UCLA Samueli Computer Science

CS32: Introduction to Computer Science

Discussion Week 6

Junheng Hao, Ramya Satish

Feb 15, 2019

Announcement

UCLA Samueli Computer Science

- Project 3 Part 1 is due on next Thursday, February 21.
- Midterm Part 2 is on Tuesday, Feb 26.
- **For all students in Lecture 2 and Lecture 3, Prof. Smallberg will have a make-up lecture in Moore 100 at 1:00-1:50pm today.**

Outline

UCLA Samueli Computer Science

- Template
- Standard Template Library (STL)

Template

Motivation: More generic class

UCLA Samueli Computer Science

- Think about the `Pair` class. The class should not work only with integers. That is we want a “generic” `Pair` class. (Well, I know you were thinking `typedef` just now~)
- Here we go: `Pair<int> p1; Pair<char> p2;`

```
class Pair {
    public:
       Pair();
       Pair(int firstValue,
            int secondValue);
       void setFirst(int newValue);
       void setSecond(int newValue);
       int getFirst() const;
       int getSecond() const;
    private:
       int m_first;
       int m_second;
};
```

```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

Template

Multi-type template

UCLA Samueli Computer Science

- What if we need pair with different types? (One with `int` value while the other with `string` value)
- Just slightly change your template class and: `Pair<int, string> p1;`

```
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```

```
template<typename T, U>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             U secondValue);
        void setFirst(T newValue);
        void setSecond(U newValue);
        T getFirst() const;
        U getSecond() const;
    private:
        T m_first;
        U m_second;
};
```

Template

Change member functions in template classes

UCLA Samueli Computer Science

- Member function should also be edited in template class as well.

```
void Pair::setFirst(int newValue)
{
  M_first = newValue;
}
```

```
template<typename T>
void Pair<T>::setFirst(T newValue)
{
  M_first = newValue;
}
```

Template

Template Specialization

UCLA Samueli Computer Science

- What if we want a template class with certain data type to have its own exclusive behaviors? For example, in `Pair` class we only allow `Pair<char>` has `uppercase()` and `lowercase()` function but not for `Pair<int>`.

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

```
Pair<int> p1;
Pair<char> p2;

p1.uppercase(); //error
p2.uppercase(); //correct
```

Template

Const references as parameters

UCLA Samueli Computer Science

- When you are not changing the values of the parameters, make them `const` references to avoid potential computational cost. (Pass by value for ADTs are slow.)

```
template<typename T>
T minimum(const T& a, const T& b)
{
  if (a < b)
    return a;
  else
    return b;
}
```

Template

Some notes

UCLA Samueli Computer Science

- Generic comparisons:
  - `bool operator>=(const ItemType& a, const ItemType& b)`
- Use the template data type (e.g. `T`) to define the type of at least one formal parameter.
- Add the prefix **`template <typename T>`** before the class definition itself and before each function definition outside the class. Also place the postfix **`<T>`** Between the class name and the :: in all function definition.

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void);
  private:
    T m_a;
};
```

```
template <typename T>
void Foo<T>::setVal(T a)
{
  m_a = a;
}
template <typename T>
void Foo<T>::printVal(void)
{
  cout << m_a << "\n";
}
```

STL: Standard Template Library

Easy and efficient implementation

UCLA Samueli Computer Science

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues…

STL: Standard Template Library

Notes on `vector` and `list`

UCLA Samueli Computer Science

- You may only use brackets to access existing items in `vector`. Keep the current size vector in mind especially after `push_back()` and `pop_back()`.
- You cannot access list element by brackets.
- Choose between vector and list:
  - `vectors` are based on dynamic arrays placed in contiguous storage. Fast on access but slow on insertion/deletion.
  - `lists` are the opposite. It offers fast insertion/deletion, but slow access to middle elements.

STL: Standard Template Library

Notes on `size` and `capacity`

UCLA Samueli Computer Science

- Bonus question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

```
size: ?
capacity: ?

size: ?
capacity: ?

max size: ?
```

STL: Standard Template Library

Notes on `size` and `capacity`

UCLA Samueli Computer Science

- Bonus question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

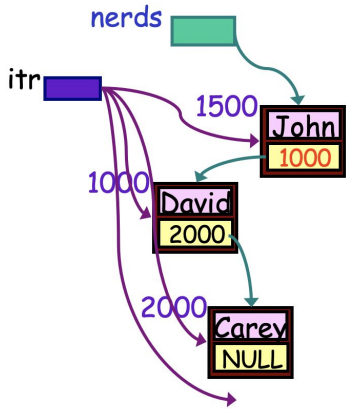
```
→ On my computer:
size:1
capacity:1
size:101
capacity:128
max size:4611686018427387903
```

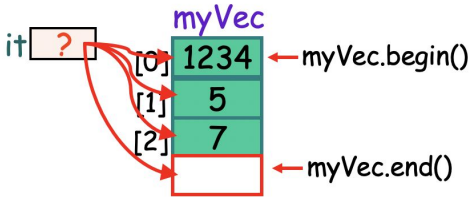
STL: Standard Template Library

Implementation example: Iterators

UCLA Samueli Computer Science

- STL Iterators: Use `.begin()` and `.end()`
  - `.begin()` : return an iterator that points to the first element.
  - `.end()`: return an iterator that points to the ***past-the- last*** element.
- A container as a `const` reference cannot use regular iterator but need to use `const` iterator. Example: `list<string>::const_iterator it;`
- Examples






```
void main()
{
  vector<int>   myVec;
  myVec.push_back(1234);
  myVec.push_back(5);
  myVec.push_back(7);
  vector<int>::iterator it;
  it = myVec.begin();
  while (  it != myVec.end() ){
    cout << (*it);
    it++;
  }
}
```

STL: Standard Template Library

Warning: using iterators for changing vector

UCLA Samueli Computer Science

- It could be dangerous to use iterator to traverse a vector when we have performed insertion/deletion.
- Safe solution: Reinitialize iterators of a vector whenever its size has been changed.

```
// Guess what is the output?
int main ()
{
  vector<int> v;
  v.push_back(50);
  v.push_back(22);
  v.push_back(10);
  vector<int>::iterator b = v.begin();
  vector<int>::iterator e = v.end();
  for (int i = 0; i < 100; i++) { v.push_back(i); }
  while (b != e) {
    cout << *b++ << endl;
  }
}
```

Standard Template Library

How to use

UCLA Samueli Computer Science

- Remember the basic provided libraries
- Check http://www.cplusplus.com/reference/stl/ for more details if needed.

STL: Standard Template Library

Some more topics

UCLA Samueli Computer Science

- More STL examples, such as `map, set, etc.`
- More STL algorithms, such as `find(), sort(), etc.`

*Inline Functions

Motivation & Examples

UCLA Samueli Computer Science

- When you define a function as being inline, you ask the compiler to directly embed the function's logic into the calling function (for speed).
- All methods with their body defined directly in the class are inline. Simply add the word inline before the function return type to make an externally defined method inline.
- Inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
  - Loops, recursion, static variables, etc
- Save time for function call vs large binary executable file?

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void){cout<<m_a<<endl;}
  private:
    T m_a;
};
```

```
inline template <typename T>
void Foo<Item>::setVal(T a)
{
  m_a = a;
}
```

UCLA Samueli Computer Science

Thank you!

Q & A