DIGITAL ARTS & ENTERTAINMENT

Overview course Programming 2

1. Classes 3 – Operator Overloading
2. std::vector and Transformations
3. Templates and Sequence containers
4. Classes 4 – Inheritance
5. Classes 5 – Polymorphism
6. Classes 6 – Copy semantics - Rule of 3
7. Classes 7 – Move semantics - Rule of 5
8. Input / Output – iostream library
9. Iterators – Algorithms
10. Associative Containers

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

DIGITAL ARTS & ENTERTAINMENT

Sequence container
std::vector

References: http://en.cppreference.com/w/cpp/concept/SequenceContainer

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

What is a container?

- The container:
  - Is a holder object.
  - Stores a collection of other objects (its elements).
  - Are implemented as class templates (see later), which allows a great flexibility in the types supported as elements.
  - Are part of the STL (Standard Template Library)
    - std namespace:
      - example: `std::vector<int> numbers`

What is a container?

- The container:
  - Manages the storage space for its elements.
  - Provides member functions to access them
    - directly or through iterators (later)

Container class templates

- Three categories:
  - Sequence containers
    - array, vector, deque, list
  - Associative containers (see later)
    - map, multimap, set , multiset
  - Container adapters
    - stack, queue
    - Example: The stack adapts the deque container to provide strict last-in, first-out (LIFO) behavior

More about the vector

- Internals: (resembles the container class from the lab)
  - Uses dynamic memory allocation
    - Dynamic array -> heap memory.
  - Elements are stored in contiguous locations.
    - They are next to each other in memory
    - Pointer offset to access elements is possible

More about the vector

- Internals: (resembles the container class from the lab)
  - Growing pains: each time an element is added to the vector, it needs more storage space:
    - Create new larger dynamic array
    - Copy the elements from the old to this larger array
    - Destroy the old array
  - That takes a lot of cpu cycles and is a called an “expensive” operation.
  - Avoid if possible: next slide

More about the vector

- Internals: (resembles the container class from the lab)
  - To avoid having to grow one element each time an element is added, we grow by more than one element, introducing "capacity": to have more storage space than elements.
    - Capacity: how much storage space is there.
    - Size: how much of the storage space is actually used.
    - the capacity is larger than the number of elements for efficiency reasons.
  - A trade of between memory and speed

Implementation

- Needs:

```
#include <vector>
```

- Usage:

```
vector<type> nameOfTheVector;
```

- Example:

```
vector<int> myNumbers;
vector<Texture*> myTexturePointers;
```

Methods or member functions

- Commonly used methods: see cplusplus.com
  - size(): returns the number of actual elements.
  - resize(): inserts/erases element(s)
  - erase(): erases element(s) (!)
  - capacity(): returns the size of the storage capacity
  - reserve(): request change in capacity
  - operator []: access element without boundary checking
  - at(): access element
  - push_back(): add element at the end
  - pop_back(): remove last element
  - clear(): Removes all elements from the vector (!)
  - data(): direct pointer to the memory array used internally

DIGITAL ARTS & ENTERTAINMENT

Example: vector of integers

```
#include <iostream>
#include <vector>

int main(int argc, char * argv[])
{
  std::vector<int> myVector;
  for (int i = 0; i < 5; i++) myVector.push_back(i * 10);

  for (size_t i = 0; i < myVector.size(); ++i)
  {
    std::cout << "Index: " << i << "    value: " << myVector[i] << '\n';
  }
}
```

Example: vector of Time pointers

```
#include <iostream>
#include <vector>
#include "Time.h"
int main(int argc, char * argv[])
{
  std::vector<Time*> times;
  for (int i = 0; i < 5; i++) times.push_back(new Time{i * 10});

  for (size_t i = 0; i < times.size(); ++i)
  {
    std::cout << "Index: " << i << "    value: " << times[i]->Print() << '\n';
  }
  for (int i = 0; i < times.size(); i++) delete times[i];
}
```

Range based for loop

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> numbers;
  for (int i = 0; i < 5; i++) numbers.push_back(i * 10);

  for (const int& element : numbers)
  {
    std::cout << ' ' << element;
  }
  return 0;
}
```

Range based for loop

```
for (int x : numbers)
{
  std::cout << ' ' << x;
}

for (int& x : numbers)
{
  ++x;
}

for (const int& x : numbers)
{
  std::cout << ' ' << x;
}
```

iterates over the elements of the container, x is a copy of each element

iterates over the elements of the container, x is a reference to each element, modifying x, modifies the element in the container (!)

iterates over the elements of the container, x is a const reference to each element

Range based for loop

```
for (int x : numbers)
{
  std::cout << ' ' << x;
}

for (int& x : numbers)
{
  ++x;
}

for (const int& x : numbers)
{
  std::cout << ' ' << x;
}
```

- Positive: Easy way to iterate over elements.
- Negative: lost index count, making it not always practically
- Available for all stl container types