# ECE 449/590 – OOP and Machine Learning
# Lecture 22 Design Your Own Vector Class

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 9, 2022

# Outline

# Reading Assignment

- This lecture: Accelerated C++ 11
- Next lecture: Smart Pointers

# Outline

# The Vec Class

```
Vec<int> empty;                       // empty vector
Vec<int> ones(100, 1);                // vector of 100 1's

for (size_t i = 0; i < ones.size(); ++i) {
    ones[i] = i;                 // access elements using []
}

for (Vec<int>::iterator it = ones.begin();
    it != ones.end(); ++it) { // iterators
    std::cout << *it << std::endl;
}
```

- Let's design a class `Vec` to learn how `std::vector` works.
  - As a good example of how resource management should be done <u>in any language</u>.
- In addition to class invariant, class design also requires to decide its interface.
  - So we follow the usual way `std::vector` is used.

# Class Invariant

```
// vec.h
template <class T>
class Vec {
public:
    typedef T value_type;
    Vec();
    Vec(size_t n, const value_type &val);
private:
    value_type *data_;
    size_t n_;
    size_t capacity_;
}; // class Vec<T>
```

- The member `capacity_` indicates the number of elements that can be held in `data_`.
- The member `n_` indicates the number of elements that are currently in `data_`.

# Flexible Memory Management

- We need to keep two kinds of things in the array `data_`.
  - `data_[0]`, ..., `data_[n_-1]` are initialized objects of type `value_type` as the elements.
  - `data_[n_]`, ..., `data_[capacity_-1]` are not objects but bytes on the heap – in other words, they are uninitialized and need to be constructed into objects.
  - That's also the class invariant.
- Clearly `new`/`delete` cannot be used.
- We need language features/library functions to
  - Determine how many memory bytes are required to hold a given number of objects
  - Allocate that many bytes from the heap, uninitialized
  - Construct individual objects on the uninitialized memory
  - Destroy individual objects to make memory uninitialized
  - Deallocate (release to the heap) the uninitialized memory

# Memory Allocation Interface

```
// vec.h
template <class T>
class Vec {
    ...
private:
    ...
    static value_type *allocate(size_t n);
    static void deallocate(value_type *p);
}; // class Vec<T>
```

- `allocate` should return a pointer to a piece of uninitialized memory that can hold `n` `value_type` object.
- `deallocate` should release the uninitialized memory pointed by `p` to the heap.
- Member functions should be declared `static` if they don't depend on any object of the class but are conceptually part of the class.

# Memory Allocation

```
// within class Vec<T>
static value_type *allocate(size_t n) {
    size_t num_of_bytes = sizeof(value_type)*n;
    void *p = ::operator new[](num_of_bytes);
    return (value_type *)p;
}
```

- For template classes, all members should be implemented within the class.
- The expression `sizeof(T)` returns the number of bytes an object of the type `T` consumes.
- The function `operator new[]` is used to allocate memory from the heap.
  - For the standard header `new`
  - When success, it returns a pointer to the memory.
  - Otherwise, it `throws` `std::bad_alloc`.
  - `::` is used to emphasize it is from the global namespace.
- `operator new[]` has no idea what is the type of the objects – the pointer it returns is of type `void *`.
  - We need to convert it to a pointer to `value_type` objects.
  - However, the memory remains uninitialized.

# Memory Deallocation

```
static void deallocate(value_type *p) {
    if (p == nullptr) return;
    ::operator delete[](p);
}
```

- `operator delete[]` is used to release memory to the heap.
- Similarly, there are two functions `operator new` and `operator delete`. Moreover, as you may guess,
  - The expression `new T` will call `operator new` to acquire the memory and then constructs the object.
  - The expression `delete p` will destroy the object pointed by `p` and then call `operator delete` to release the memory.
- They have the same functionality as `malloc` and `free` in C except they throw exceptions instead of returning NULL pointers for failures.

# Memory Initialization Interface

```
private:
    ...
    static void uninitialized_fill(value_type *uninit_b,
        value_type *uninit_e, const value_type &val);
    static void uninitialized_copy(const value_type *from_b,
        const value_type *from_e, value_type *uninit_b);
```

- We follow the standard library to create two `static` member functions for initializing a piece of memory into objects.
- `uninitialized_fill` will fill the uninitialized memory within the range `[uninit_b, uninit_e)` with objects having a value of `val`.
- `uninitialized_copy` will copy the objects within the range `[from_b, from_e)` to the uninitialized memory pointed by `uninit_b`.
    - It is programmers' responsibility to ensure that piece of memory can hold enough number of objects.

# Placement new

```
static void uninitialized_fill(value_type *uninit_b,
    value_type *uninit_e, const value_type &val) {
    for (; uninit_b != uninit_e; ++uninit_b) {
        new (uninit_b) value_type(val);
    }
}
```

- The placement new statement `new (p) T(args);` is used to construct an object of type `T` on a piece of memory pointed by `p`, using arguments `args`.
  - If you feel it's confusing, here is how to memorize the syntax: first, `T(args)` means to construct an object of type `T` using `args`; then `new (p)` means the object should be at the memory pointed by `p`.
- What if one construction `throw`s an exception?
  - Then we have no idea how many objects are actually constructed.
  - Therefore we won't be able to destroy them, potential resource leakage!

# Exception Safety for uninitialized_fill

```
static void uninitialized_fill(value_type *uninit_b,
    value_type *uninit_e, const value_type &val) {
    value_type *init_b = uninit_b;
    try {
        for (; uninit_b != uninit_e; ++uninit_b) {
            new (uninit_b) value_type(val);
        }
    }
    catch (...) {                   // catch all exceptions
        for (; init_b != uninit_b; ++init_b) {
            init_b->~value_type(); // call dtor to destroy the object
        }
        throw;                      // re-throw the exception
    }
}
```

- Postcondition of `uninitialized_fill`: All or None
  - All the objects are initialized if the function returns normally.
  - If some construction `throw`s, then the function will `throw` the same exception and no object is initialized.

# Implement uninitialized_copy

```
static void uninitialized_copy(const value_type *from_b,
    const value_type *from_e, value_type *uninit_b) {
    value_type *init_b = uninit_b;
    try {
        for (; from_b != from_e; ++from_b, ++uninit_b) {
            new (uninit_b) value_type(*from_b);
        }
    }
    catch (...) {                   // catch all exceptions
        for (; init_b != uninit_b; ++init_b) {
            init_b->~value_type(); // call dtor to destroy the object
        }
        throw;                      // re-throw the exception
    }
}
```

- Postcondition of `uninitialized_copy`: All or None

# Implement Constructors and Destructor

```
Vec(): data_(nullptr), n_(0), capacity_(0) {
}
Vec(size_t n, const value_type &val)
    : data_(allocate(n)), n_(n), capacity_(n) {
    try {
        uninitialized_fill(data_, data_+n_, val);
    }
    catch (...) {
        deallocate(data_);
        throw;
    }
}
Vec::~Vec() {
    for (size_t i = 0; i < n_; ++i) {
        data_[i].~value_type();
    }
    deallocate(data_);
}
```

- If a ctor fail, the dtor won't be called automatically, so we need to deallocate the memory.
- The dtor of `value_type` should not `throw`.

# Outline

The Vec Class

Member Functions

Efficient Updates

Copy Control

# Iterators

```
template <class T>
class Vec {
public:
    ...
    typedef value_type *iterator;
    typedef const value_type *const_iterator;
    ...
}; // class Vec<T>
```

- As the elements are stored in an array, pointers can be used as iterators for our container `Vec`.
- `const_iterator`: need a pointer pointing to `const` objects
  - Type of pointers pointing to `const` objects of type `T`: `const T *` or equivalently `T const *`
- Is `T * const` a valid type?
  - Yes, that's a `const` pointer: the pointer cannot be changed, but the object it points to can be changed.

# Member Functions

```
template <class T>
class Vec {
public:
    ...
    size_t size() const {return n_;}
    iterator begin() {return data_;}
    iterator end() {return data_+n_;}
    const_iterator begin() const {return data_;}
    const_iterator end() const {return data_+n_;}
    ...
}; // class Vec<T>
```

- Be aware of `const` and non-`const` member functions

# Operator and Operator Overloading

```
// some source file
for (size_t i = 0; i < ones.size(); ++i) {
    ones[i] = i;              // access elements using []
}
// vec.h
template <class T>
class Vec {
public:
    ...
    value_type &operator[](size_t i) {
        assert(i < n_);
        return data_[i];
    }
    ...
}; // class Vec<T>
```

- The operator `[]` need to be defined for `Vec` so that the expression `ones[i]` makes sense.
- Operator overloading: when evaluating the expression `ones[i]`, the compiler will attempt to translate it into the function call `ones.operator[](i)` .
  - You define what `[]` means for `Vec` by providing that member function, though you won't be able to call it directly.

# Constness and Operators

```
template <class T>
class Vec {
public:
    ...
    const value_type &operator[](size_t i) const {
        assert(i < n_);
        return data_[i];
    }
    ...
}; // class Vec<T>
```

- You should also provide a `const operator[]` for use with `const Vec` objects.

# Outline

# The reserve Function

```
template <class T>
class Vec {
public:
    ...
    void reserve(size_t cap);
    ...
}; // class Vec<T>
```

- We can make `push_back` even more efficient if the user can give us some hints on the number of the elements.
- The member function `reserve` should allocate enough memory to hold at least `cap` objects.

# Implement reserve

```
void reserve(size_t cap) {
    if (cap <= capacity_)
        return; // nothing to do if there is enough memory

    // prepare new memory/objects
    value_type *p = allocate(cap);
    try {
        uninitialized_copy(data_, data_+n, p);
    }
    catch (...) {
        deallocate(p);
        throw;
    }

    // get rid of old objects/memory
    for (size_t i = 0; i < n_; ++i) {
        data_[i].~value_type();
    }
    deallocate(data_);

    // update members
    data_ = p;
    capacity_ = cap;
}
```

# Implement push_back and pop_back

```
void push_back(const value_type &val) {
    if (n_ == capacity_) {
        reserve(std::max(n_+1, n_*2));
    }
    new (data_+n_) value_type(val);
    ++n_;
}

void pop_back() {
    assert(n_ > 0);
    data_[n_-1].~value_type();
    --n_;
}
```

- Each time when the container is full (`n_ == capacity_`), we need to reserve more memory.
  - It has been proved if the capacity is increased by a fixed portion, then on average `push_back` will take $O(1)$ time.
  - Let's double it every time.
- We don't need to worry about exceptions since if something goes wrong, the dtor of `Vec` will take care of the elements and the memory.

# Outline

The Vec Class

Member Functions

Efficient Updates

Copy Control

# Copy (Ctor) and (Copy) Assignment

```
Vec<int> a(100, 0);

Vec<int> b = a; // make a copy

Vec<int> c;
c = a;     // assignment
```

- Copy (Ctor): construct an object as a copy of the existing one
  - For `Vec`, elements should be constructed as a copy of those from the right-hand side (RHS).
- (Copy) Assignment: change an object into a desired value
  - For `Vec`, current elements should be destroyed and then be constructed as a copy of those from RHS.
- Copy $\neq$ Assignment
  - Before copy, the object doesn't exist.
  - Before assignment, the object is initialized.

# Anything wrong?

```
void some_function() {
    Vec<int> a(100, 0);
    Vec<int> b = a; // make a copy
    Vec<int> c;
    c = a;     // assignment
}
```

- The program will compile.
  - The compiler will generate code to handle copy and assignment.
- What's your expectation of the compiler?
  - It is very unlikely the compiler knows the elements are stored on the heap and are managed by using `data_` and `n_`.
- The compiler will simply make a copy of/assign the members.
  - Members of `a`, `b`, `c` will have the same value.
  - `a.data_` will then be deallocated three times in the dtors of `a`, `b`, and `c` when the function returns – undefined behavior!
- We need to redefine copy and assignment into meaningful operations.

# Copy Constructor

```
template <class T>
class Vec {
public
    ...
    Vec(const Vec<T> &rhs);
    ...
}; // class Vec<T>
```

- Copy ctor is a ctor that takes an object of the same type as the parameter.
  - It is called automatically when there is a need for a copy.
- The parameter should be of a reference type.
  - Otherwise passing the argument itself would require a copy, which is not defined yet.
- In most cases, the RHS object shouldn't be changed.
  - Use a `const` reference for the parameter
  - Use a reference if your class design requires you to do so (very rarely)

# Implement Copy Constructor

```
Vec(const Vec<T> &rhs)
    : data_(allocate(rhs.n_)), n_(rhs.n_), capacity_(rhs.n_) {
    try {
        uninitialized_copy(rhs.data_, rhs.data_+n_, data_);
    }
    catch (...) {
        deallocate(data_);
        throw;
    }
}
```

- Protections are per class instead of per object.
  - So it is possible to access the private members of `rhs` in the copy ctor.

# Copy Constructor and Function Calls

```
Vec<int> increment(Vec<int> v) {
    for (size_t i = 0; i < v.size(); ++i) {
        ++v[i];
    }
    return v;
}
void some_function() {
    Vec<int> zeros(100, 0);
    Vec<int> ones = increment(zeros);
}
```

- There are 3 places where a copy of the `Vec` object is made and the copy ctor is called.
  - The argument `zeros` is copied into the parameter `v`.
  - The parameter `v` is copied into the returned result, which is an `Vec` object w/o a name.
  - The returned result is copied into `ones`.
- The C++ compiler may optimize away copy ctor calls as copy ctor is supposed to perform copy only.

# Assignment Operator

```
template <class T>
class Vec {
public
    ...
    Vec<T> &operator=(const Vec<T> &rhs);
    ...
}; // class Vec<T>
```

- Assignment operator `=` can be overloaded.
  - It is called automatically for assignment.
- The parameter could be of any type.
  - We are interested in the RHS object being also a `Vec<T>` object.
  - So there are 3 choices for the type of the parameter: `Vec<T>`, `const Vec<T> &`, `Vec<T> &`.
  - Using `Vec<T> &` is rare. Let's start with `const Vec<T> &`.
- It's a common practice to require `operator=` to return the object itself.
  - In order to support assignments like `a=b=c=d`
  - So the return type must be a reference.

# Implement Assignment

```
Vec<T> &operator=(const Vec<T> &rhs) {
    if (this == &rhs)
        return *this; // nothing to do for self-assignment

    // prepare new memory/objects
    value_type *p = allocate(rhs.n_);
    try {
        uninitialized_copy(rhs_.data_, rhs_.data_+rhs_.n_, p);
    }
    catch (...) {
        deallocate(p);
        throw;
    }
    // get rid of old objects/memory
    for (size_t i = 0; i < n_; ++i) {
        data_[i].~value_type();
    }
    deallocate(data_);
    // update members
    data_ = p; n_ = capacity_ = rhs.n_;

    return *this;
}
```

# Some Ideas on Assignment

- Roughly speaking, assignment is destruction plus copy construction.
- Is there a elegant way to call the dtor and the copy ctor in `operator=` so we don't need to repeat the code?

# The swap Function

```
template <class T>
class Vec {
public
    ...
    void swap(Vec<T> &rhs) {
        std::swap(data_, rhs.data_);
        std::swap(n_, rhs.n_);
        std::swap(capacity_, rhs.capacity_);
    }
    ...
}; // class Vec<T>
```

- It's usually very easy to swap two objects of the same class type: just swap each member.
  - We have seen using references to swap two integers. Actually you can use `std::swap` for the same purpose.
- The `swap` function is so simple that it won't `throw`.

# Assignment as Copy-and-Swap

```
Vec<T> &operator=(const Vec<T> &rhs) {
    if (this == &rhs)
        return *this; // nothing to do for self-assignment

    Vec<T> copy = rhs;
    swap(copy);

    return *this;
}
```

- We don't need to worry about failures due to exceptions in this function anymore.
  - They are taken care of by ctors and dtor.

# Summary and Advice

- The compiler will synthesize the following special member functions for all types:
  - Default ctor if there is no user-defined ctor.
  - Each of dtor, copy ctor, `operator=` if it is not defined by user.
  - An synthesized function will propagate the semantics of the function to each data member.
- The rule of three: if a class owns a kind of resources, e.g. memory on the heap, then dtor, copy ctor, `operator=` should all be provided for proper resource management.
  - Ctors are always necessary for types with non-trivial class invariants.
- Modern C++ introduces the move semantics that further complicates copy and assignment for better performance.
  - We won't be able to cover these features in our lectures.