ECE326

PROGRAMMING LANGUAGES

Lecture 14 : C++ Class Template

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Class Survey

- Need more examples
- Examples are hard to follow
  - Laser pen ordered
- Need extra reading
- Practice midterm
  - In class, Friday 25th, solution will be posted afterwards
  - Tuesday 29th class will be for review only
- More correlation between assignment and lecture
  - Subsequent assignments (2 and onward) are related

Class Template

- Template for creating classes
- Typically used to implement generic constructs
  - E.g. containers
- Reuse *source code* instead of *object code*
- Allows full spectrum of C++ features
  - Inheritance
  - Dynamic dispatch
  - Ad-hoc polymorphism

Generic Stack

```
template<class T> // same as typename T
class Stack {
public:
    Stack(unsigned cnt=8);
    ~Stack() { delete [] array; }
    bool push(const T &); // push element to top of stack
    bool pop();           // remove current top of stack
    T * top();            // return current top of stack
    int empty() const { return end == -1; }
    int full() const { return end == count - 1; }
private:
    int count;  // capacity of stack
    int end;   // points to top of stack
    T * array;
} ;
```

Member Functions

- Can be defined outside of class template
  - But must be defined together, preferably in same header file

```
template<class T>
Stack<T>::Stack(unsigned cnt) : count((int)cnt), end(-1),
    array(new T[cnt]) {}

template<class T>
bool Stack<T>::push(const T & item) {
    if (full()) return false;
    array[++end] = item;
    return true;
}
```

Member Functions

```
template<class T>
bool Stack<T>::pop() {
    if(empty()) return false;
    --end;
    return true;
}

// general rule of thumb:
// return a pointer if you want to indicate error with nullptr
// return by reference (T &) if no error is possible, OR
// if you plan to raise exception upon error
template<class T>
T * Stack<T>::top() {
    if (empty()) return nullptr;
    return &array[end];
}
```

typedef

- Creates an alias for the type name
- Common use
  - Indicate different usage
    - E.g. size_t instead of unsigned long confer context
  - reduce length of type names, or improve appeal
- Again, do not abuse this feature

```
/* this is bad – hides the fact that it’s a pointer */
typedef char * cstring;
typedef Stack<double> DoubleStack;
typedef vector<int, allocator<int> > IntVector;
typedef int (* compare_f)(int, int);
```

Sample Use

```
DoubleStack ds = DoubleStack();
double * dp, d = 1.1;
cout << "Pushing elements onto stack" << endl;
while (ds.push(d)) {
    cout << d << ' '; d += 0.7;
}
cout << endl << "stack full" << endl
     << endl << "Popping elements from stack" << endl ;
while ((dp = ds.top())) {
    cout << *dp << ' '; ds.pop();
}
cout << endl << "stack empty" << endl ;
```

```
Pushing elements onto stack
1.1 1.8 2.5 3.2 3.9 4.6 5.3 6
stack full

Popping elements from stack
6 5.3 4.6 3.9 3.2 2.5 1.8 1.1
stack empty
```

Generic Container

- Duplicate code if many template instantiations
  - Can cause code bloat – large executable size
- Goal
  - Can we use a generic Stack implementation, and use templates to provide type-safety?
- Approach
  - Mix template and inheritance
- Note
  - the following example actually worsens executable size

Generic Stack

```
class Generic {
public:
    Generic(unsigned size, void (*init)(char *),
            void(*dest)(char *), unsigned cnt=8);
    ~Generic();
    /* pop, empty and full are omitted – same implementation */

private:
    void(*dest)(char *); // function pointer to destructor
    int end;             // points to top of stack
    int size;            // size of each element

protected:
   int count;            // capacity of stack
   char * buffer;
   void * push();    // returns address to place new element
   void * top();     // return current top of stack
};
```

Generic Stack Constructor

```
Generic::Generic(unsigned size, void (* init)(char *),
                 void (* dest)(char *), unsigned cnt)
      : dest(dest)
      , end(-1)
      , size(size)
      , count(cnt)
      , buffer(new char[size*cnt])
 {
      char * ptr = buffer;
      for (int i = 0; i < count; i++) {
            // calls constructor for each element
            init(ptr);
            ptr += size;
      }
 }
```

Generic Stack Destructor

```
Generic::~Generic()
{
	char * ptr = buffer;
	for (int i = 0; i < count; i++) {
		/* calls destructor for each element */
		dest(ptr);
		ptr += size;
	}

	delete [] buffer;
}
```

Generic Stack Functions

```
// returns address to place new element
void * push() {
    if (full()) return nullptr;
    return buffer + (++end)*size;
}

// return current top of stack
void * top() {
    if (empty()) return nullptr;
    return buffer + end*size;
}
```

Stack Template

```
template<class T>
class Stack : public Generic {
   static void initialize(char * ptr);
   static void destroy(char * ptr);

public:
    Stack(unsigned cnt=8) : Generic(sizeof(T),
        &Stack<T>::initialize, &Stack<T>::destroy, cnt) {}

    bool push(const T & elem) {
      T * item = (T *)Generic::push();
      if (item == nullptr) return false;
      *item = elem; // invokes copy constructor
      return true;
    }

    T * top() {
        return (T *)Generic::top();
    }
};
```

Placement New

- Calling constructor of type at preallocated address
  - When used, destructor must be called manually

```
template<class T>
void Stack<T>::initialize(char * ptr) {
    new (ptr) T();
}

template<class T>
void Stack<T>::destroy(char * ptr) {
    T * item = (T *)ptr;
    item->~T();
}
```

Sample Use

```
typedef Stack<Point> PointStack;
PointStack ps;
Point * pp;
int i = 1;
Point p(i, i);
cout << "Pushing elements onto stack" << endl;
while (ps.push(p)) {
    cout << p << ' ';
    p = Point(i, i);
    i++;
}
cout << endl << "stack full" << endl;
```

Assume a class named Point with members x and y

```
Pushing elements onto stack
(1, 1) (2, 2) (3, 3) (4, 4) (5, 5) (6, 6) (7, 7) (8, 8)
stack full
```

Static Members

- Static member functions
  - Function shared across all instances of same class
  - Does **not** have a `this` pointer
- Static member variable
  - Variable shared across all instances of same class
- Static function variable
  - Has global lifetime but has same scope as local variables
- Each template class/function has its own copy of static members, not shared across templates!

Static Class Members

```
class Puppy {
    static int num_puppies;
public:
    Puppy() { num_puppies++; }
    ~Puppy() { num_puppies--; }
    static void status() {
        cout << “I have ” << num_puppies << “ puppies” << endl;
    }
};

int Puppy::num_puppies = 0;     // instantiate static variable
…
Puppy a, b, c, d;
Puppy::status();

I have 4 puppies
```

Static Function Variable

- Persists through function calls
- Initialized once (unless changed by function)
- Can be useful if your function cannot return null
  - Must instead return default or error value

```
template<typename T, int C>
class List {
    T array[C];
public:
    const T & operator[](int idx) {    // makes elements read-only
        static T defval = T();
        if (idx < 0 || idx >= C) return defval;
        return array[idx];
    }
};
```

Static Assertion

- Compile time check for certain guarantees
  - Helps you find bugs at compile time rather than runtime

```
#include <type_traits>  // includes definition for is_base_of

struct Shape {
      virtual double area() const=0;
};

template<typename S>
class Box {
public:
    Box() { static_assert(std::is_base_of<Shape, S>::value,
                          "S must inherit from Shape"); }
};

/* static assertion failed: S must inherit from Shape */
class Box<Foo> foobox;
```

Default Template Parameter

- Template parameters can take defaults too

```
template<class T=int, int C=16>
class Stack : public Generic {
   static void initialize(char * ptr);
   static void destroy(char * ptr);

public:
    Stack() : Generic(sizeof(T),
        &Stack<T>::initialize, &Stack<T>::destroy, C) {}
    …
};

/* creates an integer stack with capacity of 16 */
Stack<> stack;
```

Partial Specialization

- Specialize only some of the template parameters

```
template<class K, class V, int S>
class Map {
    struct Pair {
        K key; V value;
    } array[S];
public:
    …
    V * find(const K & k) {
        int i;
        for (i = 0; i < S; i++) {
            if (array[i].key == k) return &array[i].value;
        return nullptr;
    }
};
```

Partial Specialization

- Specialize only some of the template parameters

```
template<class V, int S>
class Map<int, V, S> {
    V values[S];
public:
    …
    V * find(int k) {
        if (k < 0 || k >= S)
            return nullptr;
        return &values[i];
    }
};
```

Template Aliases

- Similar to typedef, except for partial templates
- Not the same as partial specialization
  - Only gives a different name to templates

```
template<class K, class V, int S>
class Map { … };

template<class K, int S>
using StringMap = Map<K, std::string, S>;
```