C++

Container classes

Introduction

- A container is an object that holds other objects
- There are three important concepts:
  - The container type (sequence, associative container, container adapters)
  - The type of objects contained (int, double, user defined class)
  - The operations on the containers and their complexity
    - ex: stack operations and their complexity

Example

- We will illustrate how containers works (and can be implemented) through a simplified example.
- We will implement a Vector.
- Underlying representation will be an array: i.e a sequence of homogeneous elements.
- Operations:
  - Assignment.
  - Assignment and access to a random element.

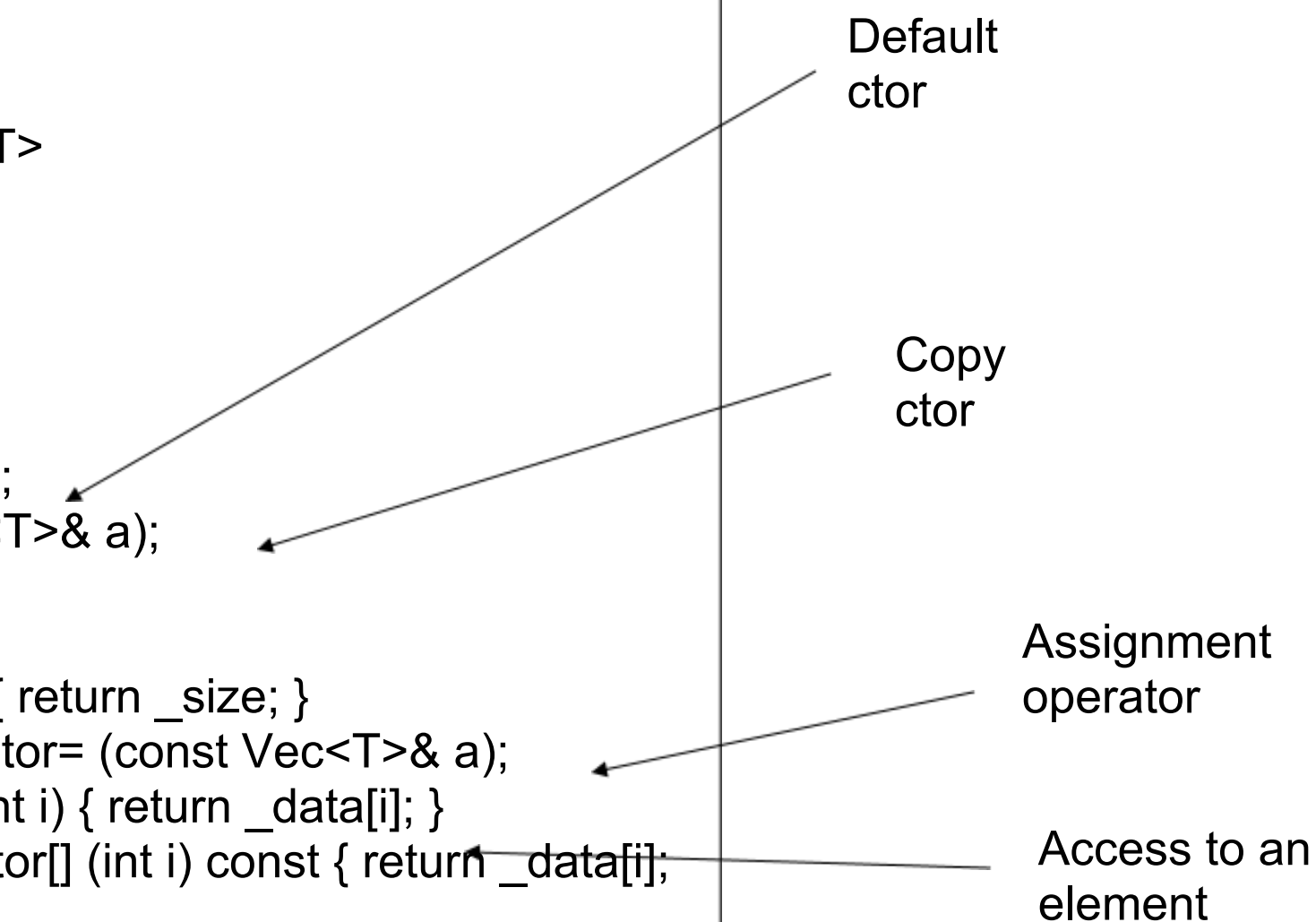
Example: Array

```
#ifndef VEC_H
#define VEC_H

template <class T>
Vec {
private:
  int _size;
  T* _data;
public:
  Vec(int sz=100);
  Vec(const Vec<T>& a);
  ~Vec();

  int size() const { return _size; }
  Vec<T>& operator= (const Vec<T>& a);
  T& operator[] (int i) { return _data[i]; }
  const T& operator[] (int i) const { return _data[i];
}
};

#endif
```



Ctor and Dtor

```
template<class T>
Vec<T>::Vec(int sz) : _size(sz), _data(new T[sz]){}

template<class T>
Vec<T>::Vec(const Vec<T>& a) : _size(a._size), _data(new T[a._size]) {
  for (int i=0; i < _size; i++) {
   _data[i] = a._data[i];
  }
}

template<class T>
Vec<T>::~Vec() {
  delete[] _data;
}
```

Copy Ctor: reminder

- Remember: if a class owns data (has member data which is a pointer), then you should provide a copy constructor (as well as a dtor and the assignment operator)
- The default copy constructor does only bit-wise copy
- Syntax:
  Vec<int> A;
  Vec<int> B = A; // or B(A)
- With the default copy ctor, _data in B and A will point to the same location
- Deleting _data, for example in B dtor, will also delete the memory allocated for A._data and potentially create memory corruption

Assignment operator

- Let us look first at a wrong implementation of the assignment operator, we will then comment and fix it

```
template<class T>
Vec<T>&
Vec<T>::operator= (const Vec<T>& a) {
  delete[] _data;
  _size = a._size;
  _data = new T[_size];
  for (int i =0; i < _size; i++){
   _data[i] = a._data[i];
  }
  return *this;
}
```

The problem is here

Assignment operator problem

- Suppose that we write:
  Vec<int> a;
  a = a; // we start by deleting a._data
- Remember the lesson on assignment operator: we need to be careful about self-assignment

Assignment operator fixed

```
template<class T>
Vec<T>&
Vec<T>::operator= (const Vec<T>& a) {
  if (this != &a) {
   delete[] _data;
   _size = a._size;
   _data = new T[_size];
   for (int i =0; i < _size; i++){
     _data[i] = a._data[i];
   }
  }
  return *this;
}
```

We can now write:

```
Vec<int> a;
a = a;
```

Printing facilities

- We can add printing facilities to Array by overloading <<
- We want to do:
  Vec<int> a;
  cout << a << endl;

```
template <class T>
ostream&
operator<< (ostream& os, const Vec<T>& a){
   for (int i = 0; i < a.size(); i++) {
     os << a[i] << " ";
   }
   return os;
}
```

Note: we already saw how to overload "<<" in the prev. lesson on op. overloading

Usage

```
#include <iostream>
#include <Vec.h>

using namespace std;

int main() {
  Vec<int> a(20);
  for(int i = 0; i < a.size(); i++) {
   a[i] = i; // call operator[]
  }
  cout << a << endl;

  Vec<int> b = a; // copy ctor
  cout << b.size() << endl;

  Vec<int> c(50);
  c = a; // assignment op
}
```

Remarks on operator[]

- There are two implementations for operator[]() in the class Vec<T>:
  - template<class T> T& Vec<T>::operator[] (int i);
  - template<class T> const T& Vec<T>::operator[] (int i) const;
- The second version is needed when e. g. passing a *const Vec* to a function that will need to read some of the elements

Remarks on operator[]

```
template <class T>
T f (const Vec<T>& a)
{
  T sum = 0;
  for (int i = 0; i < a.size(); i++)
    sum += a[i];
  return sum;
}
```

a is
const

Elements of a are read
only
but not modified

Without the second form of operator[], this
code
would not compile.

Additional operations

- We could add operations to manipulate our sequence as a stack:

```
template < class T > class Vec {
public:
  void push_back(const T& x); // add to end
  void pop_back(); // remove last element
};
```

Additional operations

- ... or as a list:

```
template < class T > class Vec {
public:
  iterator insert (iterator pos, const T& x); // add
x before pos
  iterator erase(iterator pos); // remove element
at pos
  // ...
};
```

Standard containers

- Standard containers definition usually specify:
  - member types
  - iterators
  - constructors and destructors
  - element access (e.g. operator[]() or at())
  - operations (e.g. stack operations)
  - size and capacity
  - helper functions

Standard containers

- What we implemented in our example is:
  - member types
  - iterators
  - constructors and destructors
  - element access (e.g. operator[]() or at())
  - operations (e.g. stack operations)
  - size and capacity
  - helper functions