CS100 Lecture 12

References, `std::vector`

Contents

- References
- `std::vector`

References

Declare a reference

A **reference** defines an **alternative name** for an object ("refers to" that object).

Similar to pointers, the type of a reference is `ReferredType &`, which consists of two things:

- `ReferredType` is the type of the object that it refers to, and
- `&` is the symbol indicating that it is a reference.

Example:

```
int ival = 42;
int &ri = ival; // `ri` refers to `ival`.
                // In other words, `ri` is an alternative name for `ival`.
std::cout << ri << '\n'; // prints the value of `ival`, which is `42`.
++ri;           // Same effect as `++ival;`.
```

Declare a reference

```
int ival = 42;
int x = ival;              // `x` is another variable.
++x;                       // This has nothing to do with `ival`.
std::cout << ival << '\n'; // 42
int &ri = ival;            // `ri` is a reference that refers to `ival`.
++ri;                      // This modification is performed on `ival`.
std::cout << ival << '\n'; // 43
```

Ordinarily, when we initialize a variable, the value of the initializer is **copied** into the object we are creating.

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

A reference is an alias

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

```
int ival = 42;
int &ri = ival;
++ri;           // Same as `++ival;`.
ri = 50;        // Same as `ival = 50;`.
int a = ri + 1; // Same as `int a = ival + 1;`.
```

After a reference has been defined, **all** operations on that reference are actually operations on the object to which the reference is bound.

```
ri = a;
```

What is the meaning of this?

A reference is an alias

```cpp
int ival = 42;
int &ri = ival;
++ri;           // Same as `++ival;`.
ri = 50;        // Same as `ival = 50;`.
int a = ri + 1; // Same as `int a = ival + 1;`.
```

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

After a reference has been defined, **all** operations on that reference are actually operations on the object to which the reference is bound.

```cpp
ri = a;
```

- This is the same as `ival = a;`. **It is not rebinding `ri` to refer to `a`.**

A reference must be initialized

```cpp
ri = a;
```

- This is the same as `ival = a;`. **It is not rebinding `ri` to refer to `a`.**

Once initialized, a reference remains bound to its initial object. **There is no way to rebind a reference to refer to a different object.**

Therefore, **references must be initialized.**

References must be bound to *existing objects* ("lvalues")

It is not allowed to bind a reference to temporary objects or literals ${}^{\textcolor{red}{1}}$:

```cpp
int &r1 = 42;     // Error: binding a reference to a literal
int &r2 = 2 + 3; // Error: binding a reference to a temporary object
int a = 10, b = 15;
int &r3 = a + b; // Error: binding a reference to a temporary object
```

In fact, the references we learn today are "lvalue references", which must be bound to *lvalues*. We will talk about *value categories* in later lectures.

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Therefore, there are no "references to references".

```cpp
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int & &rr = ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?

```cpp
int &ri2 = ri;
```

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Therefore, there are no "references to references".

```cpp
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int & &rr = ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?


```cpp
int &ri2 = ri; // Same as `int &ri2 = ival;`.
```

- `ri2` is a reference that is bound to `ival`.
- **Any use of a reference is actually using the object that it is bound to!**

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Pointers must also point to objects. Therefore, there are no "pointers to references".

```
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int &*pr = &ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?

```
int *pi = &ri;
```

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Pointers must also point to objects. Therefore, there are no "pointers to references".

```
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int &*pr = ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?

```
int *pi = &ri; // Same as `int *pi = &ival;`.
```

Reference declaration

Similar to pointers, the ampersand `&` only applies to one identifier.

```
int ival = 42, &ri = ival, *pi = &ival;
// `ri` is a reference of type `int &`, which is bound to `ival`.
// `pi` is a pointer of type `int *`, which points to `ival`.
```

Placing the ampersand near the referred type does not make a difference:

```
int& x = ival, y = ival, z = ival;
// Only `x` is a reference. `y` and `z` are of type `int`.
```

`*` and `&`

Both symbols have many identities!

- In a **declaration** like `Type *x = expr`, `*` is **a part of the pointer type** `Type *`.
- In a **declaration** like `Type &r = expr`, `&` is **a part of the reference type** `Type &`.
- In an **expression** like `*opnd` where there is only one operand, `*` is the **dereference operator**.
- In an **expression** like `&opnd` where there is only one operand, `&` is the **address-of operator**.
- In an **expression** like `a * b` where there are two operands, `*` is the **multiplication operator**.
- In an **expression** like `a & b` where there are two operands, `&` is the **bitwise-and operator**.

Example: Use references in range-`for`

Recall the range-based `for` loops (range-`for`):

```
std::string str;
std::cin >> str;
int lower_cnt = 0;
for (char c : str)
  if (std::islower(c))
    ++lower_cnt;
std::cout << "There are " << lower_cnt << " lowercase letters in total.\n";
```

The range-`for` loop in the code above traverses the string, and declares and initializes the variable `c` in each iteration as if ${}^{\textcolor{red}{2}}$

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i]; // Look at this!
  if (std::islower(c))
    ++lower_cnt;
}
```

Example: Use references in range-`for`

```
for (char c : str)
  // ...
```

The range-`for` loop in the code above traverses the string, and declares and initializes the variable `c` in each iteration as if ${}^{\textcolor{red}{2}}$

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i];
  // ...
}
```

Here `c` is a copy of `str[i]`. Therefore, modification on `c` does not affect the contents in `str`.

Example: Use references in range-`for`

What if we want to change all lowercase letters to their uppercase forms?

```
for (char c : str)
  c = std::toupper(c); // This has no effect.
```

We need to declare `c` as a reference.

```
for (char &c : str)
  c = std::toupper(c);
```

This is the same as

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char &c = str[i];
  c = std::toupper(c); // Same as `str[i] = std::toupper(str[i]);`.
}
```

Example: Pass by reference-to-`const`

Write a function that accepts a string and returns the number of lowercase letters in it:

```
int count_lowercase(std::string str) {
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

To call this function:

```cpp
int result = count_lowercase(my_string);
```

Example: Pass by reference-to-`const`

```cpp
int count_lowercase(std::string str) {
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```cpp
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```cpp
std::string str = my_string;
```

The contents of the entire string `my_string` are copied!

Example: Pass by reference-to-`const`

```cpp
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```cpp
std::string str = my_string;
```

The contents of the entire string `my_string` are copied! Is this copy necessary?

Example: Pass by reference-to-`const`

```cpp
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```cpp
std::string str = my_string;
```

The contents of the entire string `my_string` are copied! This copy is unnecessary, because `count_lowercase` is a read-only operation on `str`.

How can we avoid this copy?

Example: Pass by reference-to-`const`

```cpp
int count_lowercase(std::string &str) { // `str` is a reference.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```cpp
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```cpp
std::string &str = my_string;
```

Which is just a reference initialization. No copy is performed.

Example: Pass by reference-to-`const`

```
int count_lowercase(std::string &str) { // `str` is a reference.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

However, this has a problem:

```
std::string s1 = something(), s2 = some_other_thing();
int result = count_lowercase(s1 + s2); // Error: binding reference to
                                       // a temporary object.
```

`a + b` is a temporary object, which `str` cannot be bound to.

Example: Pass by reference-to-`const`

References must be bound to existing objects, not literals or temporaries.

There is an exception to this rule: References-to-`const` can be bound to anything.

```
const int &rci = 42; // OK.
const std::string &rcs = a + b; // OK.
```

`rcs` is bound to the temporary object returned by `a + b` as if

```
std::string tmp = a + b;
const std::string &rcs = tmp;
```

$\Rightarrow$ We will talk more about references-to-`const` in recitations.

Example: Pass by reference-to-`const`

The answer:

```
int count_lowercase(const std::string &str) { // `str` is a reference-to-`const`.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```
std::string a = something(), b = some_other_thing();
int res1 = count_lowercase(a);       // OK.
int res2 = count_lowercase(a + b);   // OK.
int res3 = count_lowercase("hello"); // OK.
```

Benefits of passing by reference-to-`const`

Apart from the fact that it avoids copy, declaring the parameter as a reference-to-`const` also prevents some potential mistakes:

```cpp
int some_kind_of_counting(const std::string &str, char value) {
  int cnt = 0;
  for (std::size_t i = 0; i != str.size(); ++i) {
    if (str[i] = value) // Ooops! It should be `==`.
      ++cnt;
    else {
      // do something ...
      // ...
    }
  }
  return cnt;
}
```

`str[i] = value` will trigger a compile-error, because `str` is a reference-to-`const`.

Benefits of passing by reference-to-`const`

1. Avoids copy.
2. Accepts temporaries and literals (*rvalues*).
3. The `const` qualification prevents accidental modifications to it.

[Best practice] Pass by reference-to-`const` if copy is not necessary and the parameter should not be modified.

References vs pointers

A reference

- is not itself an object. It is an alias of the object that it is bound to.
- cannot be rebound to another object after initialization.
- has no "default" or "zero" value. It must be bound to an object.

A pointer

- is an object that stores the address of the object it points to.
- can switch to point to another object at any time.
- can be set to a null pointer value `nullptr`.

Both a reference and a pointer can be used to refer to an object, but references are more convenient - no need to write the annoying `*` and `&`.

Note: `nullptr` is ***the*** null pointer value in C++. Do not use `NULL`.

`std::vector`

Defined in the standard library file `<vector>`.

A "dynamic array".

Class template

`std::vector` is a **class template**.

Class templates are not themselves classes. Instead, they can be thought of as instructions to the compiler for *generating* classes.

- The process that the compiler uses to create classes from the templates is called **instantiation**.

For `std::vector`, what kind of class is generated depends on the type of elements we want to store, often called **value type**. We supply this information inside a pair of angle brackets following the template's name:

```cpp
std::vector<int> v; // `v` is of type `std::vector<int>`
```

Create a `std::vector`

`std::vector` **is not a type itself**. It must be combined with some `<T>` to form a type.

```
std::vector v;              // Error: missing template argument.
std::vector<int> vi;        // An empty vector of `int`s.
std::vector<std::string> vs; // An empty vector of strings.
std::vector<double> vd;     // An empty vector of `double`s.
std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s.
                                   // "2-d" vector.
```

What are the types of `vi`, `vs` and `vvi`?

Create a `std::vector`

`std::vector` **is not a type itself**. It must be combined with some `<T>` to form a type.

```
std::vector v;              // Error: missing template argument.
std::vector<int> vi;        // An empty vector of `int`s.
std::vector<std::string> vs; // An empty vector of strings.
std::vector<double> vd;     // An empty vector of `double`s.
std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s.
                                   // "2-d" vector.
```

What are the types of `vi`, `vs` and `vvi`?

- `std::vector<int>`, `std::vector<std::string>`, `std::vector<std::vector<int>>`.

Create a `std::vector`

There are several common ways of creating a `std::vector`:

```
std::vector<int> v{2, 3, 5, 7};     // A vector of `int`s,
                                    // whose elements are {2, 3, 5, 7}.
std::vector<int> v2 = {2, 3, 5, 7}; // Equivalent to ↑

std::vector<std::string> vs{"hello", "world"}; // A vector of strings,
                                    // whose elements are {"hello", "world"}.
std::vector<std::string> vs2 = {"hello", "world"}; // Equivalent to ↑

std::vector<int> v3(10);     // A vector of ten `int`s, all initialized to 0.
std::vector<int> v4(10, 42); // A vector of ten `int`s, all initialized to 42.
```

Note that all the elements in `v3` are initialized to `0`.

- We hate uninitialized values, so does the standard library.

Create a `std::vector`

Create a `std::vector` as a copy of another one:

```
std::vector<int> v{2, 3, 5, 7};
std::vector<int> v2 = v; // `v2`` is a copy of `v`
std::vector<int> v3(v);  // Equivalent
std::vector<int> v4{v};  // Equivalent
```

No need to write a loop!

Copy assignment is also enabled:

```
std::vector<int> v1 = something(), v2 = something_else();
v1 = v2;
```

- Element-wise copy is performed automatically.
- Memory is allocated automatically. The memory used to store the old data of `v1` is deallocated automatically.

C++17 CTAD

"**C**lass **T**emplate **A**rgument **D**eduction": As long as enough information is supplied in the initializer, **the value type can be deduced automatically by the compiler**.

```cpp
std::vector v1{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42);     // vector<int>, deduced from 42 (int)
std::vector v4(10);         // Error: cannot deduce template argument type
```

Size of a `std::vector`

`v.size()` and `v.empty()` : same as those on `std::string` .

```cpp
std::vector v{2, 3, 5, 7};
std::cout << v.size() << '\n';
if (v.empty()) {
  // ...
}
```

`v.clear()` : Remove all the elements.

Append an element to the end of a `std::vector`

`v.push_back(x)`

```cpp
int n;
std::cin >> n;
std::vector<int> v;
for (int i = 0; i != n; ++i) {
  int x;
  std::cin >> x;
  v.push_back(x);
}
std::cout << v.size() << '\n'; // n
```

Remove the last element of a `std::vector`

`v.pop_back()`

Exercise: Given `v` of type `std::vector<int>` , remove all the consecutive even numbers in the end.

Remove the last element of a `std::vector`

`v.pop_back()`

Exercise: Given `v` of type `std::vector<int>` , remove all the consecutive even numbers in the end.


```cpp
while (!v.empty() && v.back() % 2 == 0)
  v.pop_back();
```

`v.back()` : returns the ***reference*** to the last element.

- How is it different from "returning the *value* of the last element"?

`v.back()` and `v.front()`

Return the references to the last and the first elements, respectively.

It is a **reference**, through which we can modify the corresponding element.

```cpp
v.front() = 42;
++v.back();
```

For `v.back()` , `v.front()` and `v.pop_back()` , **the behavior is undefined** if `v` is empty. They do not perform any bounds checking.

Range-based `for` loops

A `std::vector` can also be traversed using a **range-based** `for` **loop**.

```cpp
std::vector<int> vi = some_values();
for (int x : vi)
  std::cout << x << std::endl;
std::vector<std::string> vs = some_strings();
for (const std::string &s : vs) // use reference-to-const to avoid copy
  std::cout << s << std::endl;
```

Exercise: Use range-based `for` loops to count the number of uppercase letters in a `std::vector<std::string>`.

Range-based `for` loops

Exercise: Use range-based `for` loops to count the number of uppercase letters in a `std::vector<std::string>`.

```cpp
int cnt = 0;
for (const std::string &s : vs) { // Use reference-to-const to avoid copy
  for (char c : s) {
    if (std::isupper(c))
      ++cnt;
  }
}
```

Access through subscripts

`v[i]` returns the **reference** to the element indexed `i`.

- `i` $\in[0,N)$, where $N=$ `v.size()`.
- Subscript out of range is **undefined behavior**. `v[i]` performs no bounds checking.
  - In pursuit of efficiency, most operations on standard library containers do not perform bounds checking.
- A kind of "subscript" that has bounds checking: `v.at(i)`.
  - If `i` is out of range, *a* `std::out_of_range` *exception is thrown*.

Feel the style of STL

Basic and low-level operations are performed automatically:

- Default initialization of `std::string` and `std::vector` results in an empty string / container, not indeterminate values.
- Copy of `std::string` and `std::vector` is done automatically, which performs member-wise copy.
- Memory management is done automatically.

Interfaces are consistent:

- `std::string` also has member functions like `.push_back(x)`, `.pop_back()`, `.at(i)`, `.size()`, `.clear()`, etc. which do the same things as on `std::vector`.
- Both can be traversed by range-`for`.

Summary

References

- A reference is an alias.
- A reference is bound to an object during initialization. After that, any use of that reference is actually using the object it is bound to.
- A reference can only be bound to existing objects (*lvalues*). A pointer can only point to existing objects.
  - But a reference-to-`const` can be bound to anything.
- Pass arguments by reference-to-`const`: avoids copy, accepts both lvalues and rvalues, and prevents accidental modification on what should not be modified.

Summary

`std::vector`

- `std::vector` is not a type. It must be combined with some `<T>` to form a type.
- Many ways of creation.
- Copy of a `std::vector` performs member-wise copy.
- `v.size`, `v.empty`, `v.push_back`, `v.pop_back`, `v.clear`, `v[i]`, `v.at(i)`.
- Use range-`for` to traverse a `std::vector`.

Exercises

Write the exercises on page 26, 38, 40 and 43 on your own.

Notes

${}^{\textcolor{red}{1}}$ String literals (`"hello"`) are an exception to this. Integer literals, floating-point literals, character literals, boolean literals and `enum` items are rvalues, but string literals are lvalues. They do live somewhere in the memory.

${}^{\textcolor{red}{2}}$ In fact, the range-`for` uses **iterators**, not subscripts.