Not So Tiny Task №12 (1 point)

Implement an iterator in a collection you've implemented in tasks 1,2,3,4,9.

Check that ranged-base loop works well! (add some tests on that)

System Programming with C++

Basic iterators, ranged-based for, type deduction, auto and decltype

Iterators

Task: iterate all elements in some data structure

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int> v{16};
v.push(13);
v.push(42);
```

Iterators

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```cpp
Vector<int> v{16};
v.push(13);
v.push(42);

for (size_t i = 0; i < v.size(); i++) {
   std::cout << v[i];
}
```

Iterators

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```cpp
Vector<int> v{16};
v.push(13);
v.push(42);

for (size_t i = 0; i < v.size(); i++) {
   std::cout << v[i];
}
```

Do you see any problems here?

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

for (size_t i = 0, size_t e = v.size();
     i != e; ++i) {
   std::cout << v[i];
}
```

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

for (size_t i = 0, size_t e = v.size();
     i != e; ++i) {
   std::cout << v[i];
}
```

Now let's try to generalize that.

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
template <typename T>
void print_all(T& t) {
   for (size_t i = 0, size_t e = t.size();
         i != e; ++i) {
       std::cout << t[i] << std::endl;
   }
}


Vector<int> v{16};
v.push(13);
v.push(42);

print_all(v);
```

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
template <typename T>
void print_all(T& t) {
   for (size_t i = 0, size_t e = t.size();
        i != e; ++i) {
       std::cout << t[i] << std::endl;
   }
}
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

print_all(v);
```

Will work, but only for collections with random-access (operator[] overloaded) and sequential elements placement.

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
template <typename T>
void print_all(T& t) {
   for (size_t i = 0, size_t e = t.size();
        i != e; ++i) {
       std::cout << t[i] << std::endl;
   }
}
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

print_all(v);
```

Will work, but only for collections with random-access (operator[] overloaded) and sequential elements placement.

What about std::list?

Iterators

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
template <typename T>
void print_all(T& t) {
   for (size_t i = 0, size_t e = t.size();
        i != e; ++i) {
       std::cout << t[i] << std::endl;
   }
}
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

print_all(v);
```

Will work, but only for collections with random-access (operator[] overloaded) and sequential elements placement.

What about std::list?
What about hashmaps?

Iterators

We need some separate API exclusively for iterating over collections!

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
template <typename T>
void print_all(T& t) {
   for (size_t i = 0, size_t e = t.size();
         i != e; ++i) {
        std::cout << t[i] << std::endl;
   }
}
```

```
Vector<int> v{16};
v.push(13);
v.push(42);

print_all(v);
```

Will work, but only for collections with random-access (operator[] overloaded) and sequential elements placement.

What about std::list?
What about hashmaps?

Iterators

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);
```

Iterators

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

Iterators

iterator is a nested class that,
well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

Iterators

iterator is a nested class that, well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

iterators mimic pointers

stdv	13	42	66

Iterators

iterator is a nested class that, well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

iterators mimic pointers

stdv	13	42	66

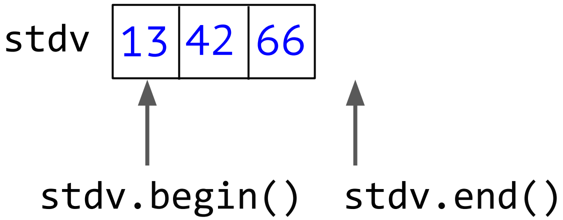
stdv.begin()

Iterators

iterator is a nested class that, well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

iterators mimic pointers

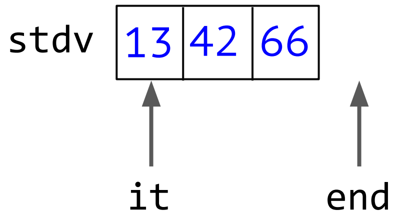


Iterators

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();
```

iterator is a nested class that, well, used for iteration.

iterators mimic pointers

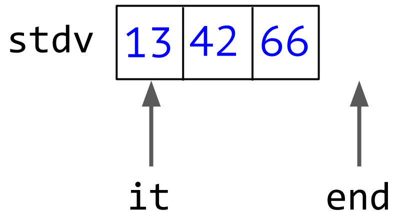


Iterators

iterator is a nested class that,
well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();

for (; it != end; ++it) {
    ...
}
```

iterators mimic pointers

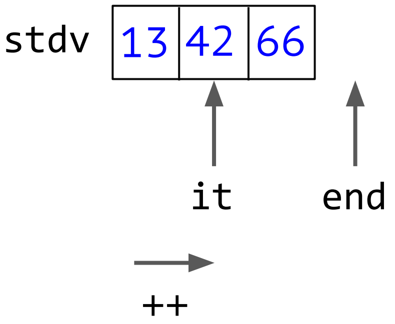


Iterators

iterator is a nested class that, well, used for iteration.

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();

for (; it != end; ++it) {
    ...
}
```

iterators mimic pointers

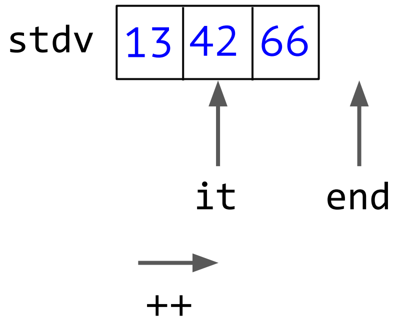


Iterators

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << *it << std::endl;
}

// 13 42 66
```

iterator is a nested class that, well, used for iteration.

iterators mimic pointers



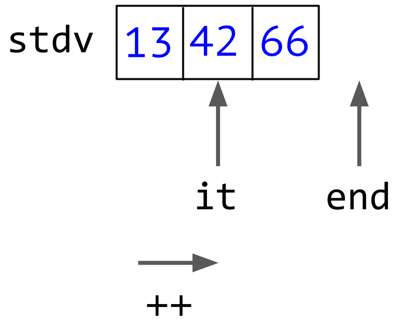
Iterators

```
std::vector<int> stdv;
stdv.push_back(13);
stdv.push_back(42);
stdv.push_back(66);

std::vector<int>::iterator it = stdv.begin();
std::vector<int>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << *it << std::endl;
}

// 13 42 66
```

iterator is a nested class that, well, used for iteration.

Usually returns reference to element

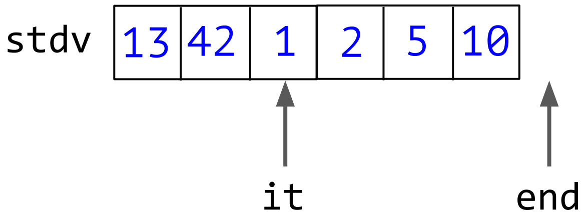
iterators mimic pointers



Iterators

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << (*it).x << std::endl;
    std::cout << (*it).y << std::endl;
}
```

iterators mimic pointers

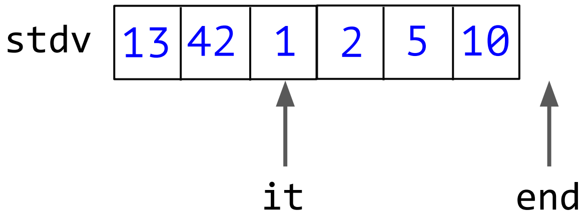


Iterators

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << (*it).x << std::endl;
    std::cout << (*it).y << std::endl;
}
```

no copying here

iterators mimic pointers

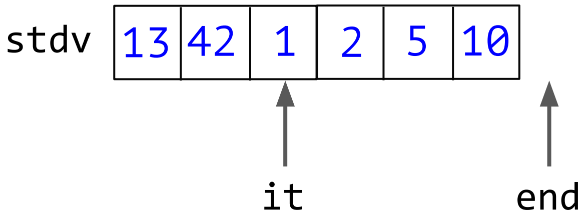


Iterators

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << it->x << std::endl;
    std::cout << it->y << std::endl;
}
```

no copying here

iterators mimic pointers



Iterators

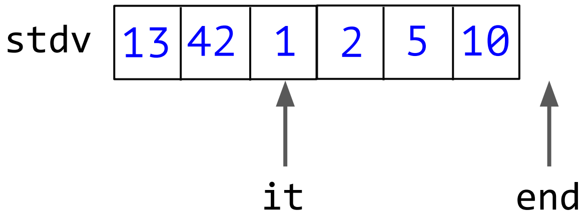
This gives you a guide, how to make your collection iterable:

1. Add a nested class for iterator (name is not important)

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << it->x << std::endl;
    std::cout << it->y << std::endl;
}
```

no copying here

iterators mimic pointers



Iterators

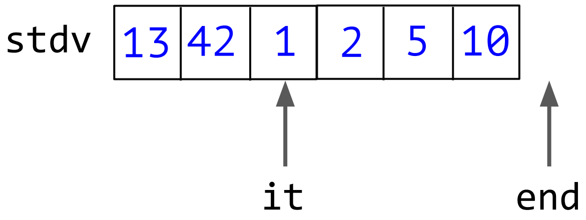
This gives you a guide, how to make your collection iterable:

1. Add a nested class for iterator (name is not important)
2. Overload !=, ++, -> and * there

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << it->x << std::endl;
    std::cout << it->y << std::endl;
}
```

no copying here

iterators mimic pointers



Iterators

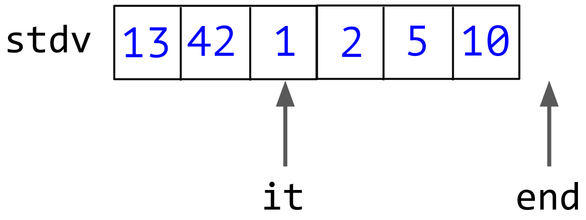
This gives you a guide, how to make your collection iterable:

1. Add a nested class for iterator (name is not important)
2. Overload !=, ++, -> and * there
3. Add functions begin() and end() that return iterators to your collection class.

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << it->x << std::endl;
    std::cout << it->y << std::endl;
}
```

no copying here

iterators mimic pointers

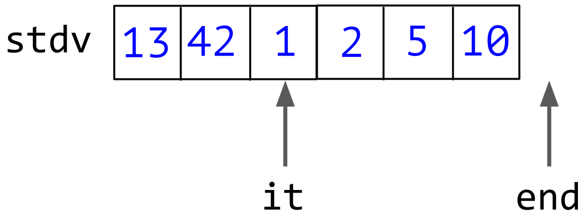


Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << it->x << std::endl;
    std::cout << it->y << std::endl;
}
```

no copying here

iterators mimic pointers



Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (Point& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Ranged-base for

```cpp
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (Point& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Semantically the same as previous one.

https://cppinsights.io/s/5a224bc4

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (Point& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Semantically the same as previous one.

Works for std collections, custom collections with begin() and end(), static arrays (not pointers).

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (Point& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Semantically the same as previous one.

Works for std collections, custom collections with begin() and end(), static arrays (not pointers).

Iterators

- Implementing an iterator is a must when developing collection: both language features and functions from std require that

Iterators

- Implementing an iterator is a must when developing collection: both language features and functions from std require that

- There are different types of iterators:
  - ✓ Can you go only forward or backward as well?
  - ✓ Can you iterate several times?
  - ✓ Can you change elements or not?
  - ✓ Should it be direct, or reverse iterator?

Iterators

- Implementing an iterator is a must when developing collection: both language features and functions from std require that

- There are different types of iterators:

  ✓ Can you go only forward or backward as well?
  ✓ Can you iterate several times?
  ✓ Can you change elements or not?
  ✓ Should it be direct, or reverse iterator?

  https://en.cppreference.com/w/cpp/iterator

Not So Tiny Task №12 (1 point)

Implement an iterator in a collection you've implemented in tasks 1,2,3,4,9.

Check that ranged-base loop works well! (add some tests on that)

Further directions

- More detailed templates implementation + meta-programming + compile-time evaluation
- Is it really necessary to specify a type explicitly each time for instantiation?
- Variadic templates and requires

Type inference for template arguments

Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

---------------------------

int a = 13, b = 42;
int c = max<int>(a, b);
```

T - formal template parameter

int - actual parameter type in this instantiation.

Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
---------------------------

int a = 13, b = 42;
int c = max<int>(a, b);
```

int - actual parameter type in this instantiation.

But isn’t it kinda obvious that we want to use int as actual parameter here?

Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

----------------------------

int a = 13, b = 42;
int c = max(a, b);
```

T - formal template parameter

int - actual parameter type in this instantiation.

But isn't it kinda obvious that we want to use int as actual parameter here? It is!

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
int a = 13, b = 42;
int c = max(a, b);
```

Compiler automatically deducted that type is int here (based on given args).

But isn't it kinda obvious that we want to use int as actual parameter here? It is!

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
---------------------------

float a = 13.0, b = 42.0;
float c = max(a, b);
```

Compiler automatically deducted that type is float here (based on given args).

But isn't it kinda obvious that we want to use float as actual parameter here? It is!

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

---------------------------

float a = 13.0;
int b = 42;
float c = max(a, b);
```

T - formal template parameter

But isn’t it kinda obvious that we want to use…

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
---------------------------

float a = 13.0;
int b = 42;
float c = max(a, b);
```

But isn't it kinda obvious that we want to use… well, it isn't.

```
error: no matching function for call to 'max(float&, int&)'
```

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
---------------------------

float a = 13.0;
int b = 42;
float c = max(a, b);
```

But isn’t it kinda obvious that we want to use… well, it isn’t.

```
error: no matching function for call to 'max(float&, int&)'
note:   template argument deduction/substitution failed:
note:   deduced conflicting types for parameter 'T' ('float' and 'int')
```

Template arguments deduction

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
float a = 13.0;
int b = 42;
float c = max<float>(a, b);
```

But isn’t it kinda obvious that we want to use… well, it isn’t.

Here you have to specify it manually.

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
   T tmp = a;
   a = b;
   b = tmp;
}

int a = 13;
int b = 42;

std::cout << a << ", " << b << std::endl;
swap(a, b);
std::cout << a << ", " << b << std::endl;
```

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
   T tmp = a;
   a = b;
   b = tmp;
}

int a = 13;
int b = 42;

std::cout << a << ", " << b << std::endl; // 13 42
swap(a, b);
std::cout << a << ", " << b << std::endl; // 13 42
```

No deduction problems, but doesn’t swap elements (quite expected)

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int a = 13;
int& ra = a;
int b = 42;
int& rb = b;

std::cout << ra << ", " << rb << std::endl;  // 13 42
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // ?? ??
```

Template arguments deduction

```cpp
template <typename T>
void swap(T a, T b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int a = 13;
int& ra = a;
int b = 42;
int& rb = b;

std::cout << ra << ", " << rb << std::endl;  // 13 42
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // 13 42
```

T deduced to int, not int&!

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int a = 13;
cosnt int& ra = a;
int b = 42;
cosnt int& rb = b;

std::cout << ra << ", " << rb << std::endl;  // 13 42
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // 13 42
```

T deduced to int, not const int&!

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int a = 13;
cosnt int& ra = a;
int b = 42;
cosnt int& rb = b;

std::cout << ra << ", " << rb << std::endl;  // 13 42
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // 13 42
```

T deduced to int, not const int&!

Why?

Template arguments deduction

```
template <typename T>
void swap(T a, T b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int a = 13, b = 42;
cosnt int& rb = b;

std::cout << a << ", " << rb << std::endl;  // 13 42
swap(a, rb);
std::cout << a << ", " << rb << std::endl;  // 13 42
```

T deduced to int, not const int&!

Why? To reduce ambiguous situations.

Template arguments deduction

Everything changes, when you explicitly specify references or pointers in template argument!

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

int a = 13;
int b = 42;

std::cout << a << ", " << b << std::endl; // 13 42
swap(a, b);
std::cout << a << ", " << b << std::endl; // 42 13
```

Template arguments deduction

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

int a = 13;
int b = 42;

std::cout << a << ", " << b << std::endl; // 13 42
swap(a, b);
std::cout << a << ", " << b << std::endl; // 42 13
```

Everything changes, when you explicitly specify references or pointers in template argument!

Here compiler thinks that you know what you're doing.

Template arguments deduction

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

int a = 13;
int& ra = a;

int b = 42;
int& rb = b;

std::cout << ra  << ", " << rb << std::endl; // 13 42
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // 42 13
```

Everything changes, when you explicitly specify references or pointers in template argument!

Here compiler thinks that you know what you're doing.

Template arguments deduction

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

int a = 13;
const int& ra = a;

int b = 42;
const int& rb = b;

std::cout << ra  << ", " << rb << std::endl; // ???
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // ???
```

Everything changes, when you explicitly specify references or pointers in template argument!

Here compiler thinks that you know what you're doing.

Template arguments deduction

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

int a = 13;
const int& ra = a;

int b = 42;
const int& rb = b;

std::cout << ra  << ", " << rb << std::endl; // ???
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // ???
```

Everything changes, when you explicitly specify references or pointers in template argument!

Here compiler thinks that you know what you're doing.

```
In instantiation of 'void swap(T&, T&) [with T = const int]':
error: assignment of read-only reference 'a'
```

Template arguments deduction

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

int a = 13;
const int& ra = a;

int b = 42;
const int& rb = b;

std::cout << ra  << ", " << rb << std::endl; // ???
swap(ra, rb);
std::cout << ra << ", " << rb << std::endl;  // ???
```

Everything changes, when you explicitly specify references or pointers in template argument!

Here compiler thinks that you know what you're doing. And it trusts you fully.

```
In instantiation of 'void swap(T&, T&) [with T = const int]':
error: assignment of read-only reference 'a'
```

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.

- Implicit casts are not taken into account during deduction!

```
double a = 13.0;
Matrix b(42.0);
double c = max(a, b); // will not work even if
                      // there is an implicit cast
                      // from Matrix to float
```

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.

- Implicit casts are not taken into account during deduction!

- During this deduction references, const and etc are cut off. The full list of rules is here.

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.

- Implicit casts are not taken into account during deduction!

- During this deduction references, const and etc are cut off. The full list of rules is here.

- You can specify that you mean references or pointers, then deduction will take it into account and not cut them off.

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.
- Implicit casts are not taken into account during deduction!
- During this deduction references, const and etc are cut off. The full list of rules is here.
- You can specify that you mean references or pointers, then deduction will take it into account and not cut them off.
- There are special rules for rvalue refs, will discuss them later.

What else can be deducted?

Class template argument deduction (CTAD)

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ...
};
```

Class template argument deduction (CTAD)

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ...
};


ScopedPointer sp{new int{13}};
```

Class template argument deduction (CTAD)

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ...
};


ScopedPointer sp{new int{13}};
```

Template argument T for the class was deducted

Class template argument deduction (CTAD)

```
template<typename T>
class ScopedPointer {
   T* pointer;

public:
   ScopedPointer(T* raw): pointer(raw) { }
   ...
};
```

```
ScopedPointer sp{new int{13}};
```

Template argument T for the class was deducted

Rules are all the same as for usual function calls (basically, ctrs are functions)

What else can be deducted?

What else can be deducted? What about locals?

auto

auto

```
int main() {
    int a = 10;

    auto x = a + 23;          // deducted to int
    auto y = x * 1.5;         // deducted to double
    std::cout << x + y;

    std::string z = "test";
    auto q = z + " print";    // deducted to std::string
    return 0;
}
```

auto

```
template <typename T>
void foo(T t);

int main() {
    int a = 10;

    auto x = a + 23;            // deducted to int
    auto y = x * 1.5;           // deducted to double
    std::cout << x + y;

    std::string z = "test";
    auto q = z + " print";    // deducted to std::string
    return 0;
}
```

auto

```
template <typename T>
void foo(T t);

int main() {
    int a = 10;
    foo(a + 23);                  // deducted to foo<int>(a + 23)
    auto x = a + 23;              // deducted to int

    auto y = x * 1.5;             // deducted to double
    std::cout << x + y;

    std::string z = "test";
    auto q = z + " print";        // deducted to std::string
    return 0;
}
```

auto

```cpp
template <typename T>
void foo(T t);

int main() {
    int a = 10;
    foo(a + 23);               // deducted to foo<int>(a + 23)
    auto x = a + 23;           // deducted to int

    foo(x * 1.5);              // deducted to foo<double>(x * 1.5)
    auto y = x * 1.5;          // deducted to double
    std::cout << x + y;

    std::string z = "test";
    foo(z + " print");         // deducted to foo<std::string>(...)
    auto q = z + " print";     // deducted to std::string
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

```
template <typename T>
void foo(T t);

int main() {
    int a = 10;
    foo(a + 23);            // deducted to foo<int>(a + 23)
    auto x = a + 23;        // deducted to int

    foo(x * 1.5);           // deducted to foo<double>(x * 1.5)
    auto y = x * 1.5;       // deducted to double
    std::cout << x + y;

    std::string z = "test";
    foo(z + " print");      // deducted to foo<std::string>(...)
    auto q = z + " print";  // deducted to std::string
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

   technical detail: when using auto you must immediately initialize var (to deduct from)

```
template <typename T>
void foo(T t);

int main() {
    int a = 10;
    foo(a + 23);               // deducted to foo<int>(a + 23)
    auto x = a + 23;           // deducted to int

    foo(x * 1.5);              // deducted to foo<double>(x * 1.5)
    auto y = x * 1.5;          // deducted to double
    std::cout << x + y;

    std::string z = "test";
    foo(z + " print");         // deducted to foo<std::string>(...)
    auto q = z + " print";     // deducted to std::string
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

2. auto works exactly the same as template arguments deduction.

```
template <typename T>
void foo(T t);

int main() {
    int a = 10;
    foo(a + 23);              // deducted to foo<int>(a + 23)
    auto x = a + 23;          // deducted to int

    foo(x * 1.5);             // deducted to foo<double>(x * 1.5)
    auto y = x * 1.5;         // deducted to double
    std::cout << x + y;

    std::string z = "test";
    foo(z + " print");        // deducted to foo<std::string>(...)
    auto q = z + " print";    // deducted to std::string
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

2. auto works exactly the same as template arguments deduction.

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);           // ---> deducted to what?
    auto aa = ra;      // ---> deducted to what?
    ...
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

2. auto works exactly the same as template arguments deduction.

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to int
    auto aa = ra;       // ---> deducted to int
    ...
    return 0;
}
```

auto

1. auto commands compiler to deduct a type of variable (no dynamic typing here, only static!!!)

2. auto works exactly the same as template arguments deduction.

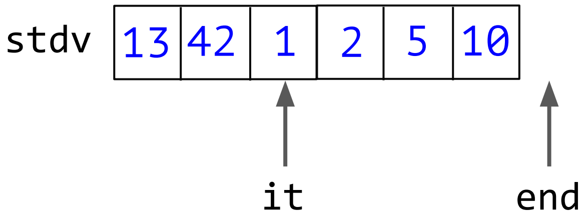
```
template <typename T> void foo(T& t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to const int&
    auto& aa = ra;      // ---> deducted to const int&
    ...
    return 0;
}
```

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

std::vector<Point>::iterator it = stdv.begin();
std::vector<Point>::iterator end = stdv.end();

for (; it != end; ++it) {
    std::cout << (*it).x << std::endl;
    std::cout << (*it).y << std::endl;
}
```

no copying here

iterators mimic pointers

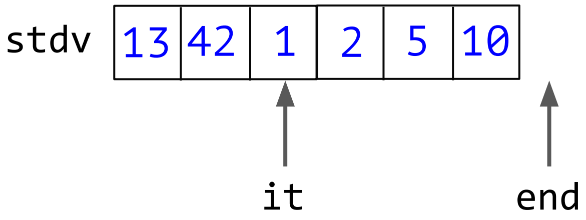


Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    std::cout << (*it).x << std::endl;
    std::cout << (*it).y << std::endl;
}
```

no copying here

iterators mimic pointers



Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (Point& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Semantically the same as previous one.

https://cppinsights.io/s/5a224bc4

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (auto element: stdv) {
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (auto element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

What will happen here?

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What will happen here?

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What will happen here?
& will be cut off,
so despite the fact that
*it (usually) returns
reference…

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What will happen here?
& will be cut off,
so despite the fact that
*it (usually) returns
reference… there will be
copying here!

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (auto element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

What will happen here?
& will be cut off, local copy will be created on each iteration

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (auto& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Now compiler thinks that you know what you are doing, so it is ok.

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

for (auto& element: stdv) {
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using auto& here?

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using auto& here?

Ranged-base for

```
std::vector<Point> stdv;
stdv.push_back(Point{13, 42});
stdv.push_back(Point{1, 2});
stdv.push_back(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

```
error: cannot bind non-const lvalue
reference of type 'Point&' to an rvalue
of type 'Point'
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

for (auto& element: stdv) {
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

```
error: cannot bind non-const lvalue
reference of type 'Point&' to an rvalue
of type 'Point'
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

Ranged-base for

So, we still need to find a better way to generalize our code. 🤔

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

for (auto& element: stdv) {
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

```
error: cannot bind non-const lvalue
reference of type 'Point&' to an rvalue
of type 'Point'
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

decltype

decltype

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to int
    auto aa = ra;       // ---> deducted to int
    ...
    return 0;
}
```

decltype

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);             // ---> deducted to int
    auto aa = ra;        // ---> deducted to int

    decltype(ra) dra = ra;


    ...
    return 0;
}
```

decltype

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);              // ---> deducted to int
    auto aa = ra;         // ---> deducted to int

    decltype(ra) dra = ra;          // ---> deducted to const int&

    ...
    return 0;
}
```

decltype

Semantics: in general we just want to take a type of the given expression

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to int
    auto aa = ra;       // ---> deducted to int

    decltype(ra) dra = ra;       // ---> deducted to const int&

    ...
    return 0;
}
```

decltype

Semantics: if argument of decltype is a name => the type will be exactly the type of the variable with this name.

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);             // ---> deducted to int
    auto aa = ra;        // ---> deducted to int

    decltype(ra) dra = ra;         // ---> deducted to const int&

    ...
    return 0;
}
```

decltype

Semantics: if argument of decltype is a name => the type will be exactly the type of the variable with this name.

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to int
    auto aa = ra;       // ---> deducted to int

    decltype(ra) dra = ra;        // ---> deducted to const int&
    decltype(ra + 1) dra2 = ra;   // ---> deducted to ???
    ...
    return 0;
}
```

decltype

Semantics: if argument of decltype is a name => the type will be exactly the type of the variable with this name.

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);              // ---> deducted to int
    auto aa = ra;         // ---> deducted to int

    decltype(ra) dra = ra;         // ---> deducted to const int&
    decltype(ra + 1) dra2 = ra;   // ---> deducted to int
    ...
    return 0;
}
```

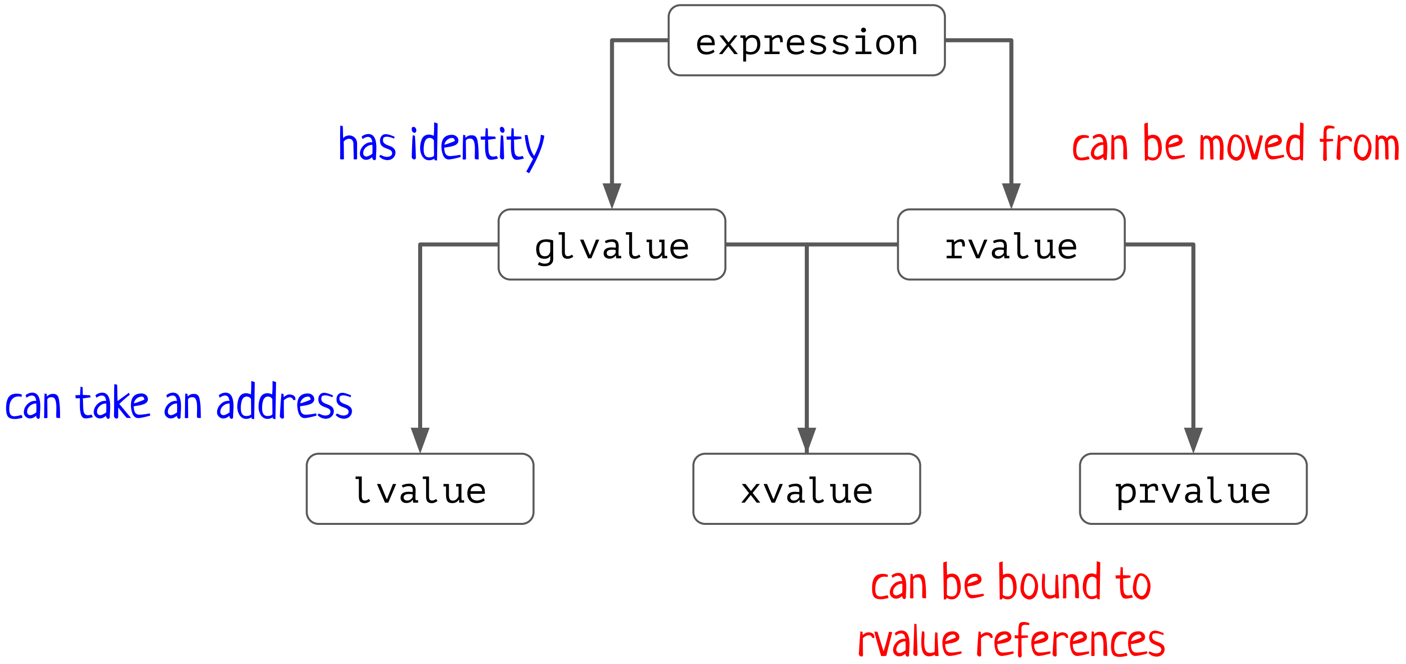
decltype

Semantics: if argument of decltype is a name => the type will be exactly the type of the variable with this name.

if argument is an expression, all depends on the category of this expression

```
template <typename T> void foo(T t);

int main() {
    int a = 10;
    const int& ra = a;

    foo(ra);            // ---> deducted to int
    auto aa = ra;       // ---> deducted to int

    decltype(ra) dra = ra;       // ---> deducted to const int&
    decltype(ra + 1) dra2 = ra;  // ---> deducted to int
    ...
    return 0;
}
```

Value categories



decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                          // ----> deducted to ???
```

decltype

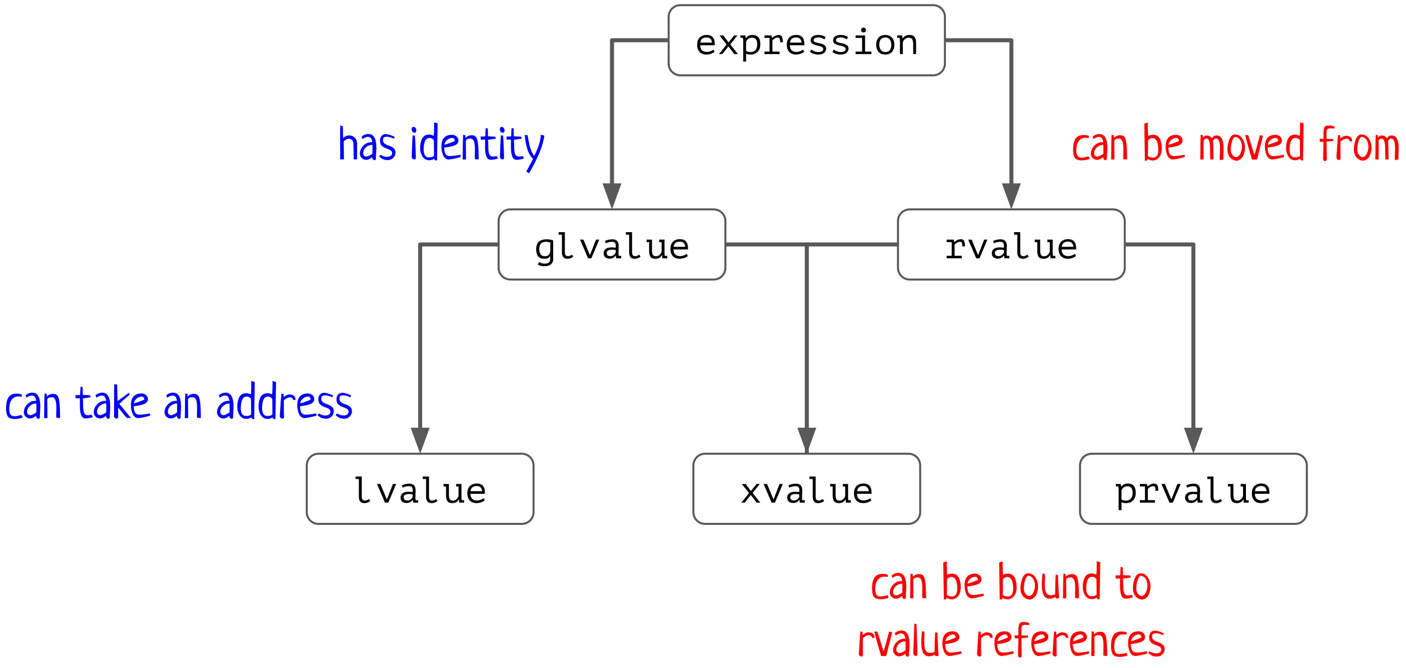
```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                          // ----> deducted to int[3]
```

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                  // ----> deducted to int[3]
decltype(arr[0]) v2                                // ----> deducted to ???
```

To what category
does it belong?

Value categories



decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                    // ----> deducted to int[3]
decltype(arr[0]) v2                                  // ----> deducted to ???
```

To what category
does it belong to?

lvalue!

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                  // ----> deducted to int[3]
decltype(arr[0]) v2                                // ----> deducted to int&
```

To what category
does it belong to?

lvalue!

lvalue => lvalue ref

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                          // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];              // ----> deducted to int&
```

To what category
does it belong to?

lvalue!

lvalue => lvalue ref

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                   // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];                       // ----> deducted to int&
decltype(arr[0] + 1) v3;
```

To what category
does it belong to?

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                    // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];                        // ----> deducted to int&
decltype(arr[0] + 1) v3;
```

To what category
does it belong to?

prvalue!

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                          // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];              // ----> deducted to int&
decltype(arr[0] + 1) v3;                   // ----> deducted to int
```

To what category
does it belong to?

prvalue!

prvalue => just a type

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                   // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];                       // ----> deducted to int&
decltype(arr[0] + 1) v3;                            // ----> deducted to int
decltype(std::move(arr[0])) v4
```

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                    // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];        // ----> deducted to int&
decltype(arr[0] + 1) v3;             // ----> deducted to int
decltype(std::move(arr[0])) v4
```

To what category does it belong to?

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                   // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];       // ----> deducted to int&
decltype(arr[0] + 1) v3;            // ----> deducted to int
decltype(std::move(arr[0])) v4      // ----> deducted to ???
```

To what category does it belong to?

xvalue!

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                           // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];               // ----> deducted to int&
decltype(arr[0] + 1) v3;                    // ----> deducted to int
decltype(std::move(arr[0])) v4              // ----> deducted to int&&
```

To what category does it belong to?

xvalue!

xvalue => rvalue ref

decltype

```
int arr[3] = {1, 2, 3};
decltype(arr) v1;                                    // ----> deducted to int[3]
decltype(arr[0]) v2 = arr[0];                        // ----> deducted to int&
decltype(arr[0] + 1) v3;                             // ----> deducted to int
decltype(std::move(arr[0])) v4 = std::move(arr[0]); // ----> deducted to int&&
```

To what category
does it belong to?

xvalue!

xvalue => rvalue ref

decltype(expr)

General idea: get the type of an argument

Semantics: if argument of decltype is a name => the type will be exactly the type of the variable with this name.

if argument is an expression, all depends on the category of this expression:

- lvalues => decltype will give you lvalue ref
- xvalues => decltype will give you rvalue ref
- prvalues => decltype will give you bare type of expr

decltype(expr)

But why to have such thing as decltype?

auto return types

```
std::vector<int> foo(int n) {
    if (n >0) {
        std::vector<int> result;
        for (int i = 0; i < n; i++) {
            result.push_back(i % 3);
        }
        return result;
    } else {
        return std::vector{1, 2, 3};
    }
}
```

auto return types

Very natural desire: to remove boilerplate code from declaration of return type.

```
auto foo(int n) {
    if (n >0) {
        std::vector<int> result;
        for (int i = 0; i < n; i++) {
            result.push_back(i % 3);
        }
        return result;
    } else {
        return std::vector{1, 2, 3};
    }
}
```

auto return types

```
auto foo(int n) {
    if (n >0) {
        std::vector<int> result;
        for (int i = 0; i < n; i++) {
            result.push_back(i % 3);
        }
        return result;
    } else {
        return std::vector{1, 2, 3};
    }
}
```

Very natural desire: to remove boilerplate code from declaration of return type.

Works well if return values on every return branch are of the same type.

auto return types

```
template <typename T, typename U>
??? baz(T a, U b) {
    std::cout << a + b << std::endl;
    return a + b;
}
```

auto return types

```
template <typename T, typename U>
??? baz(T a, U b) {
    std::cout << a + b << std::endl;
    return a + b;
}
```

The result type is type of expression: a + b!

auto return types

```
template <typename T, typename U>
decltype(a + b) baz(T a, U b) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

The result type is type of expression: a + b!

But this will just not compile (a and b are not yet declared)

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

The result type is type of expression: a + b!

But this will just not compile (a and b are not yet declared)

Solution in C++11 was new syntax.

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) {
    std::cout << a + b << std::endl;
    return a + b;
}
```

Since C++14 we can just use auto.

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability (especially if auto is used in header files)

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
   std::cout << a + b << std::endl;
   return a + b;
}

template <typename T>
auto factorial(T n) {
   return (n > 0) ?
          n * factorial(n - 1) : 1;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

2) fails in deduction

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
   std::cout << a + b << std::endl;
   return a + b;
}

template <typename T>
auto factorial(T n) {
   return (n > 0) ?
          n * factorial(n - 1) : 1;
}
```

```
error: use of 'auto factorial(T)' before
deduction of 'auto'
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

2) fails in deduction

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a + b) {
    std::cout << a + b << std::endl;
    return a + b;
}

template <typename T>
auto factorial(T n) -> decltype(n) {
    return (n > 0) ?
           n * factorial(n - 1) : 1;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

2) fails in deduction

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a) {
   std::cout << a + b << std::endl;
   return a + b;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

2) fails in deduction

3) more precise hints

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a) {
   decltype(a + b) res = a;
   res += ...;
   res += b;
   return res;
}
```

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability

2) fails in deduction

3) more precise hints

auto return types

```
template <typename T, typename U>
auto baz(T a, U b) -> decltype(a) {
   decltype(a + b) res = a;
   res += ...;
   res += b;
   return res;
}
```

So, with decltype you have much more control over the type deduction than with auto.

Since C++14 we can just use auto.

But there are reasons to save decltype as well in modern C++:

1) readability
2) fails in deduction
3) more precise hints

auto arguments types (since C++20)

auto arguments types (since C++20)

```
void bar(auto a, auto b) {
    cout << a + b << endl;
}
```

Do you have any idea
what exactly it is?

auto arguments types (since C++20)

```
void bar(auto a, auto b) {
    cout << a + b << endl;
}
```

```
template <typename T, typename U>
void bar(T a, U b) {
    cout << a + b << endl;
}
```

Do you have any idea what exactly it is?

Just short writing for template arguments 😊

Reference collapsing

Template arguments deduction

- If you do not specify template argument explicitly during function call, compiler will try to deduct it.
- Implicit casts are not taken into account during deduction!
- During this deduction references, const and etc are cut off. The full list of rules is here.
- You can specify that you mean references or pointers, then deduction will take it into account and not cut them off.
- There are special rules for rvalue refs, will discuss them later.

Reference collapsing

```
int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;




}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(a);              // ---> deducted to ???
    auto& lra2 = a;       // ---> deducted to ???

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(a);            // ---> deducted to foo1<int>(int&)
    auto& lra2 = a;     // ---> deducted to int&

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(a);            // ---> deducted to foo1<int>(int&)
    auto& lra2 = a;     // ---> deducted to int&

}
```

So, we’ve just added a lvalue reference to the given type.

Reference collapsing

```cpp
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(lra);            // ---> deducted to ???
    auto& lra2 = lra;     // ---> deducted to ???

}
```

Reference collapsing

If we will try to think in a similar way: just "add a reference" we will be in some trouble. We do not want to have rvalue ref here!

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(lra);              // ---> deducted to foo1<int&>(int& + &)
    auto& lra2 = lra;       // ---> deducted to int& + &

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(lra);          // ---> deducted to foo1<int>(int&)
    auto& lra2 = lra;   // ---> deducted to int&

}
```

If we will try to think in a similar way: just "add a reference" we will be in some trouble. We do not want to have rvalue ref here!

So, references are collapsed => lvalue ref deducted

Reference collapsing

If we will try to think in a similar way: just "add a reference" we will be in some trouble.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);              // ---> deducted to foo1<int&&>(int&& + &)
    auto& lra2 = rra;       // ---> deducted to int&& + &

}
```

Reference collapsing

If we will try to think in a similar way: just "add a reference" we will be in some trouble.

Such type just don't exist! (&&&)

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);              // ---> deducted to foo1<int&&>(int&& + &)
    auto& lra2 = rra;       // ---> deducted to int&& + &

}
```

Reference collapsing

If we will try to think in a similar way: just "add a reference" we will be in some trouble.

Such type just don't exist! (&&&)

So, we have to collapse them again. To what?

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);             // ---> deducted to foo1<int&&>(int&& + &)
    auto& lra2 = rra;      // ---> deducted to int&& + &

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);          // ---> deducted to foo1<int&&>(int&& + &)
    auto& lra2 = rra;   // ---> deducted to int&& + &

}
```

If we will try to think in a similar way: just "add a reference" we will be in some trouble.

Such type just don't exist! (&&&)

So, we have to collapse them again. To what?

Of course to lvalue ref, you've asked for that!

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);            // ---> deducted to foo1<int>(int&)
    auto& lra2 = rra;     // ---> deducted to int&

}
```

If we will try to think in a similar way: just "add a reference" we will be in some trouble.

Such type just don’t exist! (&&&)

So, we have to collapse them again. To what?

Of course to lvalue ref, you’ve asked for that!

Reference collapsing

So, currently we know
2 collapsion rules:

& + & = &
& + && = &

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo1(rra);              // ---> deducted to foo1<int>(int&)
    auto& lra2 = rra;       // ---> deducted to int&

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);             // ---> deducted to ???
    auto&& ura = lra;      // ---> deducted to ???

}
```

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to ???
    auto&& ura = lra;       // ---> deducted to ???

}
```

Naive approach: do the same as with lvalue ref, try to add to ampersands.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);          // ---> deducted to foo2<int>(int& + &&)
    auto&& ura = lra;   // ---> deducted to int& + &&

}
```

Naive approach: do the same as with lvalue ref, try to add to ampersands.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);          // ---> deducted to foo2<int>(int& + &&)
    auto&& ura = lra;   // ---> deducted to int& + &&

}
```

Naive approach: do the same as with lvalue ref, try to add to ampersands.

We can’t have 3 ampersands, so, let’s collapse them! How?

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;       // ---> deducted to int&

}
```

Naive approach: do the same as with lvalue ref, try to add to ampersands.

We can't have 3 ampersands, so, let's collapse them! How?

Reference collapsing

Naive approach failed us.

```cpp
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);          // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;   // ---> deducted to int&

}
```

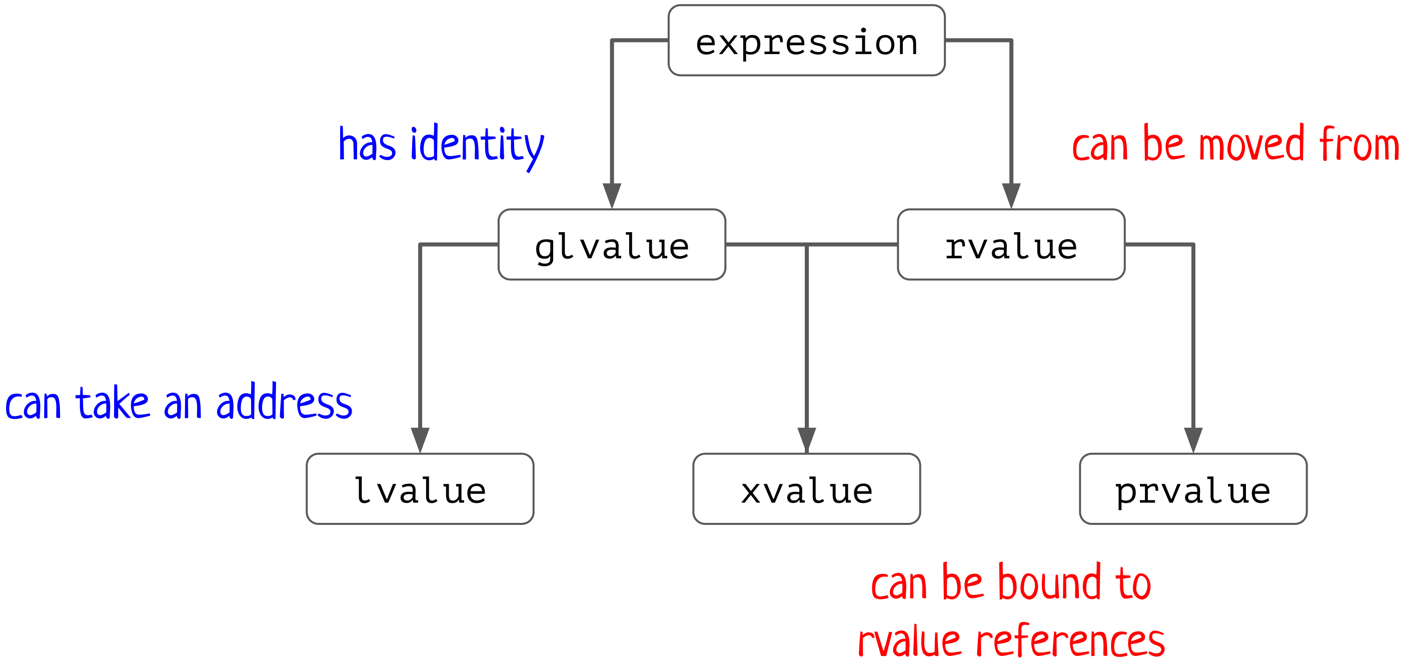
Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);            // ---> deducted to ???
    auto&& ura = lra;     // ---> deducted to ???

}
```

Value categories



Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra;     // ---> deducted to ???

}
```

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra;     // ---> deducted to ???

}
```

this is lvalue expression!

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra;     // ---> deducted to int&

}
```

this is lvalue expression!

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra + 1;     // ---> deducted to ???

}
```

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra + 1;     // ---> deducted to ???

}
```

this is prvalue expression!

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = lra + 1;      // ---> deducted to int&&

}
```

this is prvalue expression!

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = std::move(lra);     // ---> deducted to ???

}
```

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = std::move(lra);     // ---> deducted to int&&

}
```

this is xvalue expression!

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = a;     // ---> deducted to ???

}
```

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```cpp
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = a;      // ---> deducted to ???

}
```

this is lvalue expression!

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = a;     // ---> deducted to int&

}
```

this is lvalue expression!

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = rra;     // ---> deducted to ???

}
```

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = rra;     // ---> deducted to int&

}
```

this is lvalue expression!

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = rra;     // ---> deducted to int&

}
```

this is not rvalue ref

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = rra;     // ---> deducted to int&

}
```

this is universal ref

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;


    auto&& ura = rra;     // ---> deducted to int&

}
```

this is universal ref
(or forwarding ref)

Reference collapsing

`auto& - lvalue reference, of type that will be deducted`

Reference collapsing

`auto&` - lvalue reference, of type that will be deducted

`auto&&` - universal (forwarding ref) of type that will be deducted.

Reference collapsing

`auto&` - lvalue reference, of type that will be deducted

`auto&&` - universal (forwarding ref) of type that will be deducted. Depending on the context it could be either lvalue ref or rvalue ref.

Ranged-base for

So, we still need to find a better way to generalize our code. 🤔

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

for (auto& element: stdv) {
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

```
error: cannot bind non-const lvalue
reference of type 'Point&' to an rvalue
of type 'Point'
```

Now compiler thinks that you know what you are doing, so it is ok.

Can you see any problems with using `auto&` here?

What if `*it` returns a value, not a reference?

Ranged-base for

What if *it returns a value, not a reference?

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

for (auto&& element: stdv) {
   std::cout << element.x << std::endl;
   std::cout << element.y << std::endl;
}
```

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto&& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What if *it returns a value, not a reference?

In such case, auto&& will be considered as rvalue reference (to prvalue).

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto&& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What if *it returns a value, not a reference?

In such case, auto&& will be considered as rvalue reference (to prvalue). No binding error.

(copies will be created, but this is ok)

Ranged-base for

```
Vector<Point> stdv;
stdv.push(Point{13, 42});
stdv.push(Point{1, 2});
stdv.push(Point{5, 10});

auto it = stdv.begin();
auto end = stdv.end();

for (; it != end; ++it) {
    auto&& element = *it;
    std::cout << element.x << std::endl;
    std::cout << element.y << std::endl;
}
```

What if `*it` returns a value, not a reference?

In such case, `auto&&` will be considered as rvalue reference (to prvalue). No binding error.

But if `*it` returned reference => also, not a problem as `auto&&` will be considered as lvalue reference.

Reference collapsing

So, at this point we should stop and think.

What we actually mean by &&?

Actually it would be nice for T&& to become rvalue reference in case of rvalue argument and still be lvalue reference in case of lvalue expression.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to ???
    auto&& ura = lra;       // ---> deducted to ???

}
```

Universal references: template args type

```cpp
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to ???
    auto&& ura = lra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

Actually it is indeed implemented as reference collapsing.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);          // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;   // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument type should be int& + &&.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument type should be int& + &&.
Rules:

```
& + &    = &
& + &&   = &
&& + &   = &
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: template args type

Actually it is indeed implemented as reference collapsing. In case of rvalue T is deducted into just bare type.

The argument should be int&& + &&.

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra + 1);           // ---> deducted to foo2<???>
    auto&& ura = lra + 1;    // ---> deducted to ???

}
```

this is prvalue expression!

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of rvalue T is deducted into just bare type.

The argument should be int&& + &&. Rules:

```
& + &    = &
& + &&   = &
&& + &   = &
&& + &&  = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra + 1);            // ---> deducted to foo2<int>(int&&)
    auto&& ura = lra + 1;     // ---> deducted to int&&

}
```

this is prvalue expression!

Universal references: impl

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);                  // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;           // ---> deducted to int&

    foo2(lra + 1);              // ---> deducted to foo2<int>(int&&)
    auto&& ura2 = lra + 1;      // ---> deducted to int&&
}
```

Universal references: impl

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);                  // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;           // ---> deducted to int&

    foo2(lra + 1);              // ---> deducted to foo2<int>(int&&)
    auto&& ura2 = lra + 1;      // ---> deducted to int&&
}
```

This difference between deducted types of T parameter (int& and just int) is important for further topic: std::forward<T>

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of rvalue T is deducted into just bare type.

The argument should be int&& + &&. Rules:

```
& + &    = &
& + &&   = &
&& + &   = &
&& + &&  = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(std::move(lra));          // ---> deducted to foo2<int>(int&&)
    auto&& ura = std::move(lra);   // ---> deducted to int&&

}
```

this is xrvalue expression!

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument should be int& + &&. Rules:

```
& + &      = &
& + &&     = &
&& + &     = &
&& + &&    = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(a);            // ---> deducted to foo2<???>
    auto&& ura = a;     // ---> deducted to ???

}
```

this is lvalue expression!

Universal references: impl

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(a);            // ---> deducted to foo2<int&>(int&)
    auto&& ura = a;     // ---> deducted to int&

}
```

this is lvalue expression!

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument should be int& + &&. Rules:

```
& + &    = &
& + &&   = &
&& + &   = &
&& + &&  = &&
```

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument should be int& + &&. Rules:

```
& + &     = &
& + &&    = &
&& + &    = &
&& + &&   = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(rra);              // ---> deducted to foo2<int&>(int&)
    auto&& ura = rra;       // ---> deducted to int&

}
```

this is lvalue expression!

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument should be int& + &&. Rules:

```
& + &     = &
& + &&    = &
&& + &    = &
&& + &&   = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(rra);          // ---> deducted to foo2<int&>(int&)
    auto&& ura = rra;   // ---> deducted to int&

}
```

this is not rvalue ref

this is not rvalue ref

Universal references: impl

Actually it is indeed implemented as reference collapsing. In case of lvalue T is deducted into lvalue ref.

So, the argument should be int& + &&. Rules:

```
& + &     = &
& + &&    = &
&& + &    = &
&& + &&   = &&
```

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(rra);          // ---> deducted to foo2<int&>(int&)
    auto&& ura = rra;   // ---> deducted to int&

}
```

this is universal ref (or forwarding ref)

this is universal ref (or forwarding ref)

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

- However, quite fragile:

```
const auto&& x = y;  <-- always rvalue ref, no collaps
```

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We'll see more later.

- However, quite fragile:

```
const auto&& x = y;  <-- always rvalue ref, no collaps

template<typename T> class Vector {
    void emplace(T&& param) { … } <-- always rvalue
}
```

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We'll see more later.

- However, quite fragile:

```
const auto&& x = y;  <-- always rvalue ref, no collaps

template<typename T> class Vector {
    template<typename U>
    void emplace(U&& param) { … } <-- fixed
}
```

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

- However, quite fragile.

- There is a philosophy: AAA (almost always auto)

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

- However, quite fragile.

- There is a philosophy: AAA (almost always auto). And also AAARR (almost always auto rvalue ref).

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

- However, quite fragile.

- There is a philosophy: AAA (almost always auto). And also AAARR (almost always auto rvalue ref).

  - Mean that you should use auto everywhere, where it is possible (compile knows better which type you need)

auto and auto&&

- Universal references are powerful instrument for generalization of your code! We’ll see more later.

- However, quite fragile.

- There is a philosophy: AAA (almost always auto). And also AAARR (almost always auto rvalue ref).

    - Mean that you should use auto everywhere, where it is possible (compile knows better which type you need)

    - Very controversial, many drawbacks and counterexamples.

Takeaways

- C++ like iterators as unified API for iteration over elements
- Templates arguments types deduction
- `auto` is just the same as template arguments types deduction! `decltype` is different
- Universal references and references collapsing