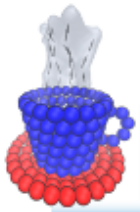



# Templates

**Axel Arnold** **Olaf Lenz**

Institut für Computerphysik
Universität Stuttgart

February 18-22, 2013

# Template functions

```
template<typename STLVectorClass>
void print(const STLVectorClass &vec) {
  for (auto val: vec) {
    cout << val << endl;
  }
}
```

- templates are regular functions or classes pretented by the **`template`** keyword
- implement compile time polymorphism
- template parameters can be types or constants
- for each data type used, a seperate function is compiled
- definitions need to be in the header files

# Template classes

```
template<typename T> class Vector {
  T *data;
  int size;
public:
  Vector(int len);
  int getSize() const { return size; }

  T       &operator[](int i)       { return data[i]; }
  const T &operator[](int i) const { return data[i]; }
};
template<typename T> Vector<T>::Vector(int len): data(0)
{ allocate(len); }
```

- template classes the same
- compare STL vector<type> (but *way* more complex)
- using templates as type is possible, e.g. `Vector<vector<int> >`
- `Vector<vector<int>>` does *not* work (missing space)

# Multiple parameters, default values

```
template<typename T, int size = 3>
class ValArray {
  T data[size];
public:
  int getSize() const { return size; }

  T       &operator[](int i)       { return data[i]; }
  const T &operator[](int i) const { return data[i]; }
};
```

- templates can have multiple parameters, here second is of type **int**
- second parameter has default value 3
- default value can be derived from first parameter `size = DefaultArraySize<T>`, for example
- commonly used in the STL, `allocator<T>`

# Class and type members

```
template<typename T, int size = 3>
class ValArray {
  T data[size];
public:
  typedef T *iterator;
  typedef const T*const_iterator;
  iterator begin() { return data; }
  iterator end()   { return data + size; }
  const_iterator begin() const { return data; }
  const_iterator end()   const { return data + size; }
};
```

- classes can contain type definitions or classes
- useful for template classes that derive types
- here: iterator types
- use for example as `ValArray<int>::iterator it`
- `for(auto v : valArrayInstance)` works with this interface!