CS100 Lecture 12

References, `std::vector`, `new` and `delete`

Contents

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

```
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.

```
ri = a;
```

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

A reference must be initialized

```
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 [1]:

```
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".

```
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?

```
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".

```
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?

```
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 [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 [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:

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

Example: Pass by reference-to-`const`

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

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

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

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

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

Example: Pass by reference-to-`const`

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

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

```
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`

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

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

```
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`

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

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

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

```
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:

```
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.

Notes

[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.

[2] In fact, the range-for uses **iterators**, not subscripts.