02393 Programming in C++

Module 5: Libraries and Interfaces

Alceste Scalas `<alcsc@dtu.dk>`

28 September 2021

Course plan

Module no.	Date	Topic	Book chapter*
0 and 1	31.08	Welcome & C++ Overview	1
2	07.09	Basic C++ and Data Types	1, 2.2 – 2.5
3	14.09	*LAB DAY*	*C++ Practice*
4	21.09	Data Types	2
4	21.09	Libraries and Interfaces	3
5	28.09	Libraries and Interfaces	3
6	05.10	Classes and Objects	4.1, 4.2 and 9.1, 9.2
7	12.10	Templates	4.1, 11.1
		Autumn break	
8	26.10	Inheritance	14.3, 14.4, 14.5
9	02.11	**Guest lecture** & *LAB DAY*	*Previous exams*
10	09.11	Recursive Programming	5
11	16.11	Linked Lists	10.5
12	23.11	Trees	13
13	30.11	Conclusion & *LAB DAY*	*Exam preparation*
	05.12	**Exam**	

* Recall that the book uses some ad-hoc libraries (e.g., for strings and vectors). We will use standard libraries

Outline

A recap from the first 3 lectures

- **The structure of a C++ program**
  - `#include` and `#define` directives, the `main` function, user-defined functions
- **Simple input/output**
  - `cin`, `cout`
- **Variables, values, and types**
  - `string`, `int`, `double`, `float`, arrays (statically and dynamically allocated), pointers, `enum`, `struct`
- **Expressions**
  - Some numeric and boolean operators and math functions, conditional expressions
- **Statements**
  - `if`, `while`, `for`, `switch`

STL (Standard Template Library)

STL is a C++ library of **container classes and algorithms**

Containers are collections of elements. Examples:

- unordered collections: `set`, `mset`
- array-like collections: `vector`, `list`, `array` *(not the built-in arrays we already know!)*
- other ordered collections: `queue`, `stack`
- dictionaries: `map`, `multimap`

It is important to know **how to deal with STL containers** and **choose the right one**

- more than one class of containers may do the job...
- ...but some may do the job better (e.g., faster)

STL `vector`: motivations

Arrays are fundamental data types in many programming languages, but in C++...

- they are **difficult/impossible to resize**
- **insertion and deletion** can be difficult and slow
- you have to **keep track of their actual size**
- you have to be careful to **index within the array bounds**

STL `vector`: motivations

Arrays are fundamental data types in many programming languages, but in C++...

- they are **difficult/impossible to resize**
- **insertion and deletion** can be difficult and slow
- you have to **keep track of their actual size**
- you have to be careful to **index within the array bounds**

The STL `vector` class solves all of these problems!

Examples and documentation:

`http://www.cplusplus.com/reference/stl/vector/`
`http://en.cppreference.com/w/cpp/container/vector`

`vector`: declaration

To use STL `vector`s, you need to include their interface (header file):

```
#include <vector>
```

The type `vector<X>` is a container of elements of **base type** `X`

- `vector<int>` is a vector whose elements are `int`egers
- `vector<double>` is a vector whose elements are `double`s
- `vector< vector<int> >` is a vector whose elements are vectors of `int`egers
- ...

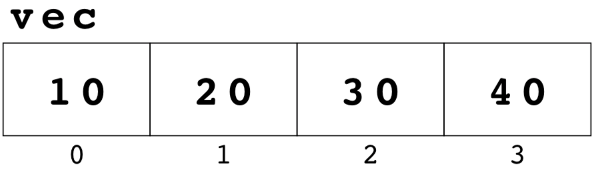
To declare a new empty `vector` object that can contain `int`egers:

```
vector<int> vec;
```

(Note: there are also other ways (*constructors*) to create vectors. . .)

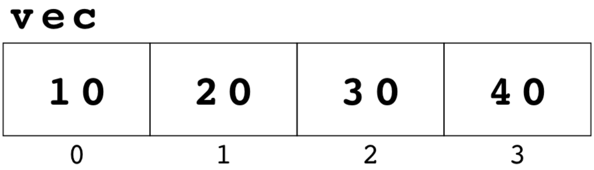
Operations on STL `vectors`

```cpp
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
```



Operations on STL `vectors`

```
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
```



Operations on STL `vectors`

```
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
```

vec

10	20	25	30	40
0	1	2	3	4

Operations on STL `vectors`

```cpp
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
vec.erase(vec.begin());

```

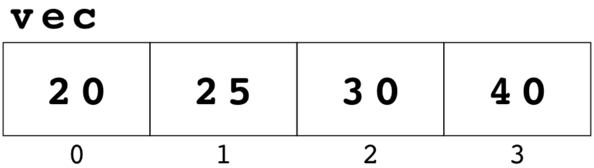
vec

10	20	25	30	40
0	1	2	3	4

Operations on STL `vectors`

```cpp
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
vec.erase(vec.begin());

```



Operations on STL `vectors`

```cpp
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
vec.erase(vec.begin());
vec[3] = 35;
```

vec

20	25	30	40
0	1	2	3

Operations on STL `vectors`

```
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
vec.erase(vec.begin());
vec[3] = 35;
```

vec

20	25	30	35
0	1	2	3

Iterating through `vector` elements

Array-style:

```
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}
```

Iterating through `vector` elements

Array-style:

```
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}
```

Using **iterators**:

```
vector<int>::iterator it;
for (it = vec.begin(); it != vec.end(); it++) {
    cout << *it << " ";
}
```

Iterating through `vector` elements

Array-style:

```
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}
```

Using **iterators**:

```
vector<int>::iterator it;
for (it = vec.begin(); it != vec.end(); it++) {
    cout << *it << " ";
}
```

Modern style ("range-based loop"):

```
for (auto e : vec) {
    cout << e << " ";
}
```

STL vectors and memory allocation

```
vector<int> makeVector() {
    vector<int> result;
    // ...
    return result;
}

int main() {
    vector<int> vec = makeVector();
    // 'vec' is used here
    return 0;
}
```

Does it work? How is memory allocated here?

STL `vectors` and memory allocation

```
vector<int> makeVector() {
    vector<int> result;
    // ...
    return result;
}

int main() {
    vector<int> vec = makeVector();
    // 'vec' is used here
    return 0;
}
```

Does it work? How is memory allocated here?

- The vector internally stores its data in a **dynamically allocated array** (using `new`)
  - Hence, **this internal array resides on the heap** (not on the stack!)

STL `vectors` and memory allocation

```
vector<int> makeVector() {
    vector<int> result;
    // ...
    return result;
}

int main() {
    vector<int> vec = makeVector();
    // 'vec' is used here
    return 0;
}
```

Does it work? How is memory allocated here?

- The vector internally stores its data in a **dynamically allocated array** (using `new`)
  - Hence, **this internal array resides on the heap** (not on the stack!)
- Some internal information of the vector (pointer to array, size) is **kept on the stack**
  - Such pointer and size are **copied to the caller**, inside `vec`, when `makeVector()` returns
  - Since the internal array survives on the heap, **the code above works**!

STL vectors and memory allocation

```
vector<int> makeVector() {
    vector<int> result;
    // ...
    return result;
}

int main() {
    vector<int> vec = makeVector();
    // 'vec' is used here
    return 0;
}
```

Does it work? How is memory allocated here?

- The vector internally stores its data in a **dynamically allocated array** (using `new`)
  - Hence, **this internal array resides on the heap** (not on the stack!)
- Some internal information of the vector (pointer to array, size) is **kept on the stack**
  - Such pointer and size are **copied to the caller**, inside `vec`, when `makeVector()` returns
  - Since the internal array survives on the heap, **the code above works**!
- When `main()` ends, `vec` is destroyed and it **automatically `delete`s its internal array**

STL `vectors` and memory allocation (cont'd)

```
void extendVector(vector<int> v) {
   v.push_back(42);
}

int main() {
    vector<int> vec;
    extendVector(vec);
}
```

Since `vec`'s internal array is on the heap, **does this code change `vec` itself**?

STL `vectors` and memory allocation (cont'd)

```
void extendVector(vector<int> v) {
   v.push_back(42);
}

int main() {
    vector<int> vec;
    extendVector(vec);
}
```

Since `vec`'s internal array is on the heap, **does this code change `vec` itself**?

- **No**: `vec` is **copied** when passed to `extendVector()` (call-by-value, as usual)
- **You need to think if copying is really what you want**
  - Do you want `extendVector(...)` to make changes to the vector that are visible in `main()`?
    If so: `void extendVector(vector<int> &v) { ... }` (call-by-reference)

STL `vector`s and memory allocation (cont'd)

Copying vectors when calling functions can be **very inefficient**

Passing references is **more efficient**, but **how do we ensure that a vector is not modified**?

STL `vectors` and memory allocation (cont'd)

Copying vectors when calling functions can be **very inefficient**

Passing references is **more efficient**, but **how do we ensure that a vector is not modified**?

We can **pass "read-only" references**

```
void printVector(const vector<int> &vec) {
    // 'vec' is "read-only" here
}
```

The keyword `const` means: the function treats its argument as a **(reference to) a constant**

- If the code of `printVector(...)` tries to change `vec`, we get a **compilation error**

STL containers and memory allocation

Memory handling in STL `vectors` makes life easier

- We can often avoid working with pointers, `new` and `delete`!

Other STL containers (`set`, `map`, `stack`, ...) have the same convenient memory handling

...but **what is going on behind the scenes** in these containers?

- We will see in the upcoming **lectures on object-oriented programming in C++**

Standard I/O and file streams

STL includes the I/O library `iostream`, based on **streams**

- the `cout` stream writes output to the console with its **insertion operator** `<<`

```
cout << "Output this string to the console" << endl;
```

- the `cin` stream takes input from console with its **extraction operator** `>>`

```
cin >> variable;
```

Standard I/O and file streams

STL includes the I/O library `iostream`, based on **streams**

- the `cout` stream writes output to the console with its **insertion operator** `<<`

```
cout << "Output this string to the console" << endl;
```

- the `cin` stream takes input from console with its **extraction operator** `>>`

```
cin >> variable;
```

STL also includes a **stream-based library for file I/O**, called `fstream`

- `ofstream` objects write output to a file with their **insertion operator** `<<`

```
file << "Output this string to a file" << endl;
```

- `ifstream` objects take input from a file with their **extraction operator** `>>`

```
file >> variable;
```

File streams: (very) basic usage

1. Declare your stream variable(s)

```
ifstream infile; // This file stream is not initialised
ofstream outfile("output.txt"); // This stream points to an open file
```

File streams: (very) basic usage

1. Declare your stream variable(s)

```
ifstream infile; // This file stream is not initialised
ofstream outfile("output.txt"); // This stream points to an open file
```

2. Open the files (if not already done)

```
infile.open("input.txt");
if (infile.fail()) {
    cout << "ERROR: cannot open the file!" << endl;
}
```

File streams: (very) basic usage

1. Declare your stream variable(s)

```
ifstream infile; // This file stream is not initialised
ofstream outfile("output.txt"); // This stream points to an open file
```

2. Open the files (if not already done)

```
infile.open("input.txt");
if (infile.fail()) {
    cout << "ERROR: cannot open the file!" << endl;
}
```

3. Read/write data from/to the open files

```
string x;
infile >> x;
outfile << "I have read the string: " << x << endl;
```

File streams: (very) basic usage

1. Declare your stream variable(s)

```
ifstream infile; // This file stream is not initialised
ofstream outfile("output.txt"); // This stream points to an open file
```

2. Open the files (if not already done)

```
infile.open("input.txt");
if (infile.fail()) {
    cout << "ERROR: cannot open the file!" << endl;
}
```

3. Read/write data from/to the open files

```
string x;
infile >> x;
outfile << "I have read the string: " << x << endl;
```

4. When done, you can close the files (but this **happens automatically** when streams go out of scope)

```
infile.close();
outfile.close();
```

STL `string`: a useful basic data type

In C++, **strings are natively represented as arrays of** `char`**s** with last element `0`

- They have **all disadvantages of arrays** (difficult to resize, risk of out-of-bound access...)

STL `string`: a useful basic data type

In C++, **strings are natively represented as arrays of `char`s** with last element `0`

- They have **all disadvantages of arrays** (difficult to resize, risk of out-of-bound access...)

The STL `<string>` header file provides the `string` type that **makes life much easier**

- We have already used it!

Operations on STL `string`s:

- assignment using `=` (makes a new copy)
- comparison using `<`, `==`, `>=`, ... (by alphabetical ordering)
- concatenation using `+`

An overview of STL `strings`

We can create objects of type `string` in several ways:

```
string str1("Hello World");
string str2 = "Hello world"; // Equivalent to the above
```

A few `string` methods:

```
str1.empty();   // Returns true if str1 is empty, false otherwise
str2.length();  // Returns the length of str2
```

An overview of STL `strings`

We can create objects of type `string` in several ways:

```
string str1("Hello World");
string str2 = "Hello world"; // Equivalent to the above
```

A few `string` methods:

```
str1.empty();   // Returns true if str1 is empty, false otherwise
str2.length();  // Returns the length of str2
```

...but **what is going on behind the scenes** in STL `string`s?

- We will see in the upcoming **lectures on object-oriented programming in C++**

Lab

Today's lab begins now. Tasks:

- make sure C++ works on your computer, request help if it doesn't
- begin working on **Assignment 5**
  - **suggestion**: have a look at the live coding files before starting. . .
- ask questions if something is unclear (including previous assignments)