - int&&
```

```
int x1 = 27;  // C++98
int x2(27);   // C++98
int x3 = {27};  // C++11
int x4 {27};  // C++11

// Replace with a keyword auto
auto x1 = 27;   // int
auto x2(27);    // int
auto x3 = {27}; // std::inializer_list<int>

auto x4 {27};   // std::inializer_list<int> until C++17
               // int since C++17

auto x5 {27, 1}; // std::inializer_list<int> until C++17
                // Error! since C++17
```

Initializers in braces is the only difference between auto type deduction and template type deduction.

```cpp
auto x1 = { 1, 2 };    // type of x - std::inializer_list<int>

auto x2 = { 1, 2. };  // Error! Error of deducing type T
                      // for template std::inializer_list<T>

template <typename T>
void func1(T param);

func1({1, 2, 3}); // Error! Error of deducing type T
                  // for template func1<T>.

template <typename T>
void func2(std::initializer_list<T> param);

func2({1, 2, 3}); // T - int,
                  // param - std::initializer_list<int>
```

UNIFORM (BRACED) INITIALIZATION

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

ADVANTAGES {}

- It prohibits implicit narrowing conversions. **char** may be expressible as **int. double** may not be expressible as **int**.
- There is no conversion between integer types and floating point types.

```
void f(double val, int val2) {
    int x2 = val;       // if val==7.9, x2 <- 7
    char c2 = val2;     // if val2==1025, c2 <- 1

    int x3 {val};       // Error
    char c3 {val2};     // Error
    char c4 {24};       // OK. char contains 24
    char c5 {264};      // Error. char doesn't contain 264
    int x4 {2.0};       // Error. int and double.
    // ...
}
```

```
int x4 {};                  // 0
double d4 {};               // 0.0
char *p {};                 // nullptr
vector<int> v4{};           // empty vector
string s4 {};               // empty string
char buf[1024] {};          // all characters
                            // are 0

vector<int> vec();          // declare function
```

{} — default value

```cpp
struct Work {
    string author;
    string name;
    int year;
};

// Aggregate initialization
Work s9 {
    "Beethoven",
    "Symphony No. 9 in D minor, Op. 125; Choral",
    1824
};

// Default Copy Constructor
Work currently_playing { s9 };

Work none {}; // as if: { {}, {}, {} } or { "", "", 0 }
```

Initializing class objects without constructors

```
struct X {
    X(int);
};

X x0;               // Error. No initialization
X x1 {};            // Error. Empty initialization
X x2 {2};           // OK
X x3 {"two"};       // Error. Wrong type.
X x4 {1,2};         // Error. Invalid count arguments.
X x5 {x4};          // OK. Default copy constructor.
```

Initialization by constructor

```cpp
struct S1 {
    int a, b;
};

struct S2 {
    int a, b;
    S2(int aa = 0, int bb = 0)
      : a(aa), b(bb) {}
};

S1 x11(1,2);  // Error. No constructor
S1 x12 {1,2}; // OK. Aggregate initialization
S1 x13(1);    // Error. No constructor
S1 x14 {1};   // OK. x14.b <- 0
S2 x21(1,2);  // OK. Constructor
S2 x22 {1,2}; // OK. Constructor
S2 x23(1);    // OK. Constructor with default args.
S2 x24 {1};   // OK. Constructor with default args.
```

```
vector<double> v { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages {
    { "Nygaard", "Simula" },
    { "Richards", "BCPL" },
    { "Ritchie", "C"}
};

map<vector<string>,vector<int>> years {
    { { "Maurice", "Vincent", "Wilkes" },
      { 1913, 1945, 1951, 1967, 2000 } },
    { { "Martin", "Richards"},
      { 1982, 2003, 2007 } },
    { { "David", "John", "Wheeler"},
      { 1927, 1947, 1951, 2004 } }
};

// How?!!
```

Arbitrary number of constructor arguments

```cpp
#include <initializer_list>
#include <iostream>

void f(std::initializer_list<int> list) {
    std::cout << "Size = " << list.size() << std::endl;

    for (int x: list)
        std::cout << x << std::endl;
}

f({1, 2});
f({23, 345, 4567, 56789});
f({}); // Empty list
```

std::initializer_list<T>

```cpp
struct X {
    X(initializer_list<int>);
    X();
    X(int);
};

X x0 {};      // X()
X x1 ({});    // X(initializer_list<int>)
X x2 {1};     // X(initializer_list<int>)
```

- If you can call a **default constructor** or a **constructor with initializer_list**, the first one is used.
- If you can call a **constructor with initializer_list** or a "usual" constructor, the first one is also used.

```cpp
vector<int> v1 {1};   // Vector contains one element: 1
vector<int> v2 {1,2}; // Vector contains two elements: 1,2
vector<int> v3(1);    // Vector contains one element: 0(by default)
vector<int> v4(1,2);  // Vector contains one element: 2
```

The difference between () and {}

КОНЕЦ ТРЕТЬЕЙ ЛЕКЦИИ