Chapter 19: Vector, Templates, and Exceptions

Plan for today

- We will talk about:
  - what we did so fat with our `vector class`
  - Changing the size
  - templates
  - range checking and exceptions
  - resources and exceptions

Vector class – what we have so far

```
class vector {
    int sz; // the size
    double* elem;// a pointer to the elements

public:
    vector(int s);  // constructor
    vector(std::initializer_list<double> lst); // initializer-list constructor
    vector(const vector& other); // copy constructor
    vector(vector&& a);  // move constructor
    ~vector();       // destructor
    int size() const;              // the current size
    void resize(int newSz); // resizes to new size, copies the existing elements

    vector& operator=(const vector& other); // overloading the assignment operator, with chaining a = b = c
    double& operator[](int n); // for non-const vectors
    double operator[](int n) const; // for const vectors
};
std::ostream& operator<<(std::ostream& out, const vector& v); // overload opeartor<<
```

Changing the size: resize() and push_back()

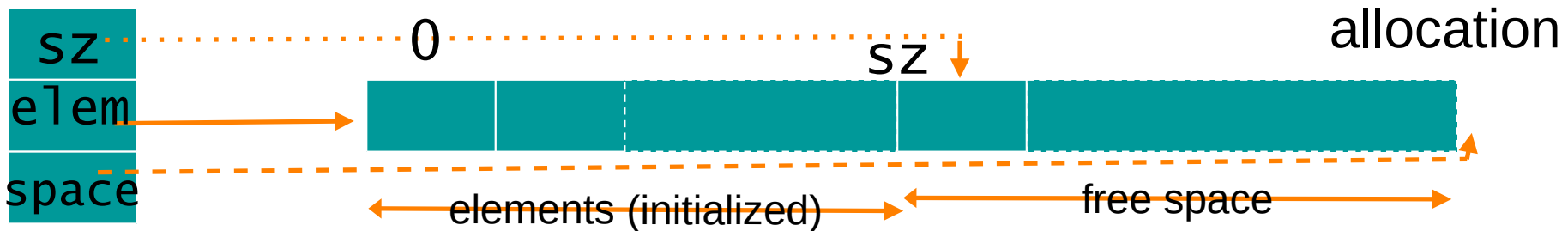
- A problem:
  - we want to have the ability to add elements to the vector

```
vector v1(7);
//… possibly initialize all 7 elements with some values
v1.push_back(20.5); // add the 8th element with value of 20.5
```

  - quite often if we see that the `push_back()` operation is used once, it maybe used more times.
  - Let's optimize our code to anticipate changes in size

Changing the size → space (capacity)

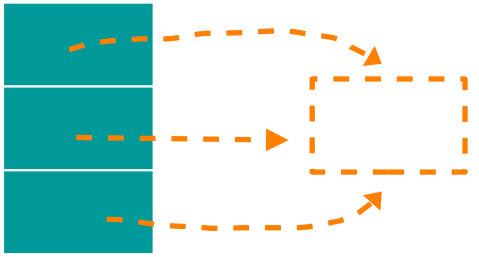
```
class vector {
    int sz;
    double* elem;
    int space;    // number of elements plus “free space”
public:
    // …
};
```



Vector class: updates

```
// Default constructor:
vector()
  : sz{0}, elem{nullptr}, space{0}
{}
```

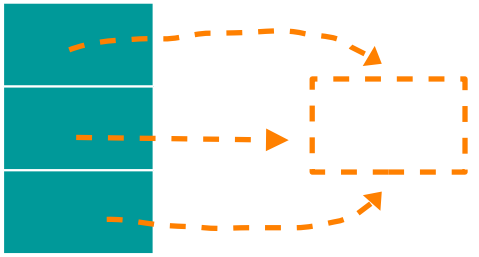
An empty vector
(no free store use):



Vector class: updates

```
// Default constructor:
vector()
  : sz{0}, elem{nullptr}, space{0}
{}
```

An empty vector
(no free store use):



```
// constructor
vector::vector(int s)
    : sz(s), elem{ new double[s] }, space{ s }
{
    for (int i{ 0 }; i < s; ++i) elem[i] = 0;
}
```

Vector class: updates

```
// we have a size function, let's add capacity:
int size() const
{
    return sz; // the current size
}
```

Vector class: updates

```
// we have a size function, let's add capacity:
int size() const
{
    return sz; // the current size
}

int capacity() const
{
    return space;
}
```

Vector class: updates

- We need to look through all the methods we defined to see if any changes must be done due to introduction of the space attribute.
- At copy: we will not copy the free space, as we have no idea how the new vector is going to be used.
- At move: we will grab all

Changing the size: resize() and push_back()

- `resize()` function we already have, but it needs modification:

Changing the size: resize() and push_back()

- `resize()` function we already have, but it needs modification:

```
/* resizes the vector to the new size, preserving all the existing  elements */
void vector::resize(int newSz) {
    double* tmp = new double[newSz]; // allocate new space
    for (int i{ 0 }; i < sz; ++i) // copy the existing values
        tmp[i] = elem[i];
    for (int i{ sz }; i < newSz; ++i) // the rest of the places are filled with 0s
        tmp[i] = 0;
    delete[] elem; // release the space pointed to by elem
    elem = tmp; // reassign the elem pointer to the newly allocated array
    sz = newSz; // update the size
}
```

Changing the size: resize() and push_back()

- `resize()` function we already have, but it needs modification:

```
/* resizes the vector to the new size, preserving all the existing  elements */
void vector::resize(int newSz) {
    double* tmp = new double[newSz]; // allocate new space
    for (int i{ 0 }; i < sz; ++i) // copy the existing values
        tmp[i] = elem[i];
    for (int i{ sz }; i < newSz; ++i) // the rest of the places are filled with 0s
        tmp[i] = 0;
    delete[] elem; // release the space pointed to by elem
    elem = tmp; // reassign the elem pointer to the newly allocated array
    sz = newSz; // update the size    size doesn't change, space does!
}
```

Changing the size: resize() and push_back()

- `resize()` function we already have, but it needs modification:

```
/* resizes the vector to the new size, preserving all the existing  elements */
void vector::resize(int newSz) {
    double* tmp = new double[newSz]; // allocate new space
    for (int i{ 0 }; i < sz; ++i) // copy the existing values
        tmp[i] = elem[i];
    for (int i{ sz }; i < newSz; ++i) // the rest of the places are filled with 0s
        tmp[i] = 0;
    delete[] elem; // release the space pointed to by elem
    elem = tmp; // reassign the elem pointer to the newly allocated array
    space = newSz; // update the size    size doesn't change, space does!
}
```

Changing the size: resize() and push_back()

- `push_back()` function considers three cases:
  - The vector is empty (let's start with the space for 8 elements)
  - The vector is full (no free space left), let's double the space
  - All is good, there is enough space to add another value

Changing the size: resize() and push_back()

- `push_back()` function considers three cases:
  - The vector is empty (let's start with the space for 8 elements)
  - The vector is full (no free space left), let's double the space
  - All is good, there is enough space to add another value

```
void push_back(double d) {
    if(space == 0)  { resize(8); }
    else if (space == sz)  { resize(2 * space); }
    elem[sz] = d; // add d at the end
    ++sz; // increment the size
}
```

Changing the size: resize() and push_back()

- `push_back()` function considers three cases:
  - The vector is empty (let's start with the space for 8 elements)
  - The vector is full (no free space left), let's double the space
  - All is good, there is enough space to add another value

```
void push_back(double d) {
    if(space == 0)  { resize(8); }
    else if (space == sz)  { resize(2 * space); }
    else {elem[sz] = d; // add d at the end
    ++sz; // increment the size }
}
```

Be careful!

The changed and highlighted code is incorrect.

Do you see why?

Changing the size: resize() and push_back()

- grab the files **vector.h** and **vectorTesting.cpp** from our website or Blackboard,
- create a project, add these two files
- compile and run the project.

Templates

- But we don’t just want vector of double
- We want vectors with element types we specify
  - `vector<double>`
  - `vector<int>`
  - `vector<Month>`
  - `vector<Record*>  // vector of pointers`
  - `vector<vector<Record>> // vector of vectors`
  - `vector<char>`
- We must make the element type a parameter to vector
- `vector` must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler; we can define our own parameterized types, called “templates”

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (e.g., hard real time and numerics)
    - Used where flexibility is essential (e.g., the C++ standard library)

Templates

- Template definitions

```
template<class T, int N>
class Buffer { /* … */ };

template<class T, int N>
void fill(Buffer<T,N>& b)
{ /* … */  }

template<typename T>
class vector
{ /* … */};
```

Templates

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf; // for buf, T is char and N is 1024
```

```
template<class T, int N>
class Buffer { /* … */ };
```

Templates

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf; // for buf, T is char and N is 1024
// for a function template, the compiler deduces the template arguments:
fill(buf);     // for fill(), T is char and N is 1024; that's what buf has
```

```
template<class T, int N>
class Buffer { /* … */ };
```

```
template<class T, int N>
void fill(Buffer<T,N>& b)
{ /* … */  }
```

Templates

- Template specializations (instantiations)

```
// for vector class template
vector<int> v(9); // v is a vector of 9 integers, T is int
vector<double> v_d;  // v_d is an empty vector of double, T is double
vector<char* > v_cp{nullptr}; // v_cp is a vector of pointers
// to double, has one pointer, nullptr, now, T is char*
```

```
template<typename T>
class vector
{ /* … */};
```

Vector class – vector<double>

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
    int space;
public:
    vector(int s);  // constructor
    vector(std::initializer_list<double> lst); // initializer-list constructor
    vector(const vector& other); // copy constructor
    vector(vector&& a);  // move constructor
    ~vector();       // destructor
    int capacity(); // return the space (capacity)
    int size() const;             // the current size
    void resize(int newSz); // resizes to new size, copies the existing elements

    vector& operator=(const vector& other);  // overloading the assignment operator, with chaining a = b = c
    double& operator[](int n); // for non-const vectors
    double operator[](int n) const; // for const vectors
    void push_back(double d); // add d to the end
};
std::ostream& operator<<(std::ostream& out, const vector& v); // overload opeartor<<
```

Vector class – vector<double>

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
    int space;
public:
    vector(int s);  // constructor
    vector(std::initializer_list<double> lst); // initializer-list constructor
    // .. skipped
    double& operator[](int n); // for non-const vectors
    double operator[](int n) const; // for const vectors
    void push_back(double d); // add d to the end
};
```

Vector class – vector<T>

```
class vector {
    int sz; // the size
    T* elem; // a pointer to the elements
    int space;
public:
    vector(int s);  // constructor
    vector(std::initializer_list<T> lst); // initializer-list constructor
    // .. skipped
    T& operator[](int n); // for non-const vectors
    T operator[](int n) const; // for const vectors
    void push_back(T d); // add d to the end
};
```

Vector class – vector<T>

```
template<typename T>
class vector {
    int sz; // the size
    T* elem; // a pointer to the elements
    int space;
public:
    vector(int s);  // constructor
    vector(std::initializer_list<T> lst); // initializer-list constructor
    // .. skipped
    T& operator[](int n); // for non-const vectors
    T operator[](int n) const; // for const vectors
    void push_back(T d); // add d to the end
};
```

Vector class – vector<T>

```
// constructor
template<typename T>
Vector<T>::vector(int s)
    : sz(s), elem{ new T[s] }, space{ s }
{
    for (int i{ 0 }; i < s; ++i) elem[i] = 0;
}
```

Vector class – vector<T>

```
// initializer-list constructor
template<typename T>
Vector<T>::vector(std::initializer_list<T> lst)
    : sz(lst.size()), elem{ new T[sz] }, space{ sz }
{
    std::copy(lst.begin(), lst.end(), elem);
}
```

Vector class – vector<T>

```
// helper-method,
// used in copy constructor and assignment operator=
template<typename T>
void vector<T>::copy(const vector<T>& other) {
    this->sz = other.sz;
    space = sz; // *** added for lecture 19 ***
    // or sz = other.sz
    elem = new T[sz];
    // copying over the elements of the other vector
    for (int i{ 0 }; i < sz; ++i)
        elem[i] = other.elem[i];
}
```

Vector class – vector<T>

What other methods were modified:

- where `vector` is mentioned
  - needs to be replaced with `vector<T>`, and
- where `double` is mentioned
  - needs to be replaced with `T`

- copy constructor
- move constructor
- copy assignment
- move assignment
- ...

Look at the files **vectorT.h** and **testvectorT.cpp**

A simple template function: addition of two operands

Consider a simple function add, that adds two integers passed as parameters, and returns the result:

```
int add(const int& a,const int& b) {
   return a + b;
}
```

A simple template function: addition of two operands

Consider a simple function add, that adds two integers passed as parameters, and returns the result:

```
int add(const int& a,const int& b) {
   return a + b;
}
```

Let's "convert" it to a template function.

Appendix:
Other modifications that were introduced to vector class:

- Set and get methods were removed,
- the line in the overloaded `operator<<`

```
out << v.get(i) << " ";
```

was replaced with

```
out << v[i] << " ";
```

Resources used for these slides

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

- Class textbook