Lecture 5.1 – Data Structures

Static & Run time Stacks

Abstract Data Type (ADT)

Abstract data types (ADT) – declaration of data; declaration of operations. Users of ADT are not concerned how a task is done, but what the ADT can do.

ADT includes set of definitions allowing the programmers to use the functions while *hiding the implementation*.

Types

What is a type?

1. Set of possible values.
2. Operations on those values.
3. Properties.

Example: Integer (`int`) type

Values : `-2147483648` to `2147483647`

Operations : `+`, `-`, `*`, `/`, `++`, `--`, etc.

Properties : Closed under `+`, `-`, `*`, `/`, `++`, `--` (i.e., `3 + 5 = 8`)

`2147483647 + 1 == -2147483648`

Abstract Data Type

A behavior-level description of the operations on and properties of some data/values.

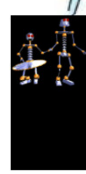
Values (or data): Instance(s) of some type/class.

Operations: Services provided by the ADT on the data.

Properties: What is known about the data.

- As the definition indicates, an ADT is mainly concerned with behavior (i.e., operations/services).

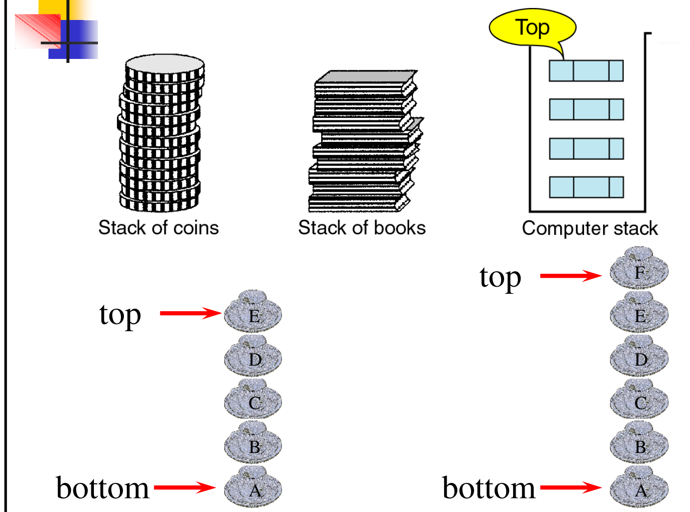
Stacks



Stacks

- Specifies a LIFO (Last-In, First-Out) interface.
- Sometimes called a "Push-down" stack:
  - Similar in concept to the spring-loaded, push-down stack of plates at a buffet-style restaurant.
  - A plate is added to the stack by placing it on top (pushing down the other plates).
  - A plate is removed from the stack by taking it off the stack (and the plates underneath it pop up one position).

5



6

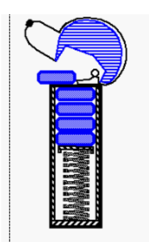
Abstract Data Type (ADT)

A *stack* is a sequence of elements in which the only element that can be removed or accessed/modified is the element that was most recently inserted.

Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.

Inserting an item is known as "pushing" onto the stack. "Popping" off the stack is synonymous with removing an item.

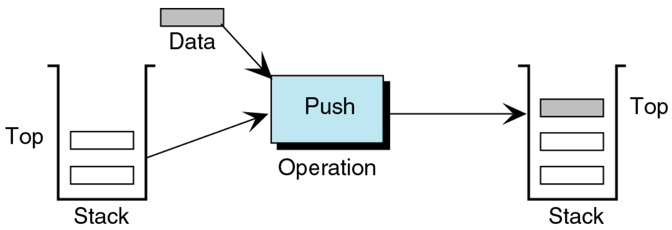
A PEZ® dispenser as an analogy:



7

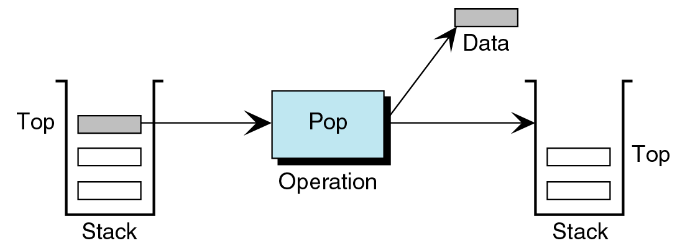
- A stack is an abstract data type (ADT) that supports two main methods:
  - push(o): Inserts object o onto top of stack
  - pop(): Removes the top object of stack and returns it; if the stack is empty, an error occurs
- The following support methods should also be defined:
  - size(): Returns the number of objects in stack
  - isEmpty(): Return a boolean indicating if stack is empty.
  - top(): Return the top object of the stack, without removing it; if the stack is empty, an error occurs.

8



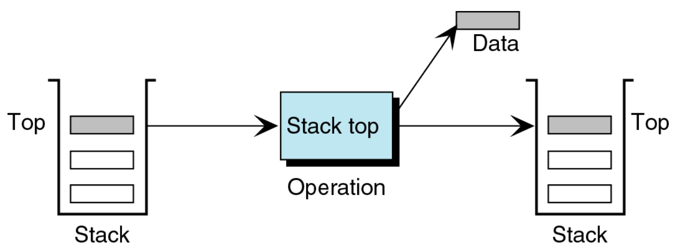
Data
Top
Push
Operation
Top
Stack
Stack

9



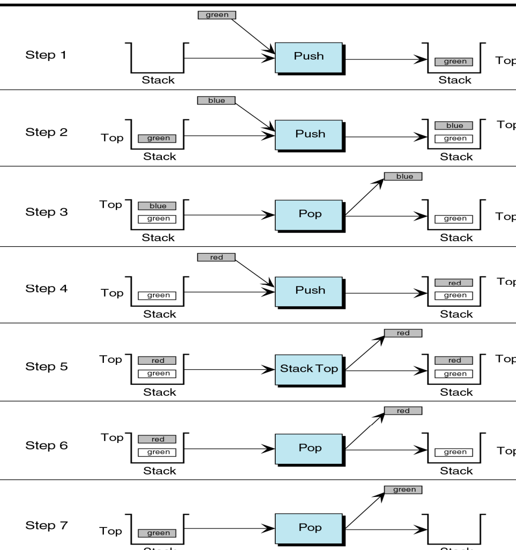
Data
Top
Pop
Operation
Top
Stack
Stack

10



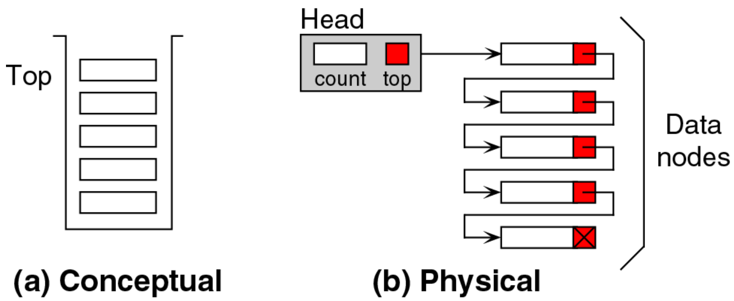
Data
Top
Stack top
Top
Operation
Stack
Stack

11



Step 1
green
Push
green
Top
Stack
Stack
Step 2
blue
Top
green
Push
blue
green
Top
Stack
Stack
Step 3
blue
Top
blue
green
Pop
green
Top
Stack
Stack
Step 4
red
Top
green
Push
red
green
Top
Stack
Stack
Step 5
red
Top
red
green
Stack Top
red
green
Top
Stack
Stack
Step 6
red
Top
red
green
Pop
green
Top
Stack
Stack
Step 7
green
Top
green
Pop
Stack
Stack

12



Definition of a stack as an ADT (abstract data type):

A stack is an *ordered collection of data items* in which *access* is possible only at one end, called the *top* of the stack.

So, we can begin the declaration of our class by selecting *data members*:

- Provide an *array* data member to hold the *stack elements.*
- Provide a *constant data* member to refer to the *capacity* of the array.
- Provide an *integer* data member to indicate the *top of the stack*.

Problems:

- We need an array declaration of the form
  ArrayElementType myArray[ARRAYCAPACITY];
- — What type should be used?
  Solution (for now): Use the **typedef** mechanism:
  typedef int StackElement;
  // put this before the class declaration. *StackElement* is now a *synonym* for *int*.
- — What about the capacity?
  const int STACK_CAPACITY = 128;
  // put this before the class declaration so it is easy to change when necessary
- — Then declare the array as a data member in the private section:
  StackElement myArray[STACK_CAPACITY];

Notes:

- The *typedef* makes *StackElement* a *synonym* for *int*. Putting it outside the class makes it accessible throughout the class and in any file that *#includes "Stack.h"*. If in the future we want a stack of *reals*, or characters, or . . ., we need only *change the typedef*:

  typedef double StackElement;

  or

  typedef char StackElement;

or . . .

- When the *class library is recompiled*, the type of the array's elements will be *double* or *char* or . . .

Example 1: Class Stack

```
/* Stack.h provides a Stack class.
 *
 * Basic operations:
 * Constructor:  Constructs an empty stack
 * empty:   Checks if a stack is empty
 * push:    Modifies a stack by adding a value at the top
 * top:     Accesses the top stack value; leaves stack unchanged
 * pop:     Modifies a stack by removing the value at the top
 * display: Displays all the stack elements
 * Class Invariant:
 * 1. The stack elements (if any) are stored in positions
 *    0, 1, . . ., myTop of myArray.
 * 2. -1 <= myTop <= STACK_CAPACITY -1
------------------------------------------------------------------*/
```



```
#include <iostream>
using namespace std;
#ifndef STACK
#define STACK
const int STACK_CAPACITY = 128;
typedef int StackElement;
class Stack
{
/***** Function Members *****/
public:
/* --- Constructor ---
 *
 * Precondition:  A stack has been declared.
 * Postcondition: The stack has been constructed as an
 *                empty stack.
 ************************************************/
Stack();
```



```
/* --- Is the Stack empty? ---
 * Receive: Stack containing this function (implicitly)
 * Returns: True if the Stack containing this function is empty
 *****************************************************/
bool empty() const;
/* --- Add a value to the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 *          A value to be added to a Stack
 * Return:  The Stack (implicitly), with value added at its
 *          top, provided there's space
 * Output:  "Stack full" message if no space for value
 ************************************************/
void push(const StackElement & value);
/* --- Display values stored in the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 *          The ostream out
 * Output:  The Stack's contents, from top down, to out
 ***************************************/
void display(ostream & out) const;
```



```
/* --- Return value at top of the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 * Return:  The value at the top of the Stack, if nonempty;
 *          else a "garbage value"
 * Output:  "Stack empty" message if stack is empty
 ****************************************/
StackElement top() const;
/* --- Remove value at top of the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 * Return:  The Stack containing this function with its top
 *          value (if any) removed
 * Output:  "Stack-empty" message if stack is empty.
 *****************************************/
void pop();
```

```
/***** Data Members *****/
 private:
  StackElement myArray[STACK_CAPACITY];
  int myTop;
};        // end of class declaration
//--- Definition of Class Constructor
inline Stack::Stack()
{ myTop = -1; }
//--- Definition of empty
inline bool Stack::empty() const
{ return (myTop == -1); }
#endif
```


Example 1: Implementation of Class Stack

```
/* Stack.cpp -- implementation file for Stack.h */
#include "Stack.h"
#include <iostream>
using namespace std;
//--- Definition of push()
void Stack::push(const StackElement & value)
{
  if (myTop < STACK_CAPACITY - 1)  // Preserve stack invariant
  {
    ++myTop;
    myArray[myTop] = value;
  }                    // or simply, myArray[++myTop] = value;
  else
    cerr << "*** Stack is full -- can't add new value ***\n"
         << "Must Increase value of STACK_CAPACITY in Stack.h\n";
}
```



```
//--- Definition of display()
void Stack::display(ostream & out) const
{
  for (int i = myTop; i >= 0; i--)
    out << myArray[i] << endl;
}
//--- Definition of top()
StackElement Stack::top() const
{
  if (myTop >= 0)
    return (myArray[myTop]);
  cerr << "*** Stack is empty -- returning 'garbage value' ***\n";
  return myArray[STACK_CAPACITY -1];
}
//--- Definition of pop()
void Stack::pop()
{
  if (myTop >= 0)    // Preserve stack invariant
    myTop--;
  else
    cerr << "*** Stack is empty -- can't remove a value ***\n";
}
```



```
/* Stack tester 1
 ****************************************/
#include "Stack.h"        // our own -- <stack> for STL version
#include <iostream>
using namespace std;
int main()
{
  Stack s;
  cout << " s empty? " << s.empty() << endl;
  for (int i = 1; i <= 128; i++) s.push(i);
    cout << "Stack should now be full\n";
  s.push(129);
}
```

```
/* Stack tester  2
*******************************************/
#include "Stack.h"          // our own -- <stack> for STL version
#include <iostream>
using namespace std;
int main()
{
  Stack s;
  cout  << "s empty? " << s.empty() << endl;
  for (int i = 1; i <= 4; i++) s.push(2*i);
  cout << "Stack contents:\n";
  s.display(cout);
  cout << "s empty? " << s.empty() << endl;
  cout << "Top value:  " << s.top() << endl;
  while (!s.empty())
   {
     cout << "Popping " << s.top() << endl;
   s.pop();
   }
  cout << "s empty? " << s.empty() << endl;
}
```


Example 2: Class Stack // Stack2.h

```
//      ItemType : the class definition of the objects on the stack.
//      MAX_ITEMS: the maximum number of items on the stack.
const int MAX_ITEMS = 5;
typedef int ItemType;
class StackType
{
public:
   StackType();
   // Class constructor.
   void MakeEmpty();
   // Function:  Sets stack to an empty state.
   // Post: Stack is empty.
   bool IsFull() const;
   // Function: Determines whether the stack is full.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is full)
```



```
 bool IsEmpty() const;
   // Function: Determines whether the stack is empty.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is empty)
   void Push(ItemType item);
   // Function: Adds newItem to the top of the stack.
   // Pre:  Stack has been initialized and is not full.
   // Post: newItem is at the top of the stack.
   void Pop(ItemType& item);
   // Function: Removes top item from the stack and returns it in item.
   // Pre:  Stack has been initialized and is not empty.
   // Post: Top element has been removed from stack.
   //       item is a copy of the removed item.
private:
   int top;
   ItemType  items[MAX_ITEMS];  // items is a pointer to an item
};
```


Definitions of Stack Operations

- *MakeEmpty* and the class *constructor* should set *top* to –1.
- *IsEmpty* should compare *top* to –1.
- *IsFull* should compare top with MAX_ITEM – 1.

Example 2: Implementation of Class Stack

```
// The function definitions for the non-templated, non-dynamic storage-
// allocation version of class StackType.
#include "Stack2.h"
StackType::StackType()
{
   top = -1;
}
void StackType::MakeEmpty()
{
   top = -1;
}
bool StackType::IsEmpty() const
{
   return (top == -1);
}
```

29

```
bool StackType::IsFull() const
{
   return (top == MAX_ITEMS-1);
}
void StackType::Push(ItemType newItem)
{
   top++;
   items[top] = newItem;
}
void StackType::Pop(ItemType& item)
{
   item = items[top];
   top--;
}
```

30

- Before we call *Push* , we must make sure that the stack is not full.

  if (!stack.IsFull( ))

  stack.Push(item) ;
- If the stack is *full* when we invoke *Push*, the resulting condition is ***stack overflow***. *Error checking for overflow* conditions may be *handled* in a number of *different ways*.
- We could check for overflow inside *Push* (instead of making the calling program do the test). We would need to tell the caller whether or not the *Push was possible*, which we could do by adding a *bool* variable *overFlow* to the *parameter list*. Here is the revised algorithm.
- **Push (Checks for Overflow)**

  IF stack is full

  Set *overFlow* to *true*

  ELSE

  Set *overFlow* to *false*

  Increment *top*

  Set *items[top]* to *newItem*
- Which version of *Push* to use in a program depends on the *specifications*.
- Our Stack *ADT specification* uses the first version because the *precondition* for *Push* states that the *stack is not full*.

- **Stack Overflow** The condition resulting from trying to push an element onto a full stack.
- To invoke *Pop* as implemented here, the caller must first test for an *empty stack*:

  if (!stack.IsEmpty ( ))

  stack.Pop(item) ;
- If the *stack is empty* when we try to *pop* it, the resulting condition is called ***stack underflow***. Just as we can test for *overflow* within the *Push* operation, we could test for *underflow* inside the *Pop* function. The algorithm for *Pop* would have to be modified slightly to return a *bool* value *underFlow* in addition to the *popped* item.
- **Stack Underflow** The condition resulting from trying to *pop* an empty stack.

32

Function Template

Forms:

template < typename TypeParam >

Function

or

template < class TypeParam >

Function

More General Form:

template < specifier TypeParam1, ..., specifier TypeParamn >

Function

Notes:

- The word *template* is a C++ keyword specifying that what follows is a *pattern* for a function.
- The keyword *typename* and *class* may be used *interchangeably* in a *type-parameter list*.
- Unlike regular functions, a *function template cannot be split across files*, that is, we cannot put its prototype in a *header* file and its definition in an *implementation* file. It *all goes in the header file.*
- In the general form, *each of the type parameters* **must** appear *at least once in the parameter list of the function*. The reason for this is that the *compiler uses only the types of the arguments in a function call to determine what types to bind to the type parameters.*
- The process of constructing a function is called **instantiation**. In each *instantiation*, the *type parameter* is said to be ***bound*** to the *actual type* passed to it.

Instantiation of a function template is carried out using the following *algorithm*:

- *Search the parameter list of the template function for type* parameters.
- If a type parameter is *found*, *determine* the *type* of the corresponding *argument*.
- *Bind these two types together*.

Example 3: Displaying an Array // Function Template

```
/* This program illustrates the use of a function template to
 * display an array with elements of any type for which << is
 * defined.
 *
 * Output:  An array of ints and an array of doubles using Display()
 ******************************************************/
#include <iostream>
using namespace std;
/* Function template to display elements of any type
 * (for which the output operator is defined) stored
 * in an array.
 *  Receive: Type parameter ElementType
 *           Array of elements of type ElementType
 *           numElements, number of elements to be displayed
 *  Output:  First numElements in array
 ***************************************************/
```

```
template <class ElementType>
void Display(ElementType array[], int numElements)
{
  for (int i = 0; i < numElements; i++)
    cout << array[i] << " ";
  cout << endl;
}
int main ()
{
  double x[10] = {1.1, 2.2, 3.3, 4.4, 5.5};
  Display(x, 5);
  int num[20] = {1, 2, 3, 4};
  Display (num, 4);
}
```

Problem: implement ***swap*** *using templates*

37

Class Templates

Example 4: Class Stack Template // StackT.h

```
/* StackT.h provides a Stack template.
 *  Receives:  Type parameter StackElement
 *  Basic operations:
 *  Constructor:  Constructs an empty stack
 *  empty:   Checks if a stack is empty
 *  push:    Modifies a stack by adding a value at the top
 *  top:     Accesses the top stack value; leaves stack unchanged
 *  pop:     Modifies a stack by removing the value at the top
 *  display: Displays all the stack elements
 *  Class Invariant:
 *  1. The stack elements (if any) are stored in positions
 *     0, 1, . . ., myTop of myArray.
 *  2. -1 <= myTop <= STACK_CAPACITY -1
 ------------------------------------------------------------------*/
```

38

```
#include <iostream>
using namespace std;
#ifndef STACKT
#define STACKT
const int STACK_CAPACITY = 128;
template <class StackElement>
class Stack
{
/***** Function Members *****/
public:
/* --- Constructor ---
 *
 * Precondition:  A stack has been declared.
 * Postcondition: The stack has been constructed as an
 *                empty stack.
 ********************************************/
Stack( );
/* --- Is the Stack empty? ---
 * Receive: Stack containing this function (implicitly)
 * Returns: True if the Stack containing this function is empty
 ************************************************/
bool empty( ) const;
```

39

```
/* --- Add a value to the stack ---
 *
 * Receive:  The Stack containing this function (implicitly)
 *           A value to be added to a Stack
 * Return:   The Stack (implicitly), with value added at its
 *           top, provided there's space
 * Output:   "Stack full" message if no space for value
 ************************************************/
void push(const StackElement & value);
/* --- Display values stored in the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 *          The ostream out
 * Output:  The Stack's contents, from top down, to out
 ************************************************/
void display(ostream & out) const;
/* --- Return value at top of the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 * Return:  The value at the top of the Stack, if nonempty;
 *          else a "garbage value"
 * Output:  "Stack empty" message if stack is empty
 ************************************************/
StackElement top( ) const;
```

40

```
/* --- Remove value at top of the stack ---
 *
 * Receive: The Stack containing this function (implicitly)
 * Return:  The Stack containing this function with its top
 *          value (if any) removed
 * Output:  "Stack-empty" message if stack is empty.
 ********************************************/
void pop( );
/***** Data Members *****/
 private:
  StackElement myArray[STACK_CAPACITY];
  int myTop;
};        // end of class declaration
```



```
//--- Definition of Class Constructor
template <class StackElement>
inline Stack<StackElement>::Stack( )
{ myTop = -1; }
//--- Definition of empty
template <class StackElement>
inline bool Stack<StackElement>::empty() const
{ return (myTop == -1); }
//--- Definition of push()
template <class StackElement>
void Stack<StackElement>::push(const StackElement & value)
{
  if (myTop < STACK_CAPACITY - 1)  // Preserve stack invariant
  {
    ++myTop;
    myArray[myTop] = value;
  }                 // or simply, myArray[++myTop] = value;
  else
    cerr << "*** Stack is full -- can't add new value ***\n"
         << "Must Increase value of STACK_CAPACITY in Stack.h\n";
}
```

```
//--- Definition of display()
template <class StackElement>
void Stack<StackElement>::display(ostream & out) const
{
  for (int i = myTop; i >= 0; i--)
    out << myArray[i] << endl;
}
//--- Definition of top()
template <class StackElement>
StackElement Stack<StackElement>::top() const
{
  if (myTop >= 0)
    return (myArray[myTop]);
  cerr << "*** Stack is empty -- returning 'garbage value' ***\n";
  return myArray[STACK_CAPACITY -1];
}
//--- Definition of pop()
template <class StackElement>
void Stack<StackElement>::pop()
{
  if (myTop >= 0)   // Preserve stack invariant
    myTop--;
  else
    cerr << "*** Stack is empty -- can't remove a value ***\n";
}
#endif
```


Driver program for Stack class Template

```
#include "StackT.h"
#include <iostream>
using namespace std;
int main()
{
  Stack<int> intSt;          // stack of ints
  Stack<char> charSt;        // stack of chars
  int i;
  for (i = 1; i <= 4; i++)
    intSt.push(i);
  while (!intSt.empty())
  {
    i = intSt.top(); intSt.pop();
    cout << i << endl;
  }
  for (char ch = 'A'; ch <= 'D'; ch++)
    charSt.push(ch);
  charSt.display(cout);
}
```

Example 5: // Stack2T.h

```
#ifndef Stack2T_H
#define StackT_H
const int MAX_ITEMS = 5;
template<class ItemType>
class StackType
{
public:
    StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType item);
    void Pop(ItemType& item);
private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

45

```
// The function definitions for class StackType in file Stack2.h.
template<class ItemType>
StackType<ItemType>::StackType()
{
    top = -1;
}
template<class ItemType>
void StackType<ItemType>::MakeEmpty()
{
    top = -1;
}
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
    return (top == -1);
}
template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
    return (top == MAX_ITEMS - 1);
}
```

46

```
template<class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
    top++;
    items[top] = newItem;
}
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
    item = items[top];
    top--;
}
#endif
```

47

Class Templates

What is wrong with *typedef*

- **Problem 1:** Since changing the *typedef* is a *change to the header file*, any program or library that uses the class must be *recompiled.*
- **Problem 2:** Suppose we need a *Stack* of *reals* and a *Stack* of *characters* (two different containers with different types of values). A name declared using *typedef* can have *only one meaning at a time*. We need to create *two different container classes with two different names.*
- **Definition 1: Generic Data Type** A type for which the *operations are defined* but the *types of the items* being manipulated are *not*.
- **Definition 2: Template** A C++ language *construct* that allows the *compiler to generate multiple versions of a **class type*** or a ***function*** by allowing parameterized types.
- **Definition 3: Container or Collection Type** A data type that is designed to hold other objects.

48

Class Template Declaration

Forms:

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

or

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

More General Form

template <specifier TypeParam1, …, specifier TypeParamn>
class SomeClass
{
// . . . . members of SomeClass
} ;

- Each specifier is the keyword *typename* or *class*.
- *TypeParam, TypeParam1 …* are *generic type* parameters naming types of data to be stored in the container class.

Notes:

- The keyword *template* specifies that what follows is *a pattern for a class*, *not an actual class declaration.*
- The keyword *typename* and *class* may be changed interchangeably in a *type-parameter list*.
- Unlike regular classes, a *class template cannot be split across files*, that is, we cannot put prototypes of functions in the header file and their definitions in an implementation file. It all goes in the header file.
- A class template is only a pattern that describes how individual classes can be constructed from given actual types. This process of creating a class is called *instantiation*. This is accomplished by attaching the actual type to the class name in the definition of an object:

  SomeClass<Actual_Type> object ;
- For example, we could instantiate our Stack class template with the definitions

  Stack<char> charStack ;
  Stack<double> dubStack ;
- When the compiler process these declarations, the *compiler will generate two distinct* ***Stack*** *classes* – two instances – one with *StackElement* replaced by *char* and the other with *StackElement* replaced by *double*. The constructor in the first class will construct *charStack* as an empty stack of *characters* and the constructor in the second class will construct *dubStack* as an empty stack of *doubles*.

Three Important Rules for Building Class Templates:

- All operations defined outside of the class declaration must be template functions.
- Any use of the *name of a template class as a type must be parameterized.*
- *Operations* on a template class should be *defined in the same file as the class declaration.*

A Stack Class Template

- If we define the functions *Stack( ), empty( ), push( ), display( ), top( ),* and *pop( )* outside the class declarations, the above three rules must be followed:

Rule 1: They must be defined as function templates.

This means that these definitions must each be preceded by a template declaration; for example,

template <class StackElement>

// definition of constructor

template <class StackElement>

// definition of empty( )

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

```
template <class StackElement>
inline void Stack<StackElement> : : Stack( )
{
    //   body of constructor
}
template <class StackElement>
inline bool  Stack<StackElement> : : empty( )
{
    //   body of empty
}
```


Rule 3: These definitions should be placed within the same file StackT.h

An Alternative Version of the *Stack* Class Template

- Templates may have *more than one type parameter* and also *ordinary value parameters*. So the *capacity* of the *myArray* in the *stack* declaration may be passed into the *Stack* class template as an *ordinary **int** parameter* along with the stack element *type in the template* declaration:


```
/*  Stack.h provides a Stack template
*  Receives:   Type parameter StackElement
*          Integer myCapacity
*  Basic Operation :
*************************************  */
#include <iostream.h>
#ifndef STACKT
#define STACKT
template <class StackElement, int myCapacity>
class Stack
{
/***** Function Members *****/
public:
// --- Prototypes of member and friend functions
/***** Data Members *****/
 private:
  StackElement myArray[myCapacity];
  int myTop;
};
// Definitions of member and friend functions
#endif
```


- This will allow the definitions of stacks like in the following in client programs:

```
Stack<int, 10> intSt ;          // stack of at most 10 ints
Stack<string,  3>  strSt ;      // stack of at most 3 strings
```

- At compile time, the compiler generates *(instantiates) two distinct class types* by *substituting the actual parameter for the formal* parameter throughout the class template. The actual parameter can be the name of any data type, *built-in or user-defined, not a variable name.*
- Note that *passing a parameter to a template* has an effect at compile time, whereas passing a *parameter to a function* has an effect at a *run time*.
- The compiler cannot instantiate a *function template* unless it knows the *actual parameter* to the template, and this *actual parameter is found in the client code*. Different compilers use different mechanisms to solve this problem.
- *One general solution* is to *compile the client code and the member functions at the same time*. A common technique is to *place both the class definition and the member function definitions into the same file of type .h file.*
- *Another technique* is to give the *include directive for the implementation of the file at the end of the header file.*

```
// File "stack2.h" contains the class definition for the Stack ADT.
// The class is templated; the array of values is statically allocated.
#ifndef Stack2_H
#define Stack2_H
const int MAX_ITEMS = 5;
template<class ItemType>
class StackType
{
public:
    StackType( );
    void MakeEmpty( );
    bool IsEmpty( ) const;
    bool IsFull( ) const;
    void Push(ItemType item);
    void Pop(ItemType& item);
private:
    int myTop;
    ItemType items[MAX_ITEMS];
};
#include "stack2.cpp"
#endif
```

Example:

Note: Either way, when the *client code* specifies *#include "StackType.h"*, the compiler has all the source code – both the *member functions* and the *client code* – available to it at once.

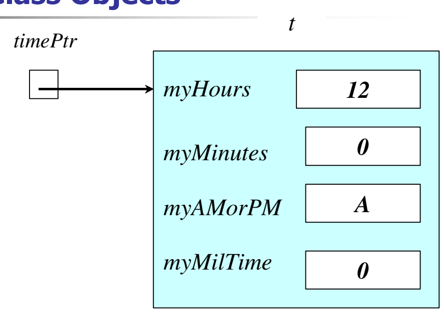
57

Dynamically Allocated Stacks

58

Pointers to Class Objects

Time t ;
*Time *timePtr = &t ;*

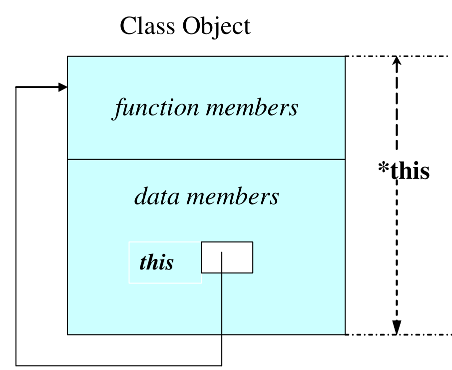


Hour( ) can be called using *timePtr* in two ways.
One way is to combine the indirection operator with the *dot operator*
*(*timePtr).Hour( )*
The other way is to use **class pointer selector operator** (→)
timePtr → Hour( ) ;
ptr → member is *equivalent* to the expression *(*ptr).member*

59

The *this* Pointer

- In every class the keyword **this** is a pointer whose value is the address of the object. The value of the dereference pointer ***this** is the object itself.



60

- To return the object we can use

  *return *this ;*

- Normally, a return statement in a function

  return object ;

- first uses the *copy constructor* to built in a copy of object, and then *returns this copy*.
- We can *force it to return object itself* by making the function's return-type a reference:

  ReturnType& functionName(parameters) ;

Example:

```
Time& Time : : setTime( int h, int m, int s)
{
    hour = (h >= 0 && h < 24) ? h : 0 ;
    minute = (m >= 0 && m < 60) ? m : 0 ;
    second = (s >= 0 && s < 60) ? s : 0 ;
    return *this ;
}
```

t.setTime(11, 59, 12).display(cout)

is evaluated as

(t.setTime(11, 59, 12)).display(cout)

Deallocating a Run-Time Array

We can use the **delete** operation in a statement of the form

delete pointerVariable ; or

delete[ ] arrayPtr;

For example:

```
int *intPtr = new int ;
delete intPtr ;
```

Good programming practice:

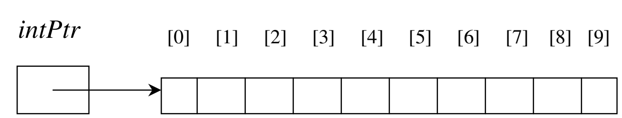
```
delete intPtr ;
intPtr = 0 ;
```

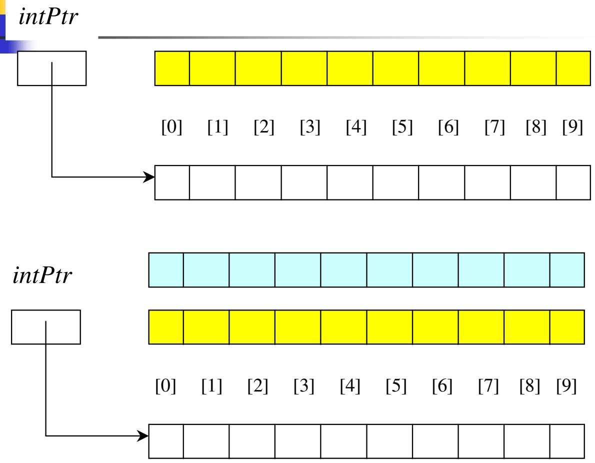
Similarly:

```
cin >> numValues ;
double *dPtr = new double[numValues] ;
delete [ ] dPtr ;
dPtr = 0 ;
```


Memory Leaks.

```
do
{
    int *intPtr = new int[10] ;
    assert(intPtr != 0) ;
// ... use the array via intPtr to solve a problem
    cout << "\nDo another (y or n) ?  " ;
    cin >> answer ;
}
while (answer != 'n') ;
```





65

```
do
{
    int *intPtr = new int[10] ;
    assert(intPtr != 0) ;
// ... use the array via intPtr to solve a problem
    delete [ ] intPtr ;
cout << "\nDo another (y or n) ?  " ;
    cin >> answer ;
}
while (answer != 'n') ;
```

66

Run-Time-Allocation in Classes

1. ***Destructors:*** To deallocate the memory.
2. ***Copy constructors:*** To make copies of objects (e.g., value parameters)
3. ***Assignment:*** To assign one storage structure to another.

67

```
//***** RTStack.h *****
#ifndef RTSTACK
#define RTSTACK
#include <iostream>
using namespace std;
template <class StackElement>
class Stack
{
/***** Function Members *****/
public:
. . .
/***** Data Members*****/
private:
StackElement * myArrayPtr;      // run-time allocated array
int myCapacity,                 // capacity of stack
myTop;                          // top of stack
};
// Definitions of member (and friend) functions
#endif
```

68

/* --- **Class constructor** ---
Precondition: A stack has been defined.
Receive: Integer numElements > 0; (default = 128)
Postcondition: The stack has been constructed with capacity numElements.
---*/

```
Stack(int numElements = 128);
//*** Definition of class constructor
template <class StackElement>
Stack<StackElement>::Stack(int numElements)
{
   assert (numElements > 0);              // check precondition
   myCapacity = numElements;              // set stack capacity
   // allocate array of this capacity
   myArrayPtr = new StackElement[myCapacity];
   if (myArrayPtr == 0)                   // check if memory available
   {
      cerr << "*** Inadequate memory to allocate stack ***\n";
      exit(-1);
   }            // or assert(myArrayPtr != 0);
   myTop = -1;
}
```

In main( )

```
cin >> num;
Stack<double> s1, s2(num);
```

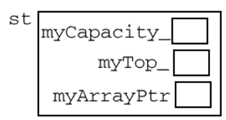

//*** **Definition of *push()***

```
template <class StackElement>
void Stack<StackElement>::push(const StackElement & value)
{
   if (myTop < myCapacity - 1)
   {
++myTop;
myArrayPtr[myTop] = value;
   }           // or simply, myArrayPtr[++myTop] = value;
   else
cerr << "*** Stack is full -- can't add new value ***\n";
}
```

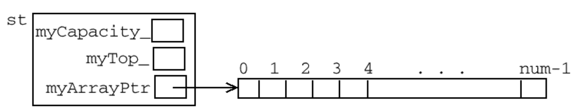

Class Destructor

In main( ):

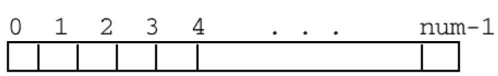
Stack<double> st(num);



- The array to store *stack elements* is not *allocated until run-time*:



- When the life-time of *st* ends, the memory allocated to *myCapacity_*, *myTop_*, and *myArrayPtr* is automatically reclaimed, but not for the run-time allocated array:



We must add a **destructor** to the class to *avoid the memory leak*.

- *Destructor's role: Deallocate memory* allocated at run-time (the opposite of the constructor's role).
- At any point in a program where an object goes out of scope, the compiler inserts a call to this destructor. That is:

When an object's lifetime is over, its destructor is called first.

Form of destructor:

- Name is the class name preceded by a tilde (~).
- It has *no arguments or return type*

~ClassName( )

```
//***** RTStack.h *****
. . .
/* --- Class destructor ---
Precondition: The lifetime of the Stack containing this function ends.
Postcondition: The run-time array in the Stack has been deallocated.
-----------------------------------------------------------------*/
~Stack();
// Definition of destructor
template <class StackElement>
Stack<StackElement>::~Stack()
{
   delete[ ] myArrayPtr;
}
```

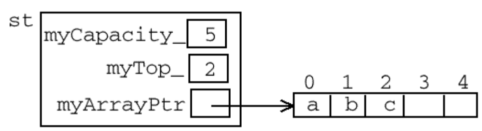
73

Purpose:

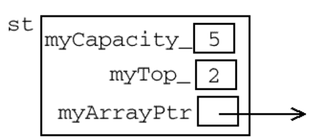
The compiler calls this function to destroy objects of type *ClassName* whenever such objects should no longer exist:

- At the end of the main function for class *ClassName* objects that are defined within *main( )* as static or global objects.
- At the end of each block in which a nonstatic *ClassName* object is defined.
- At the end of each function having a *ClassName* parameter.
- When a *ClassName* object allocated at run time is destroyed using *delete*.
- When an object containing a *ClassName* data member is destroyed.
- When an object whose type is derived from type *ClassName* is destroyed
- When a compiler-generated *ClassName* copy (made by the copy constructor) is no longer needed.

74



- When *st*'s lifetime is over



75

Copy constructor

Is needed whenever a copy of a class object must be built, which occurs:

- When a *class object is passed as a value* parameter
- When a *function returns a class object*
- If *temporary storage* of a class object is needed
- In *initializations of the form*

  Type obj = initial_value ;

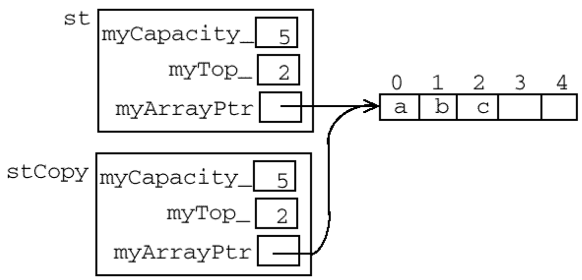
  or

  Type obj (initial_value) ;

the compiler must make a copy of *initial_value.*

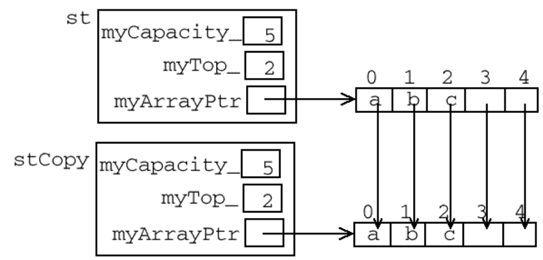
76

- *If a class has no copy constructor, the compiler uses a* ***default copy constructor*** *that does a byte-by-byte copy of the object.*



77

- That is, when a class contains a data member that is a pointer, the default copy constructor *does not make a distinct copy* of the object pointed to by that pointer
- What is needed is to create a distinct copy of *st*, in which the array in *stCopy* has exactly the same elements as the array in *st*:



78

Form of copy constructor:

- It is a constructor so it must be a member function, its name is the class name, and it has no return type.
- It needs a single parameter whose type is the class; this must be a *reference* parameter and should be *const*

79

```
/* --- Copy Constructor ---
* Precondition: A copy of a stack is needed
* Receive: The stack to be copied (as a const * reference parameter)
* Postcondition: A copy of original has been constructed.
************************************************************/
Stack(const Stack<StackElement> & original);
// end of class declaration
// Definition of copy constructor
template <class StackElement>
Stack<StackElement>::Stack(const Stack<StackElement> & original)
{
myCapacity_ = original.myCapacity_;              // copy myCapacity_ member
myTop_ = original.myTop_ ;                       // copy myTop_ member
myArrayPtr = new StackElement[myCapacity_];      // allocate array in copy
if (myArrayPtr == 0)                             // check if memory available
{
cout << "*** Inadequate memory to allocate stack ***\n";
exit(-1);
}       // or assert(myArrayPtr != 0 )
for (int pos = 0; pos < myCapacity_; pos++)      // copy array member
myArrayPtr[pos] = original.myArrayPtr[pos];
}
```

Assignment

- The default assignment operation does byte-by-byte copying.

  s2Copy = s2;
- This means that we must *overload the assignment operator* (*operator= ( )*) to create a distinct copy of the stack being assigned.

Form:

ClassName &ClassName : : operator=(const ClassName &original)
{
  // . . . make a copy of the original
  *return *this ;*
}

where

- *ClassName* is the name of the class containing this function; and *original* is a reference to the object being copied.
- *Note that the assignment operator must be a member function*.

assignment operator vs copy constructor - three main differences

First difference

- is that whereas the *copy constructor* builds and returns a *new* object, the *assignment operator* must assign this object *to an existing object* that already has a value. Usually, it must destroy the old value, deallocating its memory to avoid a memory leak, and then replace it with the new value.

The second difference

- is that the copy constructor need not to be concerned with self-assignments

  object = object ;
- But the assignment operator must. If it did not, destroying the old value of *object* on the left-hand side will also destroy the value of the *object* on the right-hand side, so there is nothing left to assign.

The third difference

- the *copy constructor* returns no value and thus has *no return* type, but the *assignment* operator should *return the object* on the left hand side of the assignment to support *chained assignments*. That is, an assignment like

  st3 = st2 = st1 ;
- will be processed as

  st3.operator= (st2.operator=(st1)) ;
- This can be accomplished by making the return type of *operator= ( )* a reference to type *Stack* and having it return **this*.

Prototype:

```
/* --- Assignment Operator ---
* Receive: Stack stRight (the right side of the assignment operator)
* object containing this member function
* Return (implicit parameter): The Stack containing this  function which will
   be a copy of stRight
* Return (function): A reference to the Stack containing  this function
********************************/
Stack<StackElement> & operator=(const Stack<StackElement> & original);
```

```
template <class StackElement>
Stack<StackElement> &Stack<StackElement>::operator=(const
   Stack<StackElement> & original)
{
   if (this != &original)              // check that not st = st
   {
   delete[] myArrayPtr;        // destroy previous array
   myCapacity_ = original.myCapacity_;
   // copy myCapacity_ member
   myTop_ = original.myTop_ ; // copy myTop_ member
   myArrayPtr = new StackElement[myCapacity_];
   // allocate array in copy
   if (myArrayPtr == 0)               // check if memory available
   {
        cerr << "* Inadequate memory to allocate stack *\n";
        exit(-1);
   }
   for (int pos = 0; pos < myCapacity_; pos++)
         // copy array member
        myArrayPtr[pos] = original.myArrayPtr[pos];
   }
   return *this;               // return reference to  this object
}
```

```
//***** Test Driver ************************
#include <iostream>
using namespace std;
#include "RTStack.h"
Print (Stack<int> st)
{
st.display(cout) ;
}
int main( )
{
int Size;
cout << "Enter stack size: ";
cin >> Size;
Stack<int> S(Size);
for (int i = 1; i <= 5; i++)
S.push(i)
Stack<int> T = S;
cout << T << endl;             //   Print(T) ;
}
```

86

Sample Runs:

Enter stack capacity: 5
5
4
3
2
1

Enter stack capacity: 3
*** Stack is full -- can't add new value ***
*** Stack is full -- can't add new value ***
3
2
1

Enter stack capacity: 0
StackRT.cc:12: failed assertion `NumElements > 0'
Abort

87

Test driver with statements in the constructor, copy constructor, and destructor to trace when they are invoked

```
//***** Test Driver *****
#include <iostream>
using namespace std;

#include "StackRTemp1"

Print (Stack<int> st)
{
st.display(cout) ;
}
```

```
Enter stack capacity: 5
**A**
CONSTRUCTOR
**B**
**C**
**C**
**C**
**C**
**C**
**D**
COPY CONSTRUCTOR
```

88

```
int main( )
{
int numElements;
cout << "Enter stack capacity: ";
cin >> numElements;

cout << "**A**\n";
Stack<int> s(numElements);
cout << "**B**\n";
for (int i = 1; i <= 5; i++)
{
cout << "**C**\n";
s.push(i);
}
cout << "**D**\n";
Stack<int> t = s;
cout << "**E**\n";
Print(t);
cout << "**F**\n";
Stack<int> u;
cout << "**G**\n";
u = t;

cout << "**H**\n";
Print(u);
cout << "**I**\n";
}
```

```
**E**
COPY CONSTRUCTOR
5
4
3
2
1
DESTRUCTOR
**F**
CONSTRUCTOR
**G**
ASSIGNMENT OPERATOR
**H**
COPY CONSTRUCTOR
5
4
3
2
1
DESTRUCTOR
**I**

DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
```

Note:

The following **is a general rule** to remember when designing a class:

- If a *class allocates memory* at a run time using *new*, then it should provide
  - A *copy constructor* that the compiler can use to make distinct copies
  - An *assignment operator* that a programmer can use to make distinct copies
  - A *destructor* that releases the run-time allocated memory to the heap
  - The class **constructors, destructor**, and **a copy constructor** are called automatically by the compiler when a class is used.

Note:

1. The ***assert*** macro tests a condition for correctness, and terminates the program if the test fails. It is used for diagnostics.

   void assert(bool expression) ;

2. *<cstdlib>* contains function prototypes for *conversions of numbers to text and text to numbers, for memory allocation, random numbers and other utility functions.*

   void exit(int status) ;

- Terminates the program execution and returns control to the operating system if the status is different than 0. *Status = 0* signals successful termination and any *nonzero value* signals unsuccessful termination.