TDDE18 & 726G77

Interface, command line and vector

interface

- An interface is an abstract type that is used to specify behavior that concrete classes must implement.
- Interfaces are used to encode similarities which the classes of various types share, but do not necessarily constitute a class relationship.
- Give the ability to use an object without knowing its type of class, but rather only that it implements a certain interface.
- Used a lot in programming language like Java and C#

interface

Below are the nature of *interface* and its C++ equivalents:

- interface can contain only body-less abstract methods; C++ equivalent is pure virtual functions.
- interface can contain only static final data members; C++ equivalent is static const data members which are compile time constants.
- Multiple interface can be implemented by a Java class, this facility is needed because a Java class can inherit only 1 class; C++ supports multiple inheritance straight away with help of virtual keyword when needed.

interface

```
class IList {
    void insert(int number) = 0;
    void remove(int index) = 0;
    static const string name{“List interface”};
};
```

Dynamic type control using typeid

- One way to find out the type of an object is to use *typeid*

```
if (typeid(*p) == typeid(Bat)) ...
```

- A *typeid* expression returns a *type_info* object (a class type)
- type checking is done by comparing two type_info objects

typeid expressions

```
typeid(*p) // p is a pointer to an object of some type

typeid(r) // r is a reference to an object of some type

typeid(T) // T is a type

typeid(p) // is usually a mistake if p is a pointer
```

typeinfo operations

== check if two type_info objects are equal

typeid(*p) == typeinfo(T)

!= check if two type_info objects are not equal

typeid(*p) != typeinfo(T)

name() returns the type name as a string – may be an internal name used by the compiler, a “mangled name”

TDDE18 & 726G77

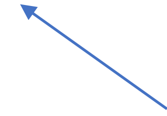
Vector

Vector

- Vector are sequence containers.
- Vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets.
- Vector can change size and capacity, in contrast to array which size is fixed.
- Very efficient in accessing its elements and relatively efficient adding or removing elements from its end.

Visualizing Vectors

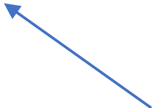
vector<T> v{7};



Datatype vector

Visualizing Vectors

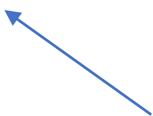
```
vector<T> v{7};
```



Name

Visualizing Vectors

vector<T> v{7};



Size

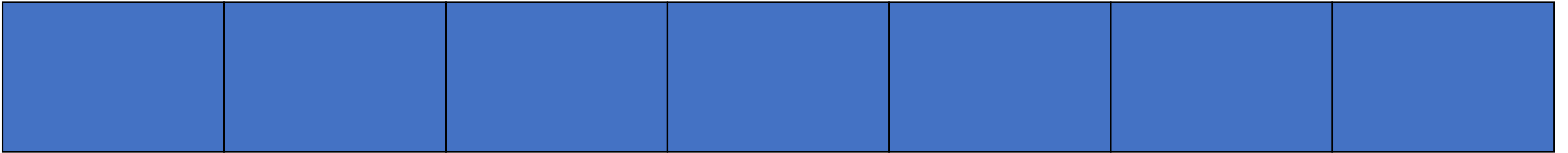
Visualizing Vectors

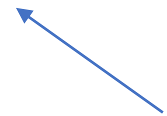
```
vector<T> v{7};
```

Templated argument

Visualizing Vectors

vector<T> v{7};

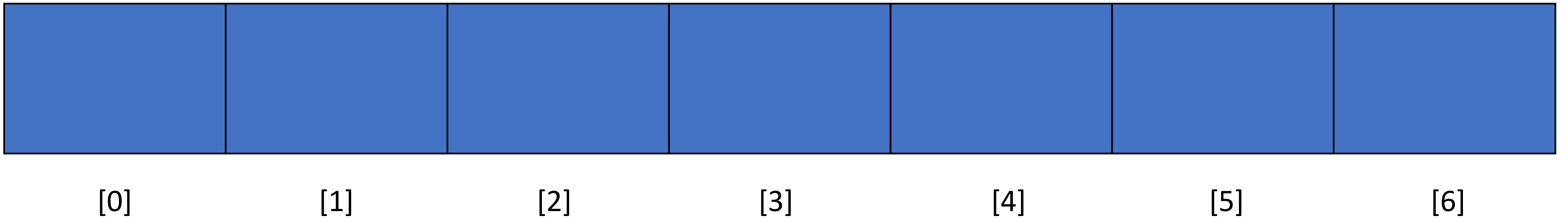




Element

Visualizing Vectors

vector<T> v{7};



- Vectors are 0 indexed

Visualizing Vectors

```
vector<double> v{7};
```

- Every element in this vector is of type double
- The size of this vector are 7
- Constructing vectors with a given size will default initialize the elements

Vector member functions

```
vector<double> v{7};
v[0] = 1;
v.at(1) = 2;
v.front();                          // 1
v.back();                           // 0
v.push_back(5);
v.back();                           // 5
v.size();                           // 8
v.pop_back();                       // remove the 5
```

auto

- When declaring variables in block scope, in initialization statements of for loops, etc., the keyword auto may be used as the type specifier.
- The compiler determines the type that will replace the keyword auto.
- auto may be accompanied by modifiers, such as const or &, which will participate in the type deduction.

```
auto i{5};                  // i will be of type int
auto i{5.0};                // i will be of type double
auto b_ptr{new Bat{}};      // b_ptr will be of type pointer to bat
```

Using the vector

```
vector<int> v;
...
for (int i{0}; i < v.size(); i++) {
    // do something with v.at(i)
}
```

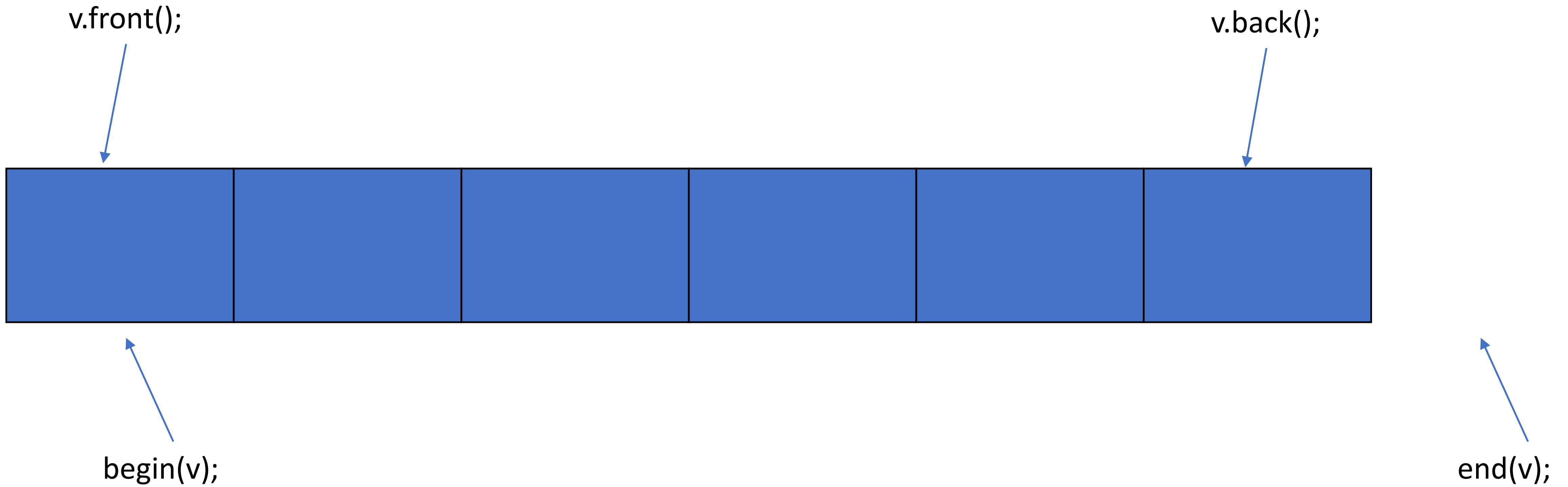
Using the vector

```
vector<int> v;
...
for (auto i{0}; i < v.size(); i++) {
    // do something with v.at(i)
}
```

Using the vector

```
vector<int> v;
...
for (auto it{begin(v)}; it != end(v); it++) {
    // do something with *it
    // (it is almost the same thing as pointer)
}
```

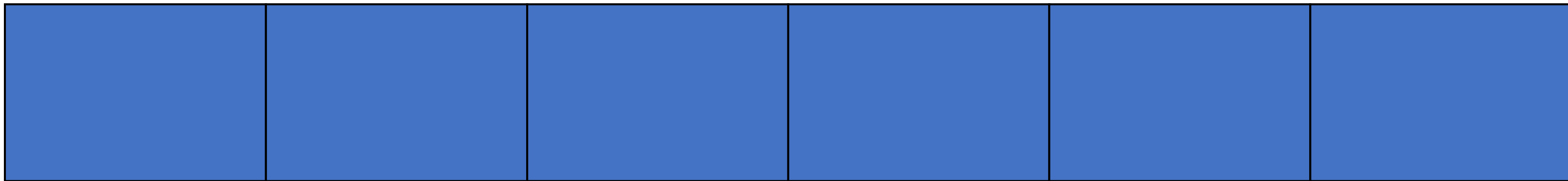
Vector – recap



Vector – recap

`begin(v)` returns a pointer to the element at index 0

`begin(v) + 1` returns a pointer to the element at index 1



begin(v) + 1;

v.push_back(5);

```
v.insert(begin(v) + 1, 3);
```

Vector – erase

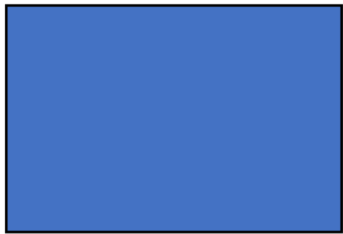
Vector’s erase takes an iterator as a argument. This argument tells the function where to erase in the vector.

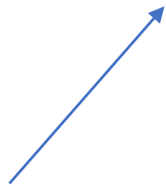
begin(v) + 1;

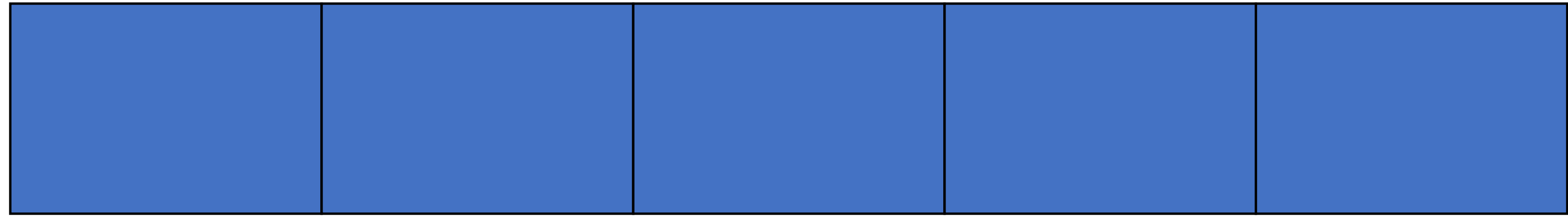
```
v.erase(begin(v) + 1);
```

Vector – erase

When using insert, everything will be moved one index down







Erased element

```
v.erase(begin(v) + 1);
```

For-loops

```
vector<int> v{1, 2, 3, 4};

for (auto i : v) {
    cout << i << “ “;
}
```

For-loops

- The simplify for-loop is just a syntactic sugar for the programmer to use. The compiler will rewrite it during compile time to

```
int i;
for (auto it{begin(v)}; it != end(v); it++) {
    i = *it;
    cout << i << " ";
}
```

auto

```
auto i{5};                  // i will be of type int
auto i{5.0};                // i will be of type double
auto i_ptr{new int{}};      // b_ptr will be of type pointer to int
auto it{begin(v)};          // it will be of type pointer to elements in v
```

auto

- auto can also be used in a function declaration to indicate that the return type will be deduced from the operand of its return statement.

```
auto foo() {     // auto will be deduced to int
    return 1;
}

auto foo() {     // auto will be deduced to double
    return 1.5;
}

auto foo() {      // auto will be deduced to vector<int>
    return vector<int>{5};
}
```

Command line arguments

- Send arguments to our program when starting from the command line

./a.out 10 20 30

Here we send the arguments 10 20 30 to the main function

Command line arguments

./a.out 10 20 30

argc: 4 arguments

argv[0]: ./a.out

argv[1]: 10

argv[2]: 20

argv[3]: 30

Command line arguments

```
int main(int argc, char* argv[]) {
    ...
}
```

```
argc: The amount of arguments sending in
argv: The arguments as an array of strings
```

Command line arguments

```
int main(int argc, char* argv[]) {
    cout << argv[1] << argv[2] << endl;
}

./a.out 10 20

prints: 10 20
```

Type conversion of argv

- Command line arguments are of data type string. To change datatype we use

- stoi – to convert to int
- stod – to convert to double
- stof – to convert to float