CS100 Recitation 9

Dinghao Cheng

May 10, 2022

Contents

Contents

Container ctors

All STL containers support this 4 ctors (`vector` as example):

```
std::vector<Type> c; // default-initialization
std::vector<Type> c(c2); // c2 also vector, with the same <Type>
std::vector<Type> c = {a,b,c,d}; // initializer-list, container size
     determined by # of arguments
std::vector<Type> c(it1,it2); // copy of elements between this 2
    iterators, the type of elements must be consistent with <Type>
```

Only Sequential Containers (except `std::array`):

```
std::vector<Type> c(n); // n elements, value-initialization
std::vector<Type> c(n,value); // n elements of value
```

Contents

Assotiative containers

- std::map, std::set, std::multimap, std::multiset
- each has according unordered version
- For map, often use together with std::pair.
- Typical usage:

```
std::map <Keytype, Valuetype> m;
m.find(key); // return iterator to the element, end() if not found
m.count(key); // return # of elements with key
m[key] = value; // (if not exist, insert!) or update the key
```

For ordered **multiple** assotiative containers,use `lower_bound()` and `upper_bound()` instead of `find()` to find the element with key.
For unordered multiple assotiative containers, bucket is used.

Contents

iterators

- Connection between Containers and Algorithm
- `begin()` `cbegin()` `rbegin()` `crbegin()` `iterator` `const_iterator` `reverse_iterator` `const_reverse_iterator`
- `end()`: the position **after the last element**
- Never save the value of `end()`!
- `insert`/`push`/`erase` may cause iterators to fail!

range for statement

An easy and efficient way to iterate a container

```
for (auto& x : c) {
    // do something
};
```

- c must represent a sequence (i.e. can return iterator `begin()` and `end()`).
- If we want to write to the elements in the sequence, the loop variable x must be a reference type.
- Still, we cannot add/erase elements in range for loop (this may cause iterators to fail!)

Contents

Top-level `const`

- Top-level `const` indicates that an object **itself** is `const` .
- Top-level `const` can appear in **any object type**.
- Since a `reference` is not an object itself (do not have entity, like a ghost), it cannot have top-level `const` .

Low-level const

- Low-level const appears in the **base type** of compound types such as pointers and references.
- The object pointed by the pointer / the object binded to the reference is const .

Low-level const appears in the **base type**

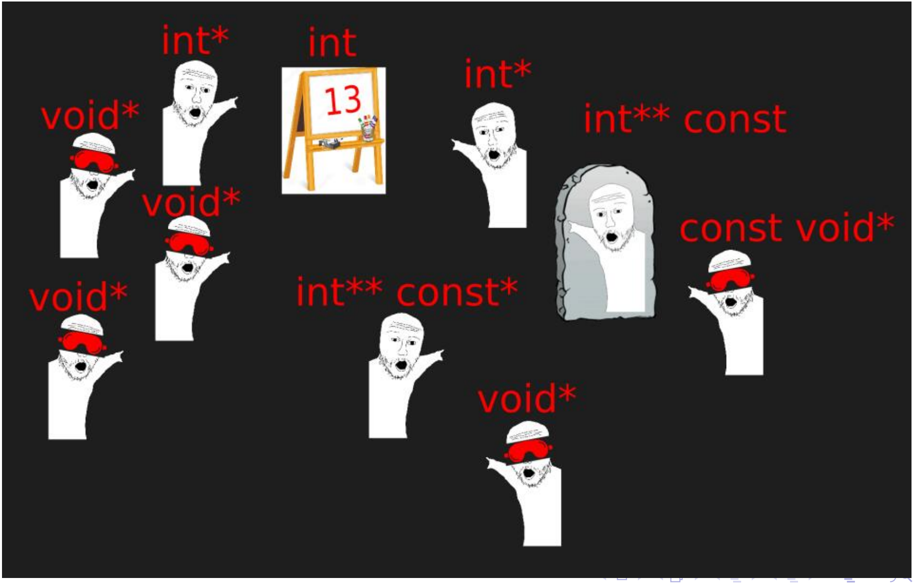
What is the base type?

```
typedef char *pchar;
const pchar ptr1 = NULL;
// a const pointer pointed to char, with top-level const
// the base type is pchar, i.e. char*
const char *ptr2;
// a pointer pointed to const char, with low-level const
// the base type is char
```

Top-level const Versus Low-level const

```
const int a = 1; // Top-level const, a itself is const
const int *const ptr = &a;
// const int: low-level const, the const int pointed by the pointer
    is const
// *const: top-level const, cannot change which int is pointed by
    the pointer
const int* const &p = ptr;
// a reference binded to a pointer with both top-level const and low
    -level const
const int *const *const ptr2 = &ptr;
// have a try?
```

Have some fun



Contents

auto

- auto ordinarily ignores top-level const s.
- As usual in initializations, low-level const s, such as when an initializer is a pointer to const , are kept:
- When auto is initialized with a reference, it will be infered as the type of the object and ignore the reference (variable c below).

```
const int ci = i;
auto &cr = ci; // const int &
auto b = ci; // int (top-level const in ci is dropped)
auto c = cr; // int (cr is an alias for ci whose const is top-level)
auto d = &i; // int* (& of an int object is int*)
auto e = &ci; // const int* (& of a const object is low-level const)
```

Contents

Why const overload?

```
class Array {
public:
    // ctor, dtor, copy ctor, ...
    int &at(std::size_t n);
    const int &at(std::size_t n) const;
    // other members...
private:
    // members...
};
```

- The nonconst version will not be viable for const objects; we can **only** call const member functions on a const object.
- We can call either version on a nonconst object, but the nonconst version will be a **better match**.

Attention: return type

An example

If we write like this...

```
class Vector {
 public:
  int &at(std::size_t n) const {
    return m_data[n];
  }
  // other members
};
```

Example: Access Elements of Vector

```
class Vector {
 public:
  int &at(std::size_t n) const {
    return m_data[n];
  }
  // other members
};
```

Still problematic:

```
const Vector v = some_value();
v.at(10) = 42;
```

Compilers may fail to detect such modification, but it is undefined behavior!

Correct Way

Const overloading.

```
class Vector {
 public:
  int &at(std::size_t n) {
    return m_data[n];
  }
  const int &at(std::size_t n) const {
    return m_data[n];
  }
  // other members
};
```

Calling a `const` member function is actually **adding low-level `const`** to the `this` pointer.

Conclusion

Pointers and references to `const` (with top-level `const`): pointers or references "that think they point or refer to const."

Recall: adding top-level `const` and low-level `const` are both **safe**.

Removing low level `const` is **dangerous**.

Notice

Use const whenever possible. (Effective C++, Item 3)

Contents

Overload

- The most clearly understood one
- Functions(in a name lookup range) with the same name and different argument lists (and optionally different return types)
- Example: ctor

```
class Point2d {
public:
    Point2d(): Point2d(0.0, 0.0) {}
    Point2d(double _x): Point2d(_x, 0.0) {}
    Point2d(double _x, double _y): x(_x), y(_y) {}
private:
    double x, y;
};
```

Contents

Override

- the most clearly defined
- override **virtual functions** in base class
- Example: `override` specifier (Remember, when using virtual function, always write `override` !)
- The `override` keyword lets the compiler check and report if the function is not actually overriding.

```
class base{
public:
    virtual void f() {puts("base::f()");}
};
class derived: public base{
public:
    void f() override {puts("derived::f()");}
};
```

Contents

Overwrite

- Actually hide
- Understand with example

Example 1:

```
class base{ public:
    virtual void f() {puts("base::f()");}
};
class d1: public base{ public:
    void f() override {puts("d1::f()");}
    void f(int a) {puts("d1::f(int)");}
    void f(double a) {puts("d1::f(double)");}
};
class d2: public d1{
};
int main(){
    d2 d;
    d.f(); // d1::f()
    d.f(3); // d1::f(int)
    d.f(3.14); // d1::f(double)
}
```

Example 2:

```
class base{ public:
    virtual void f() {puts("base::f()");}
};
class d1: public base{ public:
    void f() override {puts("d1::f()");}
    void f(int a) {puts("d1::f(int)");}
    void f(double a) {puts("d1::f(double)");}
};
class d2: public d1{ public:
    void f() override {puts("d2::f()");}
};
int main(){
    d2 d;
    d.f();
    d.f(3); //compile error!
    d.f(3.14); //compile error!
}
```

Example 3:

```
class base{ public:
    virtual void f() {puts("base::f()");}
};
class d1: public base{ public:
    void f() override {puts("d1::f()");}
    void f(int a) {puts("d1::f(int)");}
    void f(double a) {puts("d1::f(double)");}
};
class d2: public d1{ public:
    void f() override {puts("d2::f()");}
    using d1::f;
};
int main(){
    d2 d;
    d.f(); // d2::f()
    d.f(3); // d1::f(int)
    d.f(3.14); // d1::f(double)
}
```

Notice

Notice

Never redefine an inherited non-virtual function. (Effective C++, Item 36)

- Recall: polymorphism, virtual function is called according to the type of object rather than the type of pointer.
- Non-virtual functions are statically bound (i.e. according to the type of pointer rather than the object).
- This conflicts with the "is a" relationship in inheritance.

Contents

What does new expression do

```
// new expressions
auto sp = new string("a value"); // allocate and initialize a string
```

1. Calls a library function named **operator** new (or **operator** new []), which allocates raw, untyped memory.
2. Run the appropriate constructor to construct the object(s) from the specified initializers.
3. A pointer to the newly allocated and constructed object is returned.

Recall: default ctor

`new` not only allocates the memory, but also constructs the object (either default-initialize or value-initialize). However:

- Some classes are designed unable to be default-initialized.
- Some classes may contain members that are not default-initializable.
- Problem: What if we want to dynamically allocate memory for these class members?

```
auto p = new MyClass[10](233); // Compile error!
```

Array `new` cannot have initialization arguments.
(Even though, you can use braced initializer list when there is constexpr number of elements.)

placement new

If placement-params are provided, they are passed to the allocation function as additional arguments. Such allocation functions are known as "**placement** new ".

```
new (place_address) type (initializers)
new (place_address) type [size] {braced initializer list}
```

placement new is used to construct objects in allocated storage (even on the stack, e.g. in a union).

Contents

Allocator

- `allocators` allocates unconstructed memory, lets us separate allocation from construction.
- We must `construct` objects in order to use memory returned by `allocate`. Using unconstructed memory in other ways is undefined behavior!
- `std::allocator` object has method `allocate`, `deallocate` (`construct`, `destroy` are deprecated in C++17, they are just encapsulation of placement `new` and dtor call).
- `allocator_traits` instead, is used in most of the STL containers.
- Usage:

std::allocator::allocate()

- Return a pointer to the start address of the allocated memory.
- Compared to directly calling operator `new`, the allocator may use some optimization, such as memory pool in SGI.

```
std::allocator<MyClass> alloc;
auto p = alloc.allocate(10); // actual type: MyClass*
```

std::allocator::construct()

(Deprecated in C++17)
Accept one or multyple arguments which are passed to the ctor of the class.
Actually calling placement `new` .

```
std::allocator<MyClass> alloc;
auto p = alloc.allocate(10); // actual type: MyClass*
auto q = p;
for (int i=0; i<10; ++i){
    // alloc.construct(q++, i);
    new(q++) MyClass(i);
}
```

std::allocator::destroy()

(Deprecated in C++17)
We may destroy only elements that are actually **constructed**!
Actually calling dtor.

```
std::allocator<MyClass> alloc;
auto p = alloc.allocate(10); // actual type: MyClass*
auto q = p;
for (int i=0; i<10; ++i)
    new(q++) MyClass(i);
q = p;
for (int i=0; i<10; ++i)
    // alloc.destroy(q++, i);
    (q++)->~MyClass();
```

std::allocator::deallocate()

The pointer we pass to deallocate cannot be null; it must point to memory allocated by allocate.
Moreover, the size argument passed to deallocate must be the same size as used in the call to allocate that obtained the memory to which the pointer points.

```
std::allocator<MyClass> alloc;
auto p = alloc.allocate(10); // actual type: MyClass*
auto q = p;
for (int i=0; i<10; ++i)
    new(q++) MyClass(i);
q = p;
for (int i=0; i<10; ++i)
    (q++)->~MyClass();
alloc.deallocate(p, 10);
```