02393 Programming in C++

DTU

Before we start:

If you feel ill, go home
Keep your distance to others
Wash or sanitize your hands
Disinfect table and chair
Respect guidelines and restrictions

02393 Programming in C++
Module 5: Libraries and Interfaces (continued)

Lecturer:
Alceste Scalas

(Slides based on previous versions by Andrea Vandin, Alberto Lluch Lafuente, Sebastian Mödersheim)

29 September 2020

Lecture Plan

#	Date	Topic	Book chapter *
1	01.09	Introduction	
2	08.09	Basic C++	1
3	15.09	Data Types	2
4	22.09		
		Libraries and Interfaces	3
5	29.09		
6	06.10	Classes and Objects	4.1, 4.2 and 9.1, 9.2
		Autumn break	
7	20.10	Templates	4.1, 11.1
8	27.10	LAB DAY	Old exams
9	03.11	Inheritance	14.3, 14.4, 14.5
10	10.11	Recursive Programming	5
11	17.11	Linked Lists	10.5
12	24.11	Trees	13
13	01.12	Exercises & Summary	
	07.12	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g., strings and vectors).

Outline

Outline

Static vs. dynamic memory allocation

Static Allocation

- As a local variable in a scope, or as parameter of a function
- Example `i` and `j` in: `void f(int i){ int j=0; ... }`
- Allocated on the stack. To note:
  - ★ life time: until the scope ends (e.g. when a function returns)
  - ★ stack size: not much, so not suitable for huge data structures.

Dynamic Allocation

- Using the `new` operator
- Example: `int *p = new int[n];`
- Allocated in the heap (lots of memory available).
- life time: as you wish — until you say `delete[] p;`
- Rule of thumb: for every `new` there should be a corresponding `delete`. Otherwise you may get memory leaks!

Dynamic Allocation of Structures

```
struct point {
  int x;
  int y;
};

int main() {
   ...
   point *p = new point;
   ...
   // These two lines do the same
   (*p).x=7;
   p->x=7;
   ...
   delete p;
}
```

Declared arrays & dynamic arrays

- **Declared array:**
  - Example: `bool isPrime[n];`
  - On Microsoft C++ it only works if `n` is known at compile time
  - Memory is allocated automatically, all the elements are allocated on the stack: "local variable" of the present function
  - The stack has limited capacity
  - Life time: until the scope of the array variable ends

- **Dynamic array:**
  - Example: `bool *isPrime = new bool[n];`
  - Always works on Microsoft C++ (and any other compiler)
  - Memory allocated on the **heap** with the `new[]` operator
  - The heap has very large capacity (depends on system memory)
  - Life time: until you invoke `delete[]`

Outline

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 you know!
  - ★ other ordered collections: `queue`, `stack`
  - ★ dictionaries: `map`, `multimap`
- It is important to know how to deal with them
- It is important to choose the right one:
  - ★ more than one class of containers may do the job
  - ★ ...but some may do the job better (e.g., faster)

vector: motivations

Array: fundamental type in many programming languages

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

`vector`: motivations

Array: fundamental type in many programming languages

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

The `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 the interface you should:

```
#include <vector>
```

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

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

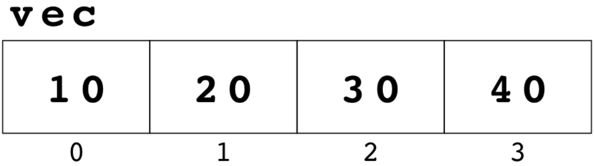
Declaring a new empty `vector` object:

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

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

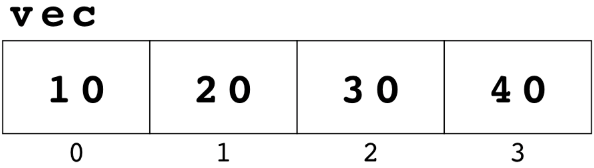
Operations on the `vector` class

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



Operations on the `vector` class

```
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 the `vector` class

```
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 the `vector` class

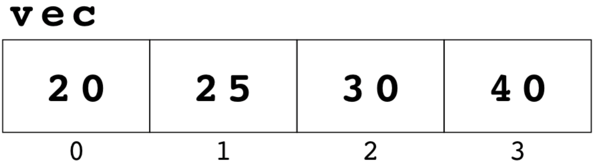
```
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2, 25);
vec.erase(vec.begin());
```

vec

10	20	25	30	40
0	1	2	3	4

Operations on the `vector` class

```
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 the `vector` class

```
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 the `vector` class

```
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-like style:

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

Iterating through `vector` elements

Array-like 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-like 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 << " ";
}
```

Vectors and Memory Allocation (1/3)

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

Does it work? How is memory allocated here?

Vectors and Memory Allocation (1/3)

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

Does it work? How is memory allocated here?

- The vector internally uses an array. This array is dynamically allocated and thus resides on the heap not on the stack
- So no problem with lifetime

Vectors and Memory Allocation (1/3)

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

Does it work? How is memory allocated here?

- The vector internally uses an array. This array is dynamically allocated and thus resides on the heap not on the stack
- So no problem with lifetime
- Some internal information of the vector (the pointer to the array, the size variable) are on the stack though
- They are copied when returning to the caller of `f()`

Vectors and Memory Allocation (2/3)

```
void f(vector<int> v) {
  v.push_back(17);
}

int main() {
  vector<int> w;
  f(w);
}
```

If the actual array is on the heap, does this change `w`, i.e., is this like call by-reference?

Vectors and Memory Allocation (2/3)

```
void f(vector<int> v) {
  v.push_back(17);
}

int main() {
  vector<int> w;
  f(w);
}
```

If the actual array is on the heap, does this change `w`, i.e., is this like call by-reference?

- No, it is being copied! This works like call-by-value
- You need to think if copying is really what you want

Vectors and Memory Allocation (2/3)

```
void f(vector<int> v) {
  v.push_back(17);
}

int main() {
  vector<int> w;
  f(w);
}
```

If the actual array is on the heap, does this change `w`, i.e., is this like call by-reference?

- No, it is being copied! This works like call-by-value
- You need to think if copying is really what you want
  - ★ Do you want the procedure to make changes to the vector that are visible outside? If so: `void f(vector<int> &v)`

Vectors and Memory Allocation (3/3)

How to void copying the vector if it is not modified:

```
void printVector(const vector<int> &vec) {
    ...
}
```

Vectors and Memory Allocation (3/3)

How to void copying the vector if it is not modified:

```
void printVector(const vector<int> &vec) {
    ...
}
```

The keyword **const** means that the function promises not to change the vector

Vectors and Memory Allocation (3/3)

How to void copying the vector if it is not modified:

```
void printVector(const vector<int> &vec) {
  ...
}
```

The keyword **const** means that the function promises not to change the vector

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

Containers and Memory Allocation

- Memory handling in vectors makes life easier
  - ⋆ We can often avoid working with pointers, new and delete!
- Other STL containers like `set`, `map`, `stack`, etc. have the same convenient memory handling
- ...but what is going on behind the scenes in these containers? We will see in the lectures on OOP (later in the course)

Outline

Standard I/O and file streams

Standard I/O (library `iostream`)

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

```
cout << "output this string to console" << endl;
```

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

```
cin >> buffer;
```

Standard I/O and file streams

Standard I/O (library `iostream`)

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

```
cout << "output this string to console" << endl;
```

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

```
cin >> buffer;
```

File streams (library `fstream`)

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

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

- `ifstream` objects input from a file with extraction operator

```
file >> buffer;
```

Using file streams

1. Declare a stream variable

```
ifstream infile;
ofstream outfile;
```

Using file streams

1. Declare a stream variable

```
ifstream infile;
ofstream outfile;
```

2. Open the file

```
infile.open("input.txt");
outfile.open("output.txt");
if (infile.fail())
   cout << "Cannot open file!" << endl;
```

Using file streams

1. Declare a stream variable

```
ifstream infile;
ofstream outfile;
```

2. Open the file

```
infile.open("input.txt");
outfile.open("output.txt");
if (infile.fail())
    cout << "Cannot open file!" << endl;
```

3. Read/write data

```
infile >> x;
outfile << "Hello world!" << endl;
```

Using file streams

1. Declare a stream variable

```
ifstream infile;
ofstream outfile;
```

2. Open the file

```
infile.open("input.txt");
outfile.open("output.txt");
if (infile.fail())
    cout << "Cannot open file!" << endl;
```

3. Read/write data

```
infile >> x;
outfile << "Hello world!" << endl;
```

4. Close the file

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

Outline

`string`: a useful basic data type

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

`string`: a useful basic data type

- In C++, strings are natively represented as arrays of `char`s with last element 0
- The `<string>` header file provides a `string` type that makes life much easier (we have already used it!)

Operations on `strings`

- assign using $=$, makes new copy
- comparison $(<, ==, >=, \ldots)$ using alphabetical ordering
- concatenation using $+$

An overview of `string`

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

- `string str("Hello World");`
- `string str = "Hello world";`

An overview of `string`

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

- `string str("Hello World");`
- `string str = "Hello world";`

A few `string` methods:

- `str.empty();`
- `str.length();`

An overview of `string`

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

- `string str("Hello World");`
- `string str = "Hello world";`

A few `string` methods:

- `str.empty();`
- `str.length();`

...but what is going on behind the scenes? We will see in the lectures on OOP (later in the course)

02393 Programming in C++

Before we leave:

Disinfect table and chair
Maintain your distance to others
Wash or sanitize your hands
Respect guidelines and restrictions outside