Chapter 19: Vector, Templates, and Exceptions

Plan for today

- We will talk about:
  - range checking and exceptions
  - resources and exceptions

Range checking: operator[ ]

- We defined the operator[] as

```
double& operator[](int n) // for non-const
{   return elem[n]; }

double operator[](int n) const // for const
{   return elem[n]; }
```

Range checking: operator[ ]

- We defined the `operator[]` as

```
double& operator[](int n) // for non-const
{   return elem[n]; }

double operator[](int n) const // for const
{   return elem[n]; }
```

- Did we check that index `n` is within the vector range?

Range checking: operator[ ]

- We defined the operator[] as

```
double& operator[](int n) // for non-const
{   return elem[n]; }

double operator[](int n) const // for const
{   return elem[n]; }
```

- Did we check that index n is within the vector range? NO!!!

Range checking: at()

- Operator `at()` will have the range check!

```
struct out_of_range
{
    // … class used to report range access errors
}
```

Range checking: at()

- Operator `at()` will have the range check!

```
struct out_of_range
{
   // … class used to report range access errors
}


template <typename T>
T& vector<t>::at(int n) // for non-const vector
{
   if( n < 0 || n >= sz )
      throw out_of_range();
   return elem[n];
}
```

Range checking: at()

- Operator `at()` will have the range check!

```
struct out_of_range
{
   // … class used to report range access errors
}


template <typename T>
T& vector<t>::at(int n) // for non-const vector
{
   if( n < 0 || n >= sz )
      throw out_of_range();
   return elem[n];
}
```

Write the code for the `const vector`

Exception handling

- We use exceptions to report errors
- We must ensure that use of exceptions
  - Doesn’t introduce new sources of errors
  - Doesn’t complicate our code
  - Doesn’t lead to resource leaks

Resource management

- A resource is something that has to be acquired and must be released (explicitly and implicitly) or reclaimed by some “resource manager”

Resource management

- A resource is something that has to be acquired and must be released (explicitly and implicitly) or reclaimed by some “resource manager”
- Examples of resources:
  - Memory
  - Locks
  - File handles
  - Thread handles
  - Sockets
  - Windows

Resource management

- A resource is something that has to be acquired and must be released (explicitly and implicitly) or reclaimed by some “resource manager”
- Examples of resources:
  - Memory
  - Locks
  - File handles
  - Thread handles
  - Sockets
  - Windows

```
void suspicious(int s, int x)
{
   int* p = new int[s];
   // …
   delete[] p;
}
```

Resource management

- A resource is something that has to be acquired and must be released (explicitly and implicitly) or reclaimed by some “resource manager”
- Examples of resources:
  - Memory
  - Locks
  - File handles
  - Thread handles
  - Sockets
  - Windows

When we allocate memory dinamically, we have make sure it is released, but it is not always easy to do.

```
void suspicious(int s, int x)
{
   int* p = new int[s];
   // … p = q...
   delete[] p;
}
```

Resource management

- A resource is something that has to be acquired and must be released (explicitly and implicitly) or reclaimed by some “resource manager”
- Examples of resources:
  - Memory
  - Locks
  - File handles
  - Thread handles
  - Sockets
  - Windows

When we allocate memory dinamically, we have make sure it is released, but it is not always easy to do.

```
void suspicious(int s, int x)
{
   int* p = new int[s];
   // …
   delete[] p;
}
```

When we add exceptions, resource leaks can become common.

Resource management

- Note: if `new` fails to find free-store memory to allocate, it will throw the standard library exception `bad_alloc`.
- The `try … catch` technique woks for this example, but we'll need several `try`-blocks.

Resource management: RAII

- Note: if `new` fails to find free-store memory to allocate, it will throw the standard library exception `bad_alloc`.
- The `try … catch` technique woks for this example, but we'll need several `try`-blocks.
- Check out this example:

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    //...
}
```

- This is better! The resource is acquired by constructor and released by matching destructor – Resource Acquisition is Initialization (RAII)

Resource management: unique_ptr

- In `<memory>` the standard library provides `unique_ptr`
- `unique_ptr` is an object, that holds a pointer, and we can think of it as some kind of pointer (we can use → and * on it)
- `unique_ptr` owns the object pointed to, hence when it is destroyed, it deletes the object it points to.

Resource management: unique_ptr

- In <memory> the standard library provides `unique_ptr`
- `unique_ptr` is an object, that holds a pointer, and we can think of it as some kind of pointer (we can use → and * on it)
- `unique_ptr` owns the object pointed to, hence when it is destroyed, it deletes the object it points to.
  - If an exception is thrown while a vector is being filled
  - If we return prematurely from a function that “builds” a vector
  - The vector will be properly destroyed

Resource management: unique_ptr

- In <memory> the standard library provides `unique_ptr`
- `unique_ptr` is an object, that holds a pointer, and we can think of it as some kind of pointer (we can use → and * on it)
- `unique_ptr` owns the object pointed to, hence when it is destroyed, it deletes the object it points to.
  - If an exception is thrown while a vector is being filled
  - If we return prematurely from a function that “builds” a vector
  - The vector will be properly destroyed
- The `release()` method of `unique_ptr` object extracts the contained pointer (so we can return it, for example) and makes the object hold the `nullptr`.

Resource management: unique_ptr

- Example (traditional, error-prone approach):

```
vector<int>* make_vec()      // make a filled vector
{
    // allocate on free store
    vector<int>> p {new vector<int>};
    // … fill the vector with data; this may throw an exception …
    return p;
}

// users have to remember to delete
// they occasionally forget: leak!
```

Resource management: unique_ptr

- Example (improved approach):

```
unique_ptr<vector<int>> make_vec()     // make a filled vector
{
    // allocate on free store
    unique_ptr<vector<int>> p {new vector<int>};
    // … fill the vector with data; this may throw an exception …
    return p;
}

// users don't have to delete; no delete in user code
// a unique_ptr owns its object and deletes it automatically
```

Resource management: make_unique

- Example (even better solution):

```
unique_ptr<vector<int>> make_vec()      // make a filled vector
{
    // allocate on free store
    auto p = make_unique<vector<int>>();
    // … fill the vector with data; this may throw an exception …
    return p;
}

// no new in user code
// make_unique is available starting from C++ 14
```

A last glance at our vector class

- Things we didn’t do:
  - `resize()` method doesn’t check if new size is larger than the previous one
    - We hope that since it is a private method, it will only be used by methods of vector class, which we define
  - no insertion operation
  - no deletion at position operation
  - cannot use range for loop
  - ...many more issues

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook