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

Sebastian Mödersheim

September 26, 2016

Lecture Plan

#	Date	Topic
1	29.8.	Introduction
2	5.9.	Basic C++
3	12.9.	Data Types, Pointers
4	19.9.	
5	26.9.	Libraries and Interfaces; Containers
6	3.10.	Classes and Objects I
7	10.10.	Classes and Objects II
		Efterårsferie
8	24.10.	Classes and Objects III
9	31.10.	Recursive Programming
10	7.11.	Lists
11	14.11.	Trees
12	21.11.	Novel C++ features
13	28.11.	Summary
	5.12.	Exam

Outline

1. **Dynamic Memory Allocation**
2. **Vectors and other Containers**
3. **File I/O**
4. **Strings**

Outline

1. Dynamic Memory Allocation
2. Vectors and other Containers
3. File I/O
4. Strings

Live Programming

Course repository for live programming

svn checkout svn://repos.gbar.dtu.dk/samo/cpp2016/
with username `student` and password `yvyebbnnq532ej3b`

Static vs. dynamic memory allocation

Static Allocation

- As a local variable in a new scope or 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 somewhere in your program 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 s;
}
```

Dynamic arrays & declared arrays

- Declared array:
  - ★ Example: `bool isPrime[n];`
  - ★ Memory is allocated automatically, all the elements are allocated on the stack: "local variable" of the present function.
  - ★ The stack has very limited capacity and the life time of the variable is until the scope of the variable ends.
- Dynamic array:
  - ★ Example: `bool * isPrime = new bool[n];`
  - ★ memory allocated on the heap with the `new[]` operator
  - ★ Items on the heap live until you say `delete[]`.
  - ★ the actual memory is not allocated until you invoke the `new[]` operator

Outline

1. Dynamic Memory Allocation
2. **Vectors and other Containers**
3. File I/O
4. Strings

STL (standard template library)

STL (standard template library): is a C++ library of container classes and algorithms

- Containers are collections 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 almost all languages

- not easy to resize :(
- you have to keep track of the actual size :(
- insertion and deletion not easy/performant in general :(
- you have to be careful to index within the array bounds :(

the `vector` class solves all of these problems!

see:

http://www.cplusplus.com/reference/stl/vector/

http://en.cppreference.com/w/cpp/container/vector

for examples and documentation of the member functions.

`vector`: declaration

to use the interface include:

```
#include <vector>
```

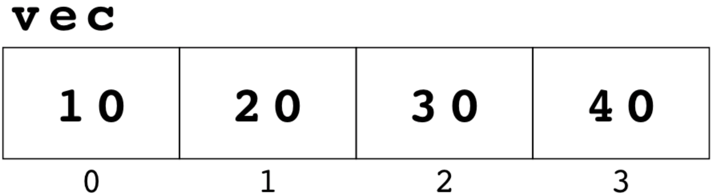
- `vector` is a container class: it contains other objects
- `vector<int>` specifies a vector whose elements are `int`s, `vector<double>` specifies a vector whose elements are `double`s, `vector<vector <int>>` specifies a vector whose elements are vectors of `int`s, ...
- The enclosed type is called the base type
- Declaring a new empty `vector` object

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

- Automatic initialization while declaration of class objects: in the above case we have a empty vector, but there are many other options (called constructors).

Operations on the `vector` class

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



The Standford Reader uses it's own closed-source library which has some functions (like `add` for vectors) that do not exist in the STL.

Operations on the `vector` class

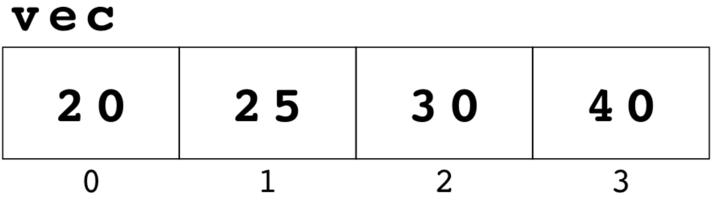
```
vector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
vec.push_back(40);
vec.insert(vec.begin()+2,25);
```

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());
```



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 the elements

Modern style

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

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 << " ";
 }
```

Vectors and Memory Allocation (1/3)

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

Does that 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 life-time.
- Some "administrative information" of the vector (the pointer to the array, the size variable) are on the stack though.
- Thus not much is to be "copied" on return.

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 that 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: call-by-reference!)
  - If your procedure does not make any changes to the vector at all, you should avoid the copying of the entire vector.

Vectors and Memory Allocation (3/3)

Avoiding the copying of the entire vector if it is not modified anyway:
Call-by-reference.
Example:

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

With the keyword **const** the function promises not to change the vector.
Adding a change to the vector then leads to a compiler error.

Containers and Memory Allocation

- This handling of memory allocation in vectors gets rid of many problems.
  - ★ We can often avoid working with pointers and new and delete entirely!
- The other containers like set, map, stack etc. have the same convenient memory management.
- We will in the lectures on OOP in part 2 of course see in more detail what is going on behind the scenes in these containers.

Outline

1. Dynamic Memory Allocation
2. Vectors and other Containers
3. **File I/O**
4. Strings

Standard I/O and file streams

Standard I/O

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

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

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

```
// read a line from the console
cin >> buffer;
```

file streams

- `ofstream` objects are used to write output to the file with insertion operator

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

- `ifstream` objects are used to input from the file with extraction operator

```
// read a line from the file
infile >> buffer;
```

Using file streams

- Declare a stream variable

```
ifstream infile;
ofstream outfile;
ifstream infile("hello.cpp");
```

- open the file

```
infile.open("hello.cpp");
outfile.open("out.cpp");
if (infile.fail())
cout<< "could not open the file!" << endl;
```

- transfer data
- close the file

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

Outline

1. Dynamic Memory Allocation
2. Vectors and other Containers
3. File I/O
4. **Strings**

String: a "special" basic data type

- a sequence of characters
- defined in the `<string>` interface
- is an abstract data type – more will be introduced later

Operation on strings

- assign using $=$, makes new copy
- compare with relational ops $(<, ==, >=, \ldots)$ using lexicographic ordering
- concatenation using overloaded $+$

string member functions

Declare a variable of type string:

```
string str("Hello World");
```

Constructor: Constructs a standard string object and initializes its content.

Invoke member functions using dot notation

```
str.function(arguments);
```

str is called the receiver string

sample member functions:

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