H.O. #9
Fall 2015
Gary Chan

Generic Programming: Overloading and Templates

N:9; D:6,11,14

Outline

- Function overloading
- Operator overloading and copy constructor
  - An example on string
- Function templates
- Class templates

Function Overloading

- Overloaded functions have
  - Same name
  - Different sets of parameters
- Compiler selects proper function to execute based on the number, types and order of arguments in the function call
- Commonly used to create several functions of the same name that perform similar tasks, but on different data types and numbers of parameters

overload.cpp (1/2)

```cpp
// function square for int values
int square( int x )
{
   cout << "square of integer " << x << " is ";
   return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
   cout << "square of double " << y << " is ";
   return y * y;
} // end function square with double argument
```

- Defining a square function for ints and doubles

overload.cpp (2/2)

```
int main()
{
   cout << square( 7 ); // calls int version
   cout << endl;
   cout << square( 7.5 ); // calls double version
   cout << endl;
   return 0; // indicates successful termination
} // end main
```

- Sample Output

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

- Output confirms that the proper function was called in each case

More on Function Overloading

- C++ allow function overloading

```
#include <stdio.h>

  int max(int a, int b) {
    if (a > b) return a;
    return b;
  }

  char *max(char *a, char * b) {
    if (strcmp(a, b) > 0) return a;
    return b;
  }

  int main() {
    printf("max(19, 69) = %d\n", max(19, 69));
// or, cout << "max(19, 69) = " << max(19, 69) << endl;
    printf("max(abc, def) = %s\n", max("abc", "def"));
// or, cout << "max("abc", "def") = " << max("abc", "def") << endl;
    return 0;
  }
```

Function Overloading

- How the compiler differentiates overloaded functions:
  - Overloaded functions are distinguished by their signatures
    - Compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage
  - The above type-safe linkage ensures that
    - Proper overloaded function is called
    - Types of the arguments conform to types of the parameters
- Creating overloaded functions with identical parameter lists and different *return* types is a compilation error
  - It is ambiguous on which function to call

Operator Overloading

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context/class objects
- Examples
  - `<<`
    - Stream insertion (output for cout)
    - bitwise left-shift (`i<<2`) `i` modified
  - +, -, *, /
    - Performs arithmetic on multiple items (integers, floats, etc.)
- Global (external) functions:

```
// Global function overloading
ostream & operator<<( ostream &, const foo & ); // prototype to output foo
foo & operator<<( foo & , int  );  // prototype to shift foo
                                   // return the shifted object for concatenation

istream & operator>>( istream &, foo & );   // input a foo object
foo & operator>>( foo & bf, int i );  //right-shift overloading to modify foo obj.
```

Operator Overloading

- Types for operator overloading
  - Built in (int, char) or user-defined (classes)
  - Can use existing operators with user-defined types/objects
  - Cannot create new operators
- Overloading operators
  - Create a function for the class
- Name of operator function
  - Keyword `operator` followed by the symbol
  - Example
    - **`operator+`** for the addition operator +

Using Operators on Class Objects

- Overloading provides concise and intuitive notation

  object2 = object1.add(object2); *vs.* object2 = object1 + object2;

- The operators must be overloaded for that class
- Default operations
  - Assignment operator (=)
    - Member-wise assignment between objects
  - Address operator (&)
    - returns address of object
  - Can be overloaded/overruled by the programmer

Restrictions on Operator Overloading

- Cannot change
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order of operators
  - Associativity (left-to-right or right-to-left)
    - 2*3*4 (=6*4) vs. 2^3^2 (=2^9)
  - Number of operands
    - e.g., !, `&` or `*` is unary, i.e., can only act on one operand as in `&i` or `*ptr`
  - How operators act on built-in/primitive data types (i.e., cannot change integer addition)
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading + and = does not mean having overloaded +=

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded							
.	.*	? :					

***Member functions* declaration:**

```
bool operator!() const;
bool operator==( const foo & ) const;
bool operator<( const foo & ) const;
bool operator!=( const foo &right ) const;
bool operator>( const foo &right ) const;
bool operator<=( const foo &right ) const;
bool operator>=( const foo &right ) const;
```

Operator Functions as *Class* Members

- *Leftmost* object must be of the same class as operator function
- Use `this` keyword to explicitly get left operand argument
- Operators `()`, `[]`, `->` or some other assignment operator *must* be overloaded as a *class member* function
  - While I/O operators >> and << must be overloaded as external function
- Called when
  - Left operand of binary operator is of this class
  - Single operand of unary operator is of this class

```
 // subscript operator: can be both modifiable lvalue and rvalue
foo & operator[]( int );  // [] can have only ONE parameter
                          // may return the object at the index

// subscript operator: if you only want it to be an rvalue
foo operator[]( int ) const; // constant member function for constant object
                            // return a value, and hence cannot be a lvalue
                           // May also be const foo & operator[]( int ) const;
                           // NOT foo & operator[]( int ) const; (compile error)

foo operator()( int, int = 0 ) const; // () can have any number of parameters
                                      // this has at most 2 parameters
```

More on Function Overloading

```
 // subscript operator: can be both modifiable lvalue and rvalue
foo & operator[]( int );  // return an index of the object to modify

// subscript operator: if you only want it to be an rvalue
foo operator[]( int ) const; // constant member function cannot be lvalue
                         // May also be const foo & operator[]( int ) const;
                         // NOT foo & operator[]( int ) const; (compile error)
```

- Normally you cannot have two overloading functions with the same parameters set. But there is an exception when the functions differ by a `const` in the prototype.
  - We call it const overloading (the prototypes above)
- The compiler will decide on which function to use based on the type of the calling object.
  - If it is `const` object, the const member function will be used. Sometimes we need this if we want to use a different member function for a const object.
  - If it is not a `const` object, the non-const function will be used, no matter whether it is a rvalue or lvalue

```
#include <iostream>
using namespace std;

class B{
public:
  const int& operator [] (int i) const{
    cout << "Constant [] function is called" << endl;
    return _data[i];
  }
  int& operator [] (int i){
    cout << "Non-constant [] function is called" << endl;
    return _data[i];
  }
private:
  int _data[10];
};

int main(){
  B b1;
  const B bc = b1;

  b1[0];
  b1[2] = bc[2];
  cout << bc[1] << endl;
  return 0;
}
```

```
Non-constant [] function is called (for b1[0])
Non-constant [] function is called (for b1[2])
Constant [] function is called (for bc[2])
Constant [] function is called (for bc[1])
0
```

Operator Functions as *Global* Functions

- Need parameters for *both* operands
- Can have object of different class
- Can be a **`friend`** to access **`private`** or **`protected`** data
- Both << and >> must be global functions
  - Cannot be class members
- Overloaded << operator
  - Left operand is of type `ostream &`
    - Such as **`cout`** object in **`cout << classObject`**
- Similarly, overloaded >> has left operand of `istream &`
  - Such as **`cin`** object in **`cin >> classObject`**

Global Functions: Commutative Operators

- May want + to be commutative
  - So both “a + b” and “b + a” work
- Suppose we have two different classes
  - If the overloaded operator is a member function, then its class is on left
- HugeIntClass + long int
  - Can be member function for HugeIntClass
- HugeIntClass + HugeIntClass
  - Can be member function as well
- long int + HugeIntClass
  - For this to work, + must be a global overloaded function
- If the function returns a local variable, return its value, NOT its reference

```
HugeInt operator+( long, const HugeInt & ); //function overloading, MUST be global
          // returning the value, i.e., the sum, because it is a local variable
HugeInt operator+( const HugeInt &, long );  // may be a const member function
HugeInt operator+( const HugeInt &, HugeInt ); // may be a const member function
```

Overloading Unary Operators

- Can overload as member function with no arguments
- Can overload as global function with *one* argument
  - Argument must be class object or reference to class object
- If *member* function, needs no arguments
  - `bool operator!() const;`
- If *global* function, needs one argument
  - `bool operator!( const foo & ),` i.e., `!f` becomes `operator!(f)`

Overloading Binary Operators

- Member function: one argument
  - `const foo & operator+=( const foo & ); //rvalue`
  - `s1 += s2;  // a string`
  - `s1 += s2 += s3;  // same as s1 += ( s2 += s3 );`
  - `(s1 += s2) += s3; // compiler yells as it tries to`
    `// modify a constant object returned by s1+=s2`
- Global function: two arguments
  - One of the arguments must be class object or reference
  - `const foo & operator+=( foo &, const foo & );`
    `// no const for the first argument as the obj is modified`
  - `y += z` becomes `operator+=( y, z )`
- Note that `int` type provides an extra variant of lvalue:

```
int i=2,j=4;
(j +=i) += 2;   // return the new j(i.e., 6), which adds to 2
cout << i << j; // output 2 8
```

Converting between Types

- If you want to convert a *primitive* type to the class: `foo = i`
  - Overload the assignment operator with integer, e.g., as a member function supporting concatenation:

    ```
    const foo_class & operator=( int i ); // assignment operator
    ```
- Casting: Convert a non-primitive class to a primitive type or another object
  - Compiler provides cast for primitive types, e.g., integers to floats, etc.
  - May need to convert between user-defined types or class
    - For implicit and explicit casting
    - `i = (int) foo; // same as i = static_cast< int > (foo);`
- Cast operator (conversion operator)
  - Convert from
    - A class to a built-in type (int, char, etc.)
    - A class to another class
  - Do *not* specify return type, but return the type to which you are converting
  - Must be non-static member function

Cast Operator Example

- Prototype

  `A::operator char *() const;`

  - Casts class A to a temporary char *
  - `static_cast< char * >( s )` calls `s.operator char *()`
    - Same as `(char *) s`
  - `static_cast< float >( obj )` calls `obj.operator float()`
    - Same as `(float) obj`
  - You may also use
    - `foo::operator int() const; //cast to int`
    - `foo::operator barClass() const; //e.g., bar_obj = (barClass) foo_obj;`
- Casting can sometimes prevent the need for overloading
  - Suppose the self-defined class `String` is only cast to `char *`
  - Then `cout << s; // s is a String:` Compiler implicitly converts s to char * for output, i.e., `cout << (char *) s;`
  - Do not have to overload <<

```
# include <iostream>
# include <string>
using namespace std;

class record{
public:
  record( string str = "" ){
    name = str;    key = str.size();
  }
  operator int() const {  // NO return type
    return key;
  }
  operator double() const {
    return key*2.1;
  }

private:
  int key;  // key of the string
  string name;  // and some other fields
};

int main(){

  record r1, r2("COMP"), r3("152");
  int k;

  cout << (int) r1 << endl;        // cast r1 to int
  cout << (double) r2 << endl;     // cast r1 to double
  k = r2;                  // implicit casting to int as k is integer
  cout << k << endl;

  if( (int) r3 < k )  // casting r3 to int
    cout << "r3 < k\n";

  return 1;
}
```

```
0
8.4
4
r3 < k
```

Copy Operations

- Assignment operator (=) can be used to assign an object to another of the *same* type
  - Each data member of the right object is assigned to the same data member in the left object
  - By definition, = or += modifies the object and hence it can NOT be a `const` function

```
// Below is a rvalue
// allow a = b = c,, which always means a = (b = c)
// because = is right associative (i.e., = in this
// statement is always rvalue).
// One can write (a=b)=c, if a=b returns a lvalue
// With below, cannot write (a = b) = c
const foo & operator=( const foo & );

// concatenation operator, below must be a rvalue
// allow s1 += s2 += s3, which always means s1 += (s2 += s3)
// with below, cannot write (s1+=s2)+=s3
const String & operator+=( const String & );
```

Copy/Assignment Operator

- `operator=` **may also return a non-constant reference, e.g.,**

`foo & operator=( const foo &)`

- **In this case, it can be both an lvalue and rvalue**
- **E.g.,**

```
int i=5, j=4;

(i=j)=3;           // lvalue, return the new i
cout << i<< j ;   // get 3 4

(i=3)=4;           // lvalue, return the new i
cout << i<< j ;   // get 4 4

 i = j = 3;       // get 3 3
```

Case Study: String Class

- Build class String
  - String creation, manipulation
  - Similar to class string in standard library
  - Character index starts at 0
  - Does NOT count the NULL terminator
- Conversion constructor
  - Any single-argument constructor
  - Turns objects of other types into class objects
  - Example: String s1("happy");
    - Creates a String from a char *
- Overloading function calls
- String.h, String.cpp, stringtester.cpp

stringtester.cpp Sample Output (1/3)

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s1 += s2 yields s1 = happy birthdayhappy birthday
```

stringtester.cpp Sample Output (2/3)

```
s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthdayhappy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: appy birthday to you
Copy constructor: appy birthday to you
Destructor: appy birthday to you
The substring of s1 starting at
location 15, s1(15), is: appy birthday to you
```

The constructor and destructor are called for the temporary String

stringtester.cpp Sample Output (3/3)

```
Destructor: appy birthday to you
Copy constructor: happy birthdayhappy birthday to you

*s4Ptr = happy birthdayhappy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthdayhappy birthday to you
Destructor: happy birthdayhappy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthdayhappy
birthday to you

Attempt to assign 'd' to s1[30] yields:
Destructor: happy
Destructor:  birthday
Destructor: Happy Birthdayhappy birthday td you
```

Overloading ++ and --

- Increment/decrement operators can be overloaded
- Prefix increment: ++x
- Postfix increment: x++
- Suppose we want to add 1 to a Date object d1
- Member-function prototype for *prefix* increment
  - ```
    Date & operator++(); // return a reference for
                         // successive operation: y=++++x
    ```
  - `++d1` becomes `d1.operator++()`
- Global-function prototype for *prefix* increment
  - ```
    Date & operator++( Date & );
    ```
  - `++d1` becomes `operator++( d1 )`

Postfix Increments

- *Postfix* increment has a dummy integer parameter
  - d++ // d++0
  - An int with value 0
- Overload `operator++` with different function parameters
- Member-function prototype for postfix increment:
  - ```
    foo operator++( int ); // MUST be rvalue, as it
                           // returns the old value of the object
    ```
  - `d1++` becomes `d1.operator++( 0 )`
- The value returned is a temporary variable inside the function, i.e., it does not and cannot return a reference
  - Can NOT use d++++ to increment d twice (because it is the value of a temporary variable returned, not the incremented object)
- Global-function prototype for postfix increment
  - `foo operator++( foo &, int );`
  - `d1++` becomes `operator++( d1, 0 )`
    ```

Overloading ++ and --

```
Date & operator++();      // prefix increment operator, ++Date
Date operator++( int );   // postfix increment operator, Date++
```

- Return values
  - *Prefix* increment (++x)
    - Increment and then return the *incremented* object
    - Return by reference `(Date &)` so that we can use ++++x to increment x multiple times
    - Theoretically can be a lvalue (i.e., can be assigned like ++x = 4), though we almost never use it as lvalue
    - Note that `y = ++x` assigns y the *value* of ++x, NOT making y an alias of x!
  - *Postfix* increment (x++)
    - Store x in a temporary object, increment x, and then return the temporary object
    - Returns by value (Returns temporary object with the object's old value)
    - Must be rvalue (must be on right side of assignment as it returns a value), e.g.,

      ```
      y = x++; // not x++=4;
      ```
- All this applies to decrement operators as well

Case Study: A Date Class

- Overloaded increment operator
  - Change day, month and year
- Function to test for leap years
- Function to determine if a day is last of month
- Date.h, Date.cpp, datetester.cpp

Return Types of Member Function

- A function may return
  - the value of an object,
  - a constant value of an object,
  - a reference to an object, or
  - a constant reference to an object
- For good programming practice, if you are to return a function's *local* variable, use return by value or constant value.
  - Both cannot be lvalue
  - Returning a value or constant value allows cascading operations on the object. In such case, the function should not modify the object because modifying a temporary object is meaningless
  - E.g., operator+, which returns the local value of the sum for later cascading (to support `a+b-c`)
  - If you return a constant value and want to do cascading call, you can only call its constant functions
- If you are returning a permanent object, usually one uses return by reference or constant reference.
  - Save copying overhead upon function return
  - Allows cascading operations on the object
  - E.g., operator+=, which returns `*this` as a reference

Function Returns

- Returning a value *may* call the copy constructor to copy the returned value to some temporary location upon the exit of the function
  - Compiler dependent – the compiler may do some optimization on function return so as to minimize calling copy constructor
- Returning a reference or a constant reference does NOT call the copy constructor
  - More efficient
  - However, remember NOT to return a local variable in your function as a reference!
- `return_func.cpp`

```
# include <iostream>
using namespace std;

class foo{
public:
  foo(){ cout << "constructor\n"; }
  foo( const foo & f ){ cout << "copy
constructor\n"; }
};

// return the VALUE of the object
foo bar1(){
  foo f;
  return f;
}

// Meaningless, though allowed
const foo bar2(){
  foo f;
  return f;
}

//can be rvalue or lvalue
foo & bar3( foo & f ){
  return f;
}

// must be rvalue
const foo & bar4( foo & f){
  return f;
}

int main(){
  foo f, g;
```

```
 cout << "\nreturn foo by bar1\n";
 bar1() = f;      // No compilation error but no use

 cout << "\nreturn const foo by bar2\n";
 f = bar2();     // foo2() = f is an error

 cout << "\nreturn foo & by bar3\n";
 f = bar3( g );

 cout << "\nreturn const foo & by bar4\n";
 f = bar4( g ); // foo4( g ) = f is error

 return 0;
}
```

```
constructor (for foo f in main)
constructor (for g in main)

return foo by bar1
constructor (for foo f in bar1)
copy constructor (Unix has that, but not
Linux)

return const foo by bar2
constructor (for foo f in bar2)
copy constructor (Unix has that, but not
Linux)

return foo & by bar3

return const foo & by bar4
```

Function Templates and Class Templates

Evolution of Reusability and Genericity

- Major theme in development of programming languages
  - Reuse code to avoid repeatedly reinventing the wheel
- Trend contributing to this
  - Use of generic code
  - Can be used with different *types* of data
- Function and class templates
  - Enable programmers to specify an entire range of related functions and related classes
  - → Generic programming

Function Templates

- Used to produce overloaded functions that perform identical *operations/algorithms* on different types of data
  - Programmer writes a single function-template definition
  - Compiler generates separate object-code functions (function-template specializations) based on *argument types* in calls to the function template

Function Genericity: Overloading and Templates

- Initially code was reusable by encapsulating it within functions
- Swap example:

```
void swap (int & first, int & second)
{
  int temp = first;
  first = second;
  second = temp;
}
```

  - Then call `swap(x,y);`

Function Genericity: Overloading and Templates

- To swap variables of different types, write another function
  - Overloading allows functions to have same name
  - Signature (types and numbers of parameters) keep them unique to the compiler
- This could lead to a library of swap functions
  - One function for each standard/primitive type
  - Compiler chooses which to use from signature
- But … what about swapping user-defined types such as an object?
  - We cannot cover the swap function for ALL possible class objects

Passing Types (Instead of Fixed Types)

- Using function overloading, note how similar each of the swap functions would be
  - The three places where the type is specified
- What if we passed the type somehow?!!
- Templates make this possible
  - Declare functions that receive both data and types via parameter
- Thus code becomes more *generic*
  - Easier to reuse and extend to other types

Function Templates

- More compact and convenient form of overloading
  - Identical program logic and operations/algorithms for each data type
- Function template definition
  - Written by programmer once
  - Essentially defines a whole family of overloaded functions
  - Begins with the `template` keyword
  - Contains template parameter list of formal type parameters for the function template enclosed in angle brackets (<>)
  - Formal type parameters
    - Preceded by keyword `typename` or keyword `class`
    - Placeholders for fundamental types or user-defined types

Writing Template

- A function template is a <u>pattern</u>
  - describes how specific functions is constructed
  - constructed based on given actual types
  - type parameter said to be "bound" to the actual type passed to it
- Calling a function with template type inside the function
  - ```
    template <class T>
    void foo( void ){T a; …}
    //called with foo<int>();
    ```

```
void Swap(______ & first, ______ & second)
{
    ________ temp  = first;
    first = second;
    second = temp;
}
```
```

General Form of Template

```
template <typename TypeParam>
```

FunctionDefinition

where

- `TypeParam` is a type-parameter (placeholder) naming the "generic" type of value(s) on which the function operates
- *FunctionDefinition* is the definition of the function, using type `TypeParam`

Swap Function Template

```
template <typename ElementType>
void Swap(ElementType &first, ElementType &second)
{
   ElementType hold = first; // need to overload copy constructor
   first = second; // need to overload assignment operator
   second = hold;
}
```

- `<typename ElementType>` names `ElementType` as a type parameter

Template Instantiation

- In and of itself, the template does nothing
- When the compiler encounters a template
  - it stores the template
  - but doesn't generate any machine instructions or codes
- When a function template is instantiated
  - Compiler finds type parameters in list of function template
  - For each type in the function parameter list, type of corresponding argument is determined
  - These two function type and argument type are then bound together
- E.g., when it encounters a call to `Swap()`
  - Example: `Swap(int1, int2);`
  - it generates an integer instance of `Swap()`
- The type will be determined …
  - by the compiler (at compilation time)
  - from the type of the arguments passed when `Swap()` is called
- Cannot specify data type at run time

.h and .cpp Files for Template Functions

- Three files: `foo.h` (template declaration), `foo.cpp` (template definition), and `main.cpp` (using template functions)
- The compiler, in the compilation of `foo.cpp` to object codes, has to know the data type input into the template functions, and replace all the template occurrences by the actual data type
  - Therefore, the *callers* of the template functions have to be known at compile time. This is different from the non-template functions where the compiler does not need to know the callers to generate proper object codes.
  - That means `main.cpp` has to include both `foo.cpp`, and `foo.h`
  - That also means `foo.h` and `foo.cpp` have to be combined into one single file
- A function template cannot be split across files for separate compilation
  - Specification/declaration and implementation/definition usually are in the <u>same file</u>
  - This sometimes causes some inconvenience in makefiles (need to combine .h and .cpp)

displayarray.cpp

```
template <typename ElementType>
void display(ElementType array[], int numElements) {
   for (int i = 0; i < numElements; i++)
      cout << array[i] << "  ";
   cout << endl;
}

int main() {
   double x[] = {1.1, 2.2, 3.3, 4.4, 5.5};
   display(x, 5);
   int num[] = {1, 2, 3, 4};
   display (num, 4);
}
```

display<double> created

display<int> created

- Function-template specializations are generated automatically by the compiler to handle each type of call to the function template
- If an array of user-defined objects is used, need to overload << operator of the object class.
- Sample Output

```
1.1 2.2 3.3 4.4 5.5
1 2 3 4
```

Template Function Example: maximum.h and maximum.cpp

- Sample Output

```
Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C
```

Class Templates

- Recall our Stack class:

```
const int STACK_CAPACITY = 128;
typedef int StackElement;
class Stack
{
 /***** Function Members *****/
 public:
    . . .
 /***** Data Members *****/
 private:
   StackElement myArray[STACK_CAPACITY];
   int myTop;
};
```

- How did we create a new version of a stack for a different type of element?
  - Change the meaning of `StackElement` by merely change the type following `typedef`

What's wrong with typedef?

- Changes the header file
  - Any program that uses this must be recompiled → inconvenient and time-consuming
- A name declared using `typedef` can have only *one* meaning
  - What if we need *two* stacks of *different types* in the same program?
- We would like to pass the type into the class object

Type-Independent Container for Stack

- Use a class template:
  - the class is parameterized
  - it receives the type of data stored in the class via a parameter (like function templates)

```
const int STACK_CAPACITY = 128;
template <typename StackElement>
class Stack
{
  /***** Function Members *****/
  public:
     . . .
  /***** Data Members *****/
  private:
    StackElement myArray[STACK_CAPACITY];
    int myTop;
};
```

StackElement is a "blank" type (a type placeholder) to be filled in later

General Form Of Class Template Declaration

```
template <typename TypeParam >
class SomeClass
{
      // ... members of SomeClass ...
};
```

- More than one type parameter may be specified:

```
template <typename TypeParam1,...,typename TypeParamN>
class SomeClass
{
      // ... members of SomeClass ...
};
```

- As opposed to template function, the argument parameter does NOT have to appear in the class
  - E.g., `template<typename T> class bar{}; // ok`

Instantiating Class Templates

- Instantiate it by using declaration of form

  `ClassName<Type> object;`

- Passes Type as an argument to the class template definition

```
Stack<int> intSt;
Stack<string> stringSt;
```

- Compiler will generate two distinct definitions of `Stack`
  - two instances: one for `int` and one for `strings`

Rules For Class Templates

1. Definitions of member functions outside class declaration *must* be function templates
   - E.g., `template< typname T > foo_class<T>::…`
2. All uses of class name as a type must be parameterized with <…>
   - E.g., `foo_class<T>…`
3. Member functions must be defined in the *same* file as the class definition
   - Same reason as in function template (i.e., compiler needs to know the exact data types at calling to generate appropriate object codes at compile time)

Applying the Rules to Our Stack Class

1. Each member functions definition preceded by

```
template <typename StackElement>
```

2. The class name Stack preceding the scope operator (::) is used as the name of a type, and must therefore be parameterized

```
template <typename StackElement>
void Stack<StackElement>::push(const StackElement
& value)
{ /* ... body of push() ... */ }
```

3. Specification, implementation in the same file

Applying the Rules to `friend` Functions

- Consider the addition of a friend function `operator<<`
- Inside the `Stack` class, if the parameter is of type `Stack`, it must be parameterized. For example:

```
template<class U>
friend ostream & operator<<(ostream & out,
const Stack<U>& st);
```

- Non-member (global) functions must be defined as a function template:

```
template<typename StackElement>
ostream & operator<<(ostream & out,
        const Stack<StackElement> & st)
{  . . . }
```

Stack Class Template

- Application of all these principles
  - A Stack class template (Stack.h)
  - Note that there is *not* a separate .cpp file
- Templates may have more than one type parameter
- Thus possible to specify a `Stack` class differently
  - Could specify with a dynamic array and pass an integer for the capacity

Nontype Parameters and Default Types

- Nontype template parameters
  - Nontype parameter: those primitive types, not a generic type
  - Can have default arguments
  - Are treated as consts to generate machine codes
  - Template header: `template< typename T, int elements >`
  - Declaration: `Stack< double, 100 > salesFigures;`
- Type parameters can have default arguments too
  - Template header: `template< typename T = string >`
  - Declaration: `Stack<> jobDescriptions; // default to string`

Explicit Template Specializations

- Looks like overloading
- Used when a particular type will not work with the general template or requires customized processing
- Example for an explicit `Stack< Employee >` specialization, where `Employee` is a defined class

```
template<>
class Stack< Employee >
{
        // tailor-made implementation here …
};
```

- This is a complete replacement for the general template
    - This does not use anything from the original class template and can even have different members

Template Specialization: An Example

```
template< typename T >
class bar{  // general class
};

template<>
class bar<int>{  // specialized class
public:
  void hello( void );
};

// Do NOT put template<> here
void bar<int>::hello( void ){
  cout << "hello world" << endl;
}

int main(){
  bar<int> a;  // specialized class
  a.hello();
  return 1;
}
```

Templates and Static Members

- Each class-template specialization has its *own* copy of each *static* data member
  - All objects of that specialization share that one static data member
  - static data members must be defined and, if necessary, initialized at file scope
- Each class-template specialization gets its own copy of the class template's static member functions
- See `special_template.cpp` for default and static members