C++ - Templates and Abstract Data Types

December 15, 2017

Abstract Data Types

- A data type represents a set of values (think of int, char, etc) and defines a collection of operations (e.g + - *).
- An abstract data type (ADT) represents a set of values and operations but does not specify any implementation for those operations.
- When people talk about ADTs, they mostly mean containers.
- A container, or collection, stores a number of data of the same type. Containers are inevitable in almost all semi-serious programming cases (one-to-many associations).
- We've seen some already: arrays, linked lists, maps, etc.
- We'll study some other common instances of them, and introduce C++ Templates along the way.

A Simple Container

Arrays vs. Linked Lists:

- The nth element in an array can be accessed by `a[n]`, while for a linked list one must count from the head or tail (doubly-linked)
- Linked lists are flexible in size; arrays are fixed-size.

Lets make a best-of-both:

ExtendableIntArray
count : int capacity: int store : int[]
operator[](int index) : int& append(int value) : void ensureCapacity(int cap) : void compact() : void

```
class ExtendableIntArray {
  int _count, _capacity;
  int* store;

  void resizeStore() {
    int* newStore = new int[_capacity];
    for (int i = 0; i < _count; i++)
      newStore[i] = store[i];
    delete[] store;
    store = newStore;
  }

public:
  ExtendableIntArray(int cap = 10) {
    if (cap < 0) cap = 0;
    _count = 0;
    _capacity = cap;
    store = new int[cap];
  }
```

```
virtual ~ExtendableIntArray() {
  delete[] store;
}

int count() const {
  return _count;
}

int capacity() const {
  return _capacity;
}

int& operator[](int index) {
  return store[index];
}
```

```
  const int& operator[](int index) const {
    return store[index];
  }
  void ensureCapacity(int cap) {
    if (_capacity < cap) {
      while (_capacity < cap) (_capacity *= 2)++;
      resizeStore();
    }
  }

  void append(int value) {
    ensureCapacity(_count + 1);
    store[_count++] = value;
  }

  void compact() {
    if (_capacity > _count) {
      _capacity = _count; resizeStore();
    }
  }
};
```

Action	Count	Cap.	Contents
ExtendableIntArray arr(5)	0	5	
arr.append(1)..arr.append(5)	5	5	1, 2, 3, 4, 5
arr.append(6)	6	11	1, 2, 3, 4, 5, 6
arr[2] = 30	6	11	1, 2, 30, 4, 5, 6
arr.ensureCapacity(50)	6	95	1, 2, 30, 4, 5, 6
arr.compact()	6	6	1, 2, 30, 4, 5, 6

What we have achieved:

- An array object with a flexible size!
- We (almost) designed a new ADT and implemented it in C++
- Exercise: Add a `crop(int)` method to crop the array to specified size
- What do you need to do in that method body?

Extendable Double Array

Now we need an extendable double array.

ExtendableDoubleArray
count : int capacity: int store : double[]
operator[](int index) : double& append(double value) : void ensureCapacity(int cap) : void compact() : void

```
class ExtendableDoubleArray {
  int _count, _capacity; double* store;
  ... // repeat everything, changing int to double
      // except for count and capacity
      // suboptimal!
};
```

Class Templates

- A class template is a blueprint from which multiple individual classes can be constructed.
- The similarities of these classes are written in the template; the differences are reflected as template parameters.
- Templates may take types, constant values and functions as parameters.
- Template parameters may be used inside the body of the template.
- An actual class is created through instantiation of a class template. A class template instantiation may appear wherever a type is expected.
- Template instantiation is textual substitution.

To make ExtendableArray class suitable for any data type, we make it into a template:

ExtendableArray<T>
count : int capacity: int store : T[]
operator[](int index) : T& append(T value) : void ensureCapacity(int cap) : void compact() : void

```
template <typename T> //<class T> is also valid
class ExtendableArray {

  int _count, _capacity;
  T* store;

  void resizeStore() {
    T* newStore = new T[_capacity];
    for (int i = 0; i < _count; i++)
      newStore[i] = store[i];
    delete[] store;
    store = newStore;
  }
```

```
public:

  ExtendableArray(int cap = 10) {
    if (cap < 0) cap = 0;
    _count = 0;
    _capacity = cap;
    store = new T[cap];
  }

  virtual ~ExtendableArray() { delete[] store; }
  int count() const { return _count; }
  int capacity() const { return _capacity; }
  T& operator[](int index) { return store[index]; }

  const T& operator[](int index) const {
    return store[index];
  }
```

```
  void ensureCapacity(int cap) {
    if (_capacity < cap) {
      while (_capacity < cap) (_capacity *= 2)++;
      resizeStore();
    }
  }

  void append(T value) {
    ensureCapacity(_count + 1);
    store[_count++] = value;
  }

  void compact() {
    if (_capacity > _count) {
      _capacity = _count; resizeStore();
    }
  }
};
```

```
int main() {

  ExtendableArray<double> arr(5);

  arr.append(1.1);
  arr.append(2.2);
  arr.append(3.3);
  arr.append(4.4);
  arr.append(5.5); // capacity (5) reached
  arr.append(6.6); // internal store extended to 11

  arr[2] = 30;

  arr.ensureCapacity(50); // extended to 95
  arr.compact(); // internal store shrunk to 6

  return 0;

}
```

Under the hood, when we use `ExtendableArray<double>`, the compiler instantiates the template into this template class:

```
class ExtendableArray<double> {

  int _count, _capacity;

  double *store;

  void resizeStore() {
    int *newStore = new double[_capacity];

  ...

};
```

Advantages and Disadvantages of Class Templates

- The good: we only need to write the code once; we can even use different instantiations in the same program.
- The bad: class templates are not compiled; only when they are instantiated will any compile-time errors be reported.
- The ugly: because the compiler needs the whole template in textual form, we cannot separate the class template definition from class template declaration when compiling (this means: all code in the header file or additional include statements required)

A Better Container

`ExtendableArray<T>::ensureCapacity(int)` may try to create an array too large to handle. Better place a cap on the size of the internal store.

We will also add two more operations, `insert(T, int)` and `remove(int)`, to make `ExtendableArray<T>` more useful.
Design choices:

- Appending or inserting to an ExtendableArray at full capacity: ~~do nothing~~, return false, ~~throw exception~~
- Removing data from an invalid location: ~~do nothing~~, return false, ~~throw exception~~.

```
template < typename T, int MAXCAP = INT_MAX >
class ExtendableArray {

  // private members same as before

public:

  ExtendableArray(int cap = 10) {
    if (cap < 0) cap = 0;
    else if (cap > MAXCAP) cap = MAXCAP;
    _count = 0;
    _capacity = cap;
    store = new T[cap];
  }

  // ~ExtendableArray, count(), capacity(),
  // 2 x operator[], compact() all same as before
```

```
bool ensureCapacity(int cap) {

  if (cap > MAXCAP) return false;

  if (_capacity < cap) {
    while (_capacity < cap) {
      if (_capacity < MAXCAP/2) {
        (_capacity *= 2)++;
      } else {
        _capacity = MAXCAP;
        break;
      }
    }
    resizeStore();
  }

  return true;
}
```

```
bool insert(T value, int index) {

  if (index < 0) index = 0;

  int new_count =
    (index > _count ? index : _count) + 1;

  if (!ensureCapacity(new_count)) return false;

  // shifts elements:
  for (int i = _count; i > index; i--)
    store[i] = store[i - 1];

  // actual insertion:
  store[index] = value;
  _count = new_count;

  return true;
}
```

```
  bool append(T value) {
    return insert(value, _count);
  }

  bool remove(int index) {

    if (index < 0 || index > _count)
      return false;

      /* shifts elements, overwriting the removed: */
    for (int i = index + 1; i < _count; i++)
      store[i - 1] = store[i];
    _count--;
    return true;
  }
};
```

The example above demonstrates:

- Constant value template parameter (`int MAXCAP`);
- Default argument for constant value template parameter: must be constant and known at compile time.

Instantiating this template:

- `ExtendableArray<int, INT_MAX>`
- `ExtendableArray<int>` (same type as above)
- `ExtendableArray<int, 10>`
- `ExtendableArray<int, 11>`

Compiler considers `ExtendableArray<int, INT_MAX>` and `ExtendableArray<int>` as the same type, but all other pairs of the instantiations above are different types!

Now the main function:

```
typedef ExtendableArray<char, 10> TinyArray;

int main() {

  TinyArray arr(1);
  for (char c = 'a'; c < 'i'; c++)
    arr.append(c);

  arr.insert('X', 5);
  arr.insert('Y', 0);
  arr.insert('Z', 2);
  arr.remove(7);
  arr.remove(-3);
  arr.remove(8);

  return 0;
}
```

Action	Result	Count	Cap.	Contents
append('a')	true	1	1	a
append('b')	true	2	3	a,b
append('c')	true	3	3	a,b,c
append('d')	true	4	7	a,b,c,d
append('e')	true	5	7	a,b,c,d,e
append('f')	true	6	7	a,b,c,d,e,f
append('g')	true	7	7	a,b,c,d,e,f,g
append('h')	true	8	10	a,b,c,d,e,f,g,h
insert('X', 5)	true	9	10	a,b,c,d,e,X,f,g,h
insert('Y', 0)	true	10	10	Y,a,b,c,d,e,X,f,g,h
insert('Z', 2)	false	10	10	Y,a,b,c,d,e,X,f,g,h
remove(7)	true	9	10	Y,a,b,c,d,e,X,g,h
remove(-3)	false	9	10	Y,a,b,c,d,e,X,g,h
remove(8)	true	8	10	Y,a,b,c,d,e,X,g

Sorted Lists

Consider a class of data, for every two of which an ordering is defined (`<`, `==`, `>`). We will implement a container that ensures and maintains a sorted list of such elements:

- If `sl` is a newly created sorted list, then `sl` is empty
- `sl.count()` gives the number of data stored in `sl`
- if `0 <= i < sl.count()`, then `sl.get(i)` retrieves the data at position `i` in the list
- the list stores elements in order: `if 0 <= i1 < i2 < sl.count()`, then `sl.get(i1) <= sl.get(i2)`
- `sl.add(d)` adds the value `d` into the list at an appropriate index so that the invariant above is maintained, and causes `sl.count()` to be incremented by 1

- if `0 <= i < sl.count()`, then `sl.removeAt(i)` removes the value at `i` and causes `sl.count()` to be decremented by 1
- `sl.indexOf(d)` returns the index at which `d` is stored in the list `sl`, or `-1` if such data is not found

Design choices for our implementation:

- if `i < 0 or sl.count() <= i`, then `sl.get(i)` returns an `errVal`, ~~throws exception~~
- if `0 <= i < sl.count()`, then `sl.removeAt(i)` removes the element stored at position `i` and then returns true, ~~does nothing in addition~~, otherwise it returns false, ~~throws exception~~.

How can we quickly locate the index to insert a new value, in a list of pre-sorted data?

Binary Search

Task

To quickly locate the index for a new value, without comparing it with every stored data, one by one, from the list.

Idea

Pick the middle value in the list, if new value < mid value, the new value should be inserted into the first half of the list; otherwise it should go to the second half. Because either half of the list is again a sorted list, the process can continue to halve the search space until a single location remains.

Benefit

We only need to search for at most $log(n)$ times, instead of n times (where n is the size of the list). This method also works if the task is to quickly find a value in the list, or check if a value is in the list.

This method is generally known as Binary Search.

Example: value (val) to insert is 8, and we have narrowed down the search space to between head (incl.) and tail (excl.):

	h		m			t	$m = (h + t)/2$
...	1	3	5	7	9	...	$get(m) < val$, so set $h = m + 1$
				h	m	t	$m = (h + t)/2$
...	1	3	5	7	9	...	$get(m) > val$, so set $t = m$
				h m	t		$m = (h + t)/2$
...	1	3	5	7	9	...	$get(m) < val$, so set $h = m + 1$
					h t		$h = t$, location to insert val is found.
...	1	3	5	7	9	...	

Sorted List Implementation

```
template < typename T, int MAXCAP = INT_MAX >
class SortedList :
    private ExtendableArray<T, MAXCAP> {

  typedef ExtendableArray<T> Base;

  int locate(T value) const {
    int head = 0, tail = count();
    while (head < tail) {
      int mid = (head + tail) / 2;
      if ((*this)[mid] < value) head = mid + 1; //X
      else if ((*this)[mid] == value) return mid;
      else tail = mid;
    }
    return head;
  }
```

```
  const T errVal;

public:

  SortedList(int cap = 10, const T ev = 0)
    : Base(cap), errVal(ev) {}

  using Base::count;

  void add(T value) {
    insert(value, locate(value));
  }

  bool removeAt(int index) {
    return remove(index);
  }
```

```
  T get(int index) const {
    if (0 <= index && index < count())
      return (*this)[index];
    return errVal;
  }

  int indexOf(T value) const {
    int index = locate(value);
    if (index < count() &&
        (value == (*this)[index])) return index;
    return -1;
  }
};
```

```
void print(SortedList<char>& sl) {
  cout << "{";
  if (sl.count() > 0) {
    cout << sl.get(0);
    for (int i = 1; i < sl.count(); i++)
      cout << "," << sl.get(i);
  }
  cout << "}" << endl;
}

int main() {

  SortedList<char> sl(10, '@');

  sl.add('c'); print(sl);      // {c}
  sl.add('e'); print(sl);      // {c,e}
  sl.add('a'); print(sl);      // {a,c,e}
  sl.add('b'); print(sl);      // {a,b,c,e}
  sl.add('d'); print(sl);      // {a,b,c,d,e}
  sl.add('c'); print(sl);      // {a,b,c,c,d,e}
```

```
cout << sl.indexOf('b') << endl;  // 1
cout << sl.indexOf('z') << endl;  // -1

cout << sl.get(3) << endl;        // c
cout << sl.get(7) << endl;        // @

cout << (sl.removeAt(1) ? "true" : "false") << endl;
                                  // true
print(sl);                        // {a,c,c,d,e}

cout << (sl.removeAt(7) ? "true" : "false") << endl;
                                  //false
```

```
    print(sl);                          // {a,c,c,d,e}

    sl.removeAt(4); print(sl);          // {a,c,c,d}
    sl.removeAt(0); print(sl);          // {c,c,d}

    return 0;
}
```

This example also demonstrated that:

- template classes can take advantage of inheritance
- use of data member instead of constant value template parameter

The SortedList class template is almost perfect, except when we use a custom class or struct:

```
...
  struct Point {
    int x, y;
    Point(int x = 0, int y = 0)
      : x(x), y(y) {}
  };


  SortedList<Point> slp(10, Point(-1, -1));
/* Compile error:
    < is not defined between Point and Point
   at line X in the SortedList template file    */
```

To allow the consumer of this class template to provide different comparison operations for different instantiations, we can use a function template parameter.

```
template <typename T,
  int (*comp)(const T& a, const T& b),
  int MAXCAP = INT_MAX>
class SortedList :
    private ExtendableArray<T, MAXCAP> {

  typedef ExtendableArray<T> Base;
  int locate(T value) const {
    int head = 0, tail = count();
    while (head < tail) {
      int mid = (head + tail) / 2;
      int cp = comp((*this)[mid], value);
      if (cp < 0) head = mid + 1;
      else if (cp == 0) return mid;
      else tail = mid;
    }
    return head;
  }
  ... // rest of template as before
};
```

```
struct Point {
  ... // as before
};

int compareInt(const int& a, const int& b) {
  return a < b ? -1 : a == b ? 0 : 1; }

int compareChar(const char& a, const char& b) {
  return a < b ? -1 : a == b ? 0 : 1; }

int compareDouble(const double& a, const double& b) {
  return a < b ? -1 : a == b ? 0 : 1; }

int comparePoint(const Point& a, const Point& b) {
  int comp = compareInt(a.x, b.x);
  if (comp != 0) return comp;
  return compareInt(a.y, b.y);
}
```

```
int main() {
  SortedList<int, compareInt> isl;
  SortedList<char, compareChar> csl;
  SortedList<double, compareDouble> dsl;
  SortedList<Point, comparePoint> psl;
}
```

- This solves the Point comparison problem
- However, for every instantiation of SortedList we need to define an equality function
- Worse: most of these equality functions have the same body!
- We can combine these similar functions into a template (function template)

Function Templates

A function template is a blueprint from which multiple individual functions can be constructed.

Additionally, function template instantiation is cleverer: the compiler may be able to figure out the template arguments (template argument deduction):

- by looking at the function parameter list
- the types of function arguments must be clear at compile time

Then we do not need to (but can) specify the template arguments when using the function template.

Makes it possible to use template operators (e.g. `operator<<`). There is no syntax to specifiy template arguments for operators.

```
template <typename T>
int simpleCompare(const T& a, const T& b) {
  return a < b ? -1 : a == b ? 0 : 1; }

int main() {
  char x = 'x', y = 'y';
  struct Point {
    int x, y;
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
  } p1(1, 1);
  cout  << "x vs y: "
        << simpleCompare<char>(x, y) << endl;
  // OUT: x vs y: -1
        << simpleCompare(x, y) << endl;
  // Implicit instantiation: simpleCompare<char>
  // OUT: x vs y: -1
  cout  << "p1.x vs p1.y: "
        << simpleCompare(p1.x, p1.y) << endl;
  // Implicit instantiation: simpleCompare<int>
  // OUT: p1.x vs p1.y: 0
}
```

Now that we have a function template, we can use it as the default argument for the comparison function in `SortedList`.

```
template <typename T>
int simpleCompare(const T& a, const T& b) {
  return a < b ? -1 : a == b ? 0 : 1;
}

template <typename T,
  int (*comp)(const T& a, const T& b) = simpleCompare
  int MAXCAP = INT_MAX>
class SortedList
  : private ExtendableArray<T, MAXCAP> {

  ... // everything same as before

};
```

```
struct Point {

  int x, y;

  Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}

  static int compare(const Point& a, const Point& b){

    int comp = a.x - b.x;
    if (comp != 0) return comp;
    return a.y - b.y;
  }

};
```

```
ostream& operator<<(ostream& o, const Point& p) {
  return o << "P[" << p.x << "," << p.y << "]";
}

template <typename T,
          int (*comp)(const T& a, const T& b),
          int MAXCAP>
void print(SortedList<T, comp, MAXCAP>& sl) {
  cout << "{";
  if (sl.count() > 0) {
    cout << sl.get(0);
    for (int i = 1; i < sl.count(); i++)
      cout << "," << sl.get(i);
  }
  cout << "}" << endl;
}
```

```
int main() {

  SortedList<char> sl(10, '@');
  ... // previous part of demonstration

  SortedList<Point, Point::compare> slp;
  slp.add(Point(3, 2));
  print<Point, Point::compare, INT_MAX>(slp);
    // OUT: {P[3,2]}
  slp.add(Point(2, 1)); print(slp);
    // OUT: {P[2,1],P[3,2]}
  slp.add(Point(2, 3)); print(slp);
    // OUT: {P[2,1],P[2,3],P[3,2]}
  cout << slp.indexOf(Point(2, 1)) << endl;
    // OUT: 0
  cout << slp.get(3) << endl;
    // OUT: P[0,0]

  return 0;
}
```

Default Template Arguments - Summary

- They provide default types, constant values, and functions for template parameters.
- All parameters without defaults should go before those with defaults.
- Template instantiation can omit zero or more arguments, from right to left, where defaults are provided.
- For a class template instantiation, the angled brackets cannot be omitted even if all parameters are provided with defaults. For example, using a class template with all defaults: `template_name<>`

Look-up Tables

Let's see how we might implement a simple look-up table (or map) using templates.

`Map<TKey, TValue>`
`insert(key : TKey, val : TValue) : void`
`get(key : TKey) : TValue*`
`remove(key : TKey) : bool`

- `insert`: adds a mapping from `key` to `val`. If key exists then `val` overrides the previous data, ~~does nothing~~, ~~throw exception~~
- `get`: retrieves a pointer to the value associated with `key`. (Why use `TValue*` instead of `TValue` or `TValue&`?)
- `remove`: removes the mapping where `key` is the key, if any, and returns whether such mapping existed before the removal.

```
template <typename TKey, typename TVal,
    int (*keyComp)(const TKey& a, const TKey& b)
    = simpleCompare>
class Map {

  // Struct to represent a Key-Value pair
  struct KVP {

    TKey key; TVal val;

    KVP(TKey _key) : key(_key) {}
    KVP(TKey _key, TVal _val)
      : key(_key), val(_val) {}

    static int comp(KVP* const& a, KVP* const& b) {
      return keyComp(a->key, b->key);
    }
  };
```

```
  SortedList<KVP*, KVP::comp> base;

  int locate(TKey key) const {
    KVP dummy(key);
    return base.indexOf(&dummy);
  }

public:

  virtual ~Map() {
    for (int i = 0; i < base.count(); i++) {
      delete base.get(i);
    }
  }
```

```
  void insert(TKey key, TVal val) {
    int idx = locate(key);
    if (idx == -1) base.add(new KVP(key, val));
    else base.get(idx)->val = val;
  }

  TVal* get(TKey key) const {
    int idx = locate(key);
    if (idx == -1) return nullptr;
    return &(base.get(idx)->val);
  }

  bool remove(TKey key) {
    int idx = locate(key);
    if (idx == -1) return false;
    delete base.get(idx);
    base.removeAt(idx); return true;
  }
};
```

The example demonstrates:

- Nested class/struct: logically expresses ownership or other close relations between outer class and nested class.
- `const` keyword to the right of *: constant pointer; reading types from right to left.
- Another way to implement *in-terms-of* relationships (see private inheritance): data member (composition).

```
class Course {
  const int _code;
  const char* const _name;

public:
  Course(int code, const char* name)
    : _code(code), _name(name) {}

  int code() const { return _code; }
  const char* name() const { return _name; }
};

ostream& operator<<(ostream& o, const Course& c) {
  return o << "Course " << c.code()
           << ": "       << c.name();
}
```

```
class Textbook {
  const int _id;
  const char* const _title;

public:
  Textbook(int id, const char* title)
    : _id(id), _title(title) {}

  int id() const { return _id; }
  const char* title() const { return _title; }

};

ostream& operator<<(ostream& o, const Textbook& tb){
  return o <<"Textbook " << tb.id()
           << ": " << tb.title();
}
```

```
int courseComparer(const Course* const& a,
                   const Course* const& b) {
  return a->code() - b->code();
}

int main() {
  Course* oo = new Course(517, "OO Design & Prog"),
        * cs = new Course(515, "Computer Systems"),
        * lg = new Course(518, "Logic & AI Prog");

  Textbook* ps =
          new Textbook(11, "Problem Solving w/ C++"),
    *ca = new Textbook(33, "Computer Architecture"),
    *cd = new Textbook(55, "C++ for Dummies");

  Map<const Course*, Textbook*, courseComparer> ctm;
```

```
ctm.insert(oo, ps);
ctm.insert(cs, ca);

Textbook** result = ctm.get(oo);
cout  << "[" << result << "] "
      << **result << endl;
// OUT: [0x???] Textbook 11: Problem Solving ...

result = ctm.get(cs);
cout  << "[" << result << "] "
      << **result << endl;
// OUT: [0x???] Textbook 33: Struct. Comp. Org'n

result = ctm.get(lg);
cout << "[" << result << "] " << endl;
// OUT: [0x0]
```

```
    ctm.insert(oo, cd);
    result = ctm.get(oo);
    cout  << "[" << result << "] "
          << **result << endl;
    // OUT: [0x???] Textbook 55: C++ for Dummies

    bool rem = ctm.remove(lg);
    cout << (rem ? "true" : "false") << endl;
    // false
    rem = ctm.remove(cs);
    cout << (rem ? "true" : "false") << endl;
    // true
    result = ctm.get(cs);
    cout << "[" << result << "] " << endl;
    // OUT: [0x0]

    delete oo; delete cs; delete lg;
    delete ps; delete sc, delete cd;        return 0;
}
```

Templates Summary

- Templates are blueprints of classes/functions. Compiler generates actual classes/functions by textual substitution.
- Template declaration takes 3 kinds of parameters. Templates can provide default arguments to their parameters.
- Class template instantiation is explicit; function template instantiation can be inferred.
- Instantiated entities from the same template are equivalent (i.e. have the same type) if the arguments supplied are the same.
- Templates and inheritance can be combined.
- ADTs make heavy use of templates to avoid code repetition.
- Java and C# Generics are the successors to C++ templates.