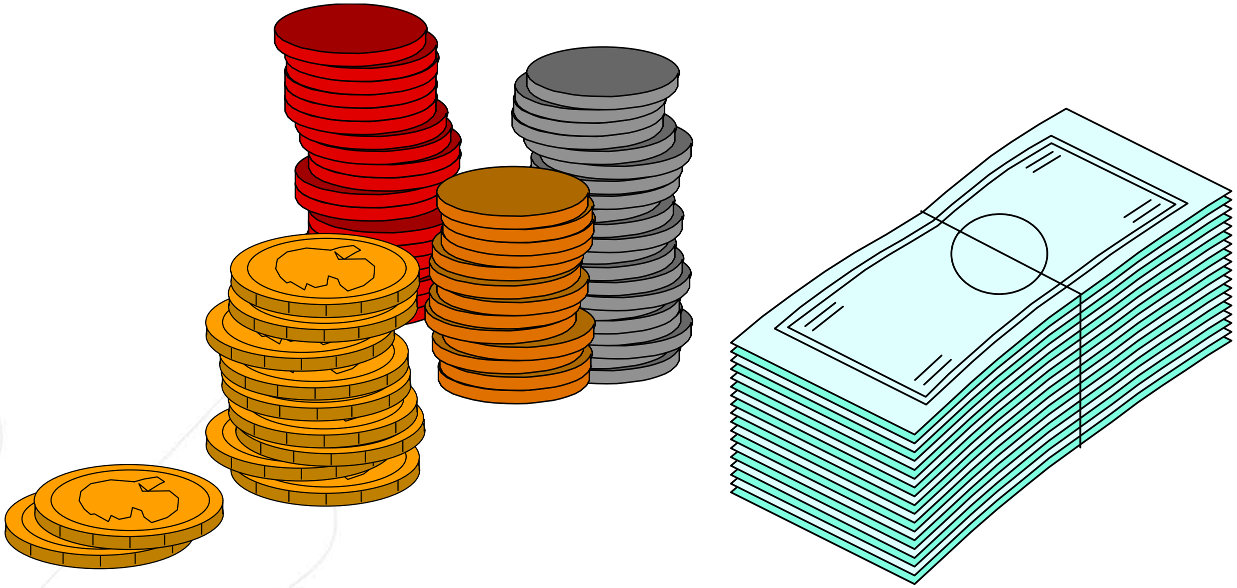
CSE 2017 Data Structures and Lab

Lecture #4: Stack

Eun Man Choi

Stacks of Coins and Bills



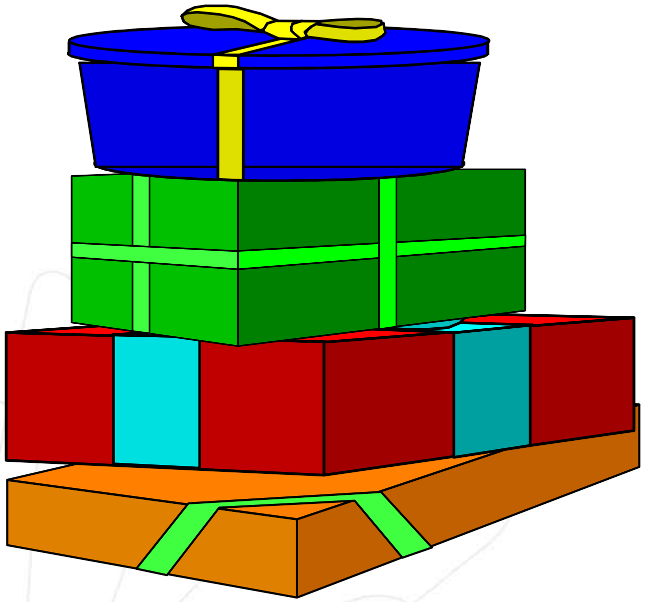
Stacks

What is a stack?

- *Logical (or ADT) level:* **A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.**
- **A stack is a LIFO "last in, first out" structure.**

Stacks of Boxes and Books

TOP OF THE STACK



TOP OF THE STACK

Stack ADT Operations

- MakeEmpty -- Sets stack to an empty state.
- IsEmpty -- Determines whether the stack is currently empty.
- IsFull -- Determines whether the stack is currently full.
- Push (ItemType newItem) -- Adds newItem to the top of the stack.
- Pop (ItemType& item) -- Removes the item at the top of the stack and returns it in item.

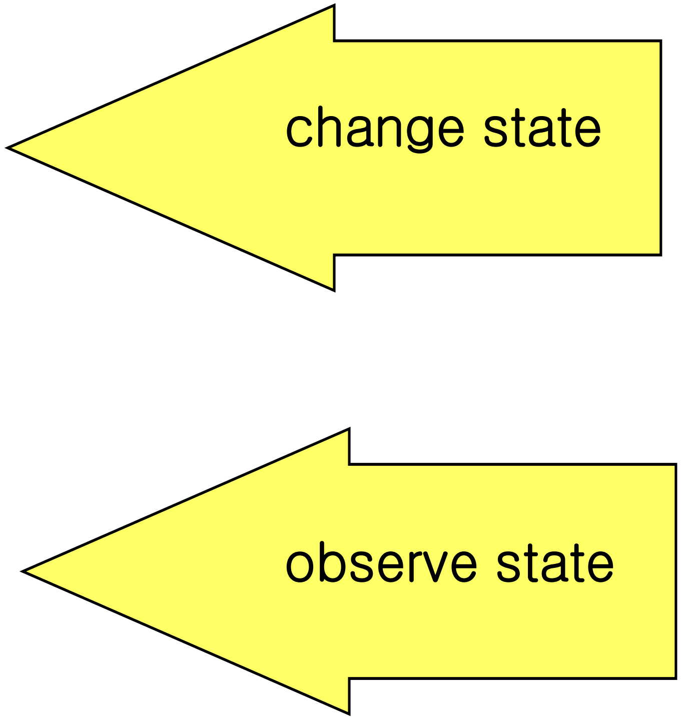
ADT Stack Operations

Transformers

- MakeEmpty
- Push
- Pop

Observers

- IsEmpty
- IsFull



change state

observe state

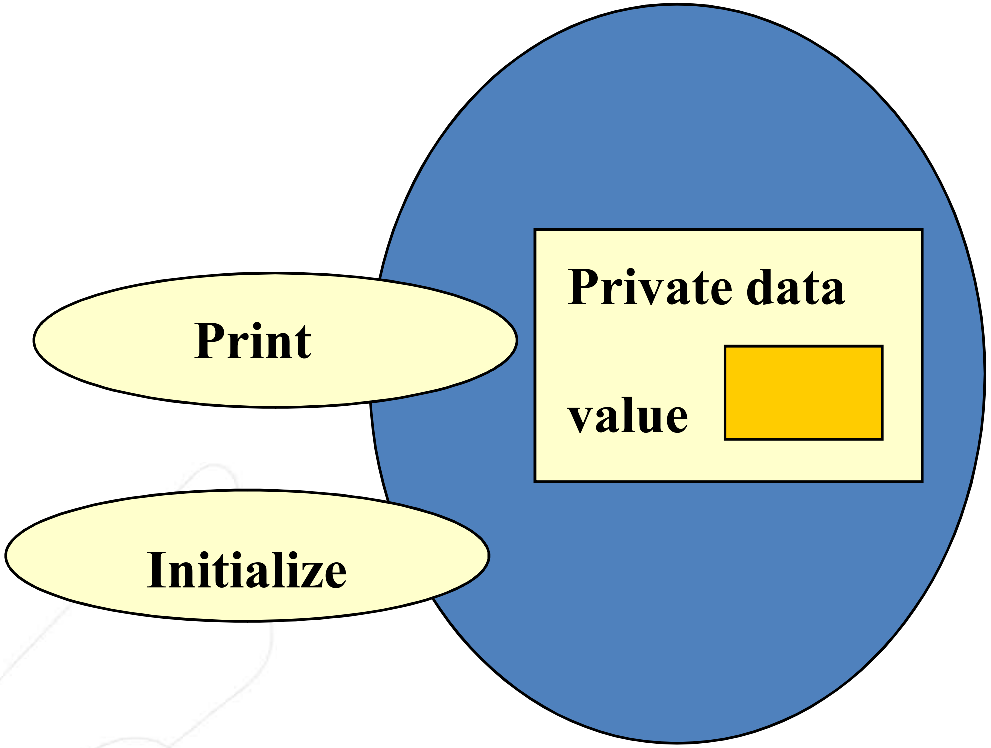
```
//----------------------------------------------------------
// SPECIFICATION FILE (stack.h)
//----------------------------------------------------------
#include "bool.h"
#include "ItemType.h"          // for MAX_ITEMS and
                               // class ItemType definition

class StackType  {
public:
  StackType( );
        // Default constructor.
        // POST:  Stack is created and empty.

  void MakeEmpty( );
        // PRE:   None.
        // POST:  Stack is empty.

  bool IsEmpty( ) const;
        // PRE:   Stack has been initialized.
        // POST:  Function value = (stack is empty)
```

// SPECIFICATION FILE continued (Stack.h)

```
  bool IsFull( ) const;
       // PRE:    Stack has been initialized.
       // POST:   Function value = (stack is full)

  void Push( ItemType newItem );
       // PRE:   Stack has been initialized and is not full.
       // POST: newItem is at the top of the stack.


  void Pop( ItemType&  item );
       // PRE:   Stack has been initialized and is not empty.
       // POST: Top element has been removed from stack.
       //       item is a copy of removed element.

private:
  int       top;
  ItemType  items[MAX_ITEMS];     // array of ItemType
};
```

ItemType Class/Struct Interface Diagram

class ItemType



```
//-------------------------------------------------------
// IMPLEMENTATION FILE (Stack.cpp)
//------------------------------------------------------
// Private data members of class:
//           int      top;
//           ItemType items[MAX_ITEMS];
//-------------------------------------------------------
#include “bool.h”
#include “ItemType.h”

StackType::StackType( )
       //------------------------------------------------
       // Default Constructor
       //------------------------------------------------
{
  top = -1;
}
```

```
// IMPLEMENTATION FILE continued    (Stack.cpp)
//--------------------------------------------------------

void StackType::MakeEmpty( )
        //---------------------------------------------------
        // PRE:    None.
        // POST:   Stack is empty.
        //---------------------------------------------------
{
  top = -1;
}
```

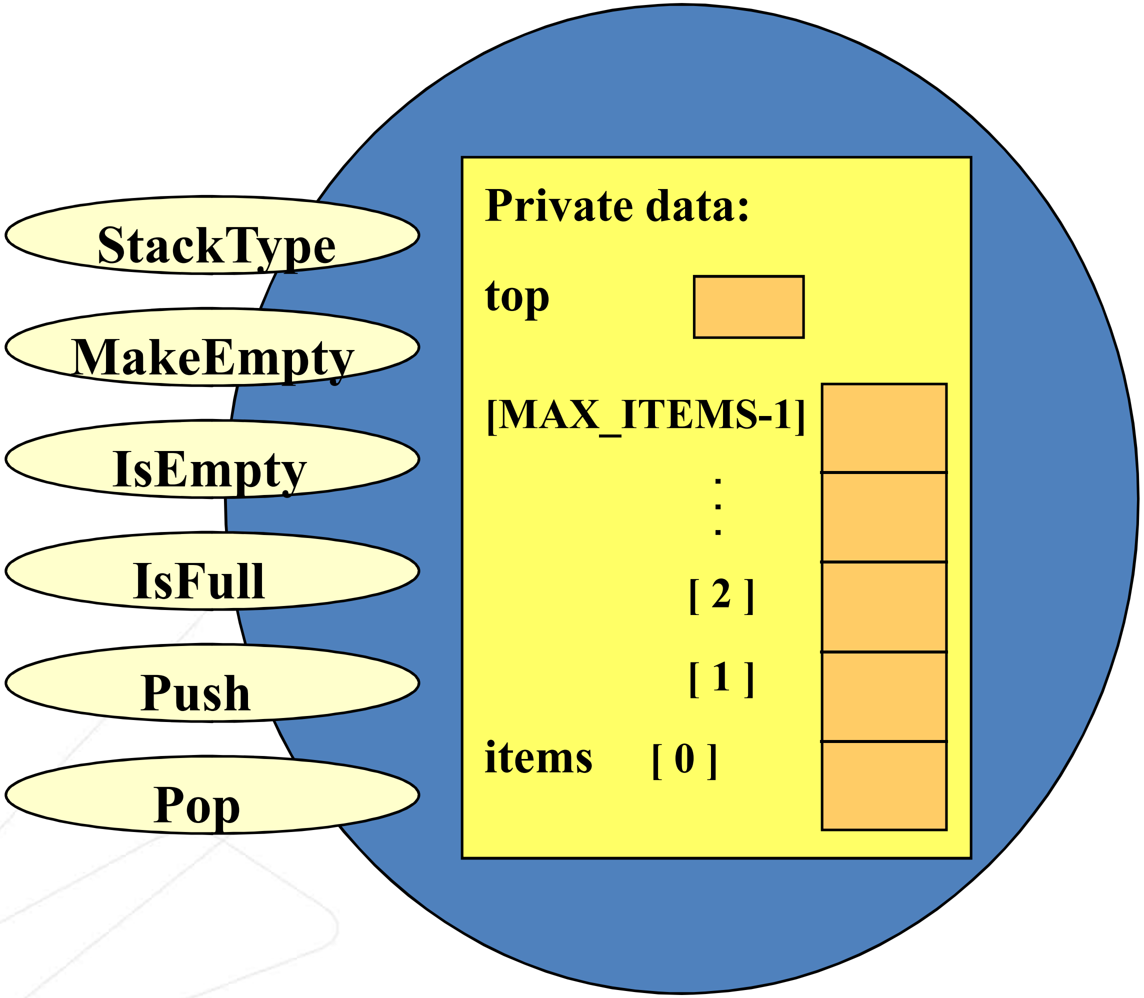
```
// IMPLEMENTATION FILE continued    (Stack.cpp)
//----------------------------------------------------------------------

bool StackType::IsEmpty( ) const
        //------------------------------------------------------------
        // PRE:    Stack has been initialized.
        // POST:   Function value = (stack is empty)
        //------------------------------------------------------------
{
  return ( top == -1 );
}

bool StackType::IsFull( ) const
        //------------------------------------------------------------
        // PRE:    Stack has been initialized.
        // POST:   Function value = (stack is full)
        //------------------------------------------------------------
{
  return  ( top == MAX_ITEMS-1 );
}
```

```
// IMPLEMENTATION FILE continued    (Stack.cpp)
//--------------------------------------------------------------

void StackType::Push ( ItemType newItem )
       //------------------------------------------------------
       // PRE:  Stack has been initialized and is not full.
       // POST: newItem is at the top of the stack.
       //------------------------------------------------------
{
  top++;
  items[top] = newItem;
}
```

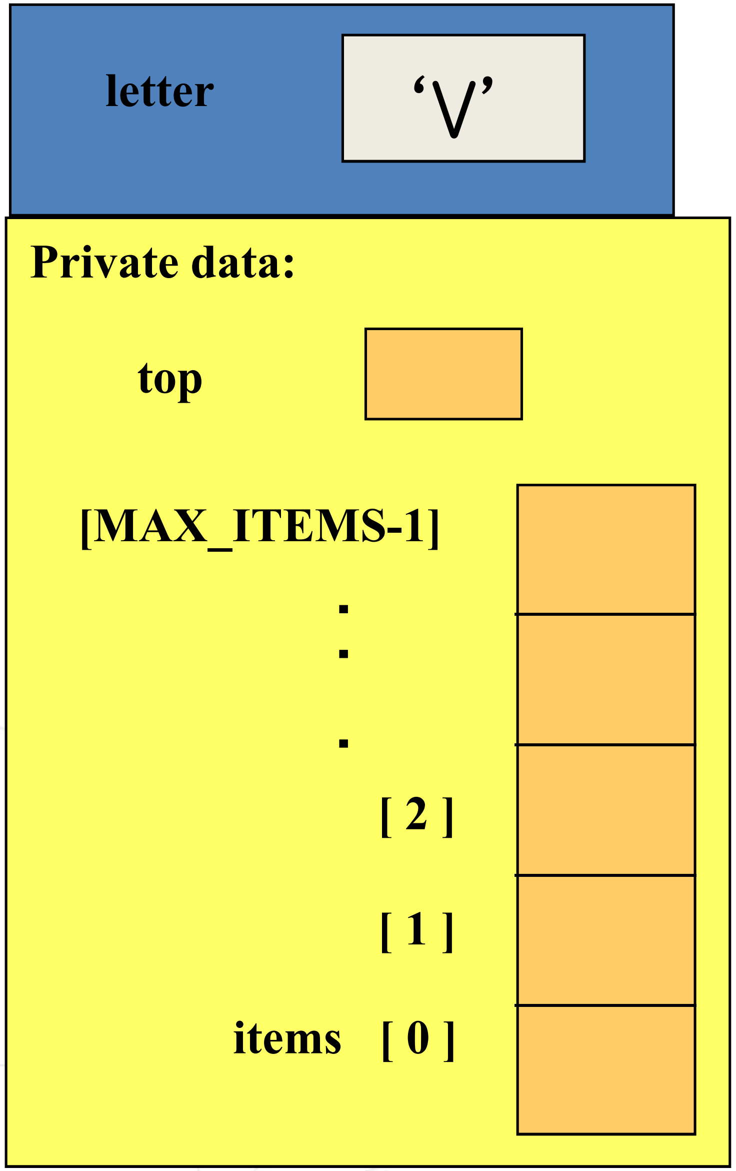
```
// IMPLEMENTATION FILE continued    (Stack.cpp)
//------------------------------------------------------------


void StackType::Pop ( ItemType&  item )
       //-----------------------------------------------------
       // PRE:  Stack has been initialized and is not empty.
       // POST: Top element has been removed from stack.
       //       item is a copy of removed element.
       //-----------------------------------------------------
{
  item = items[top];
  top--;
}
```

Class Interface Diagram

StackType class



Tracing Client Code



letter ‘V’

Private data:

top

[MAX_ITEMS-1]

.
.
.

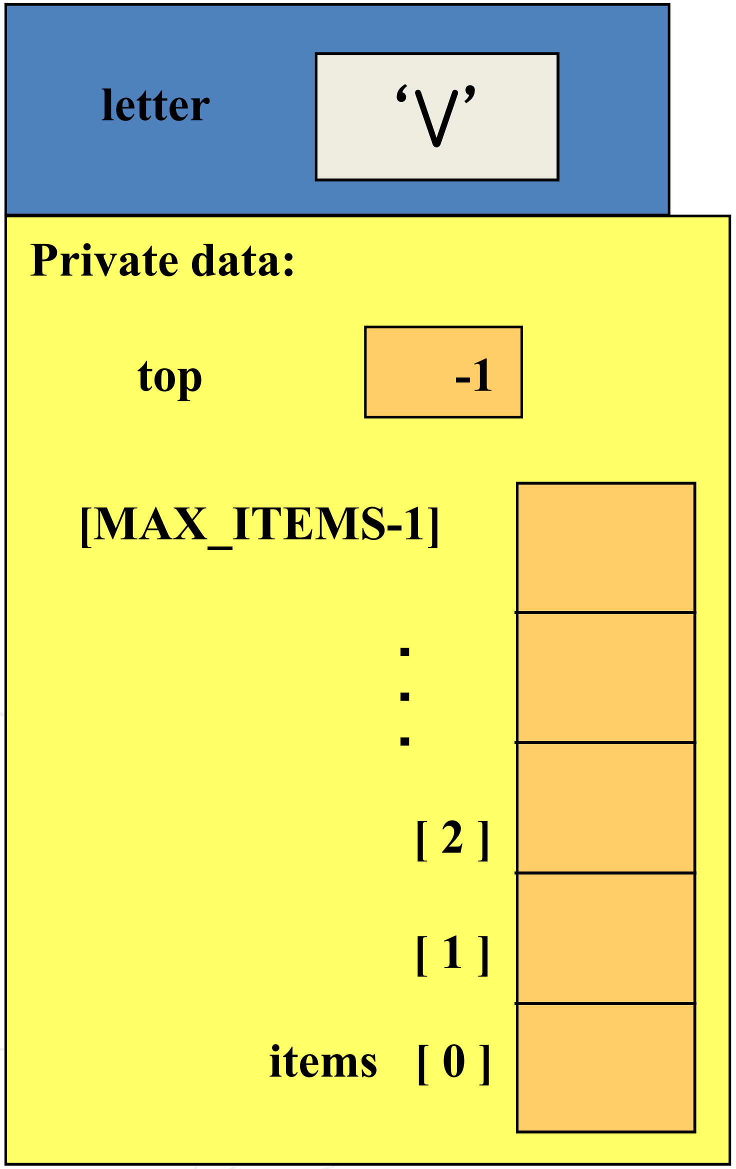
[2]

[1]

items [0]

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
      charStack.Pop(letter);
```

Tracing Client Code



letter ‘V’

Private data:

top -1

[MAX_ITEMS-1]

.
.
.

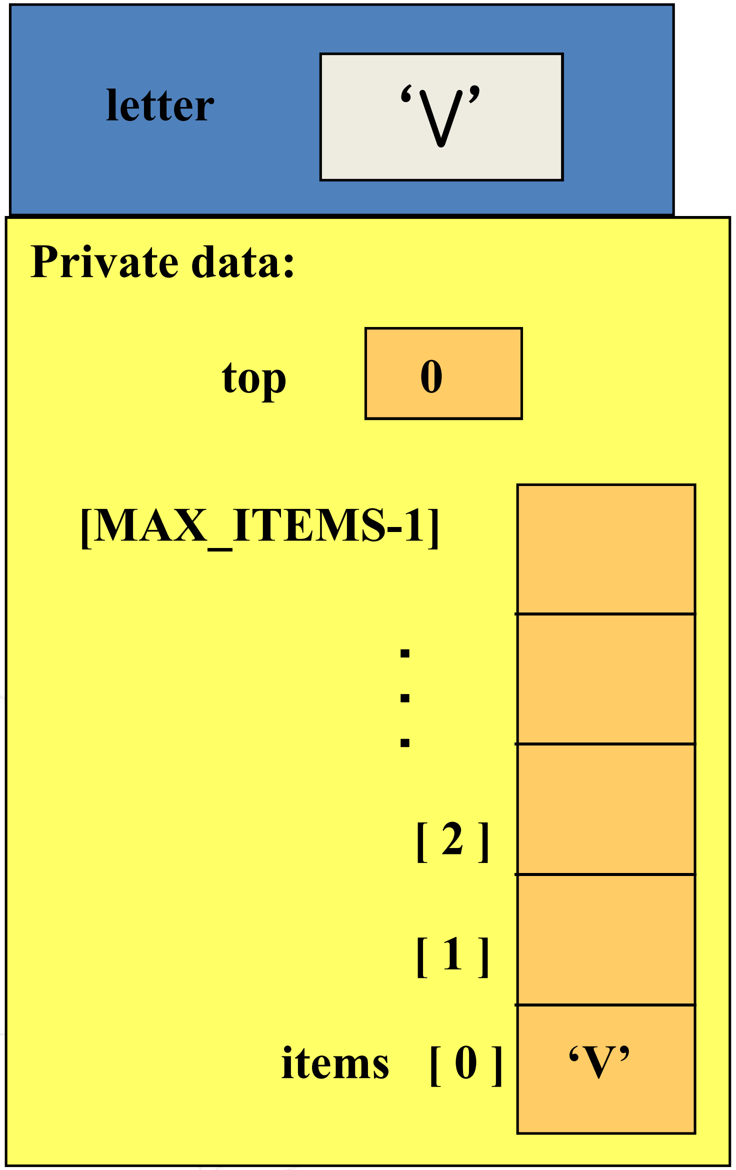
[2]

[1]

items [0]

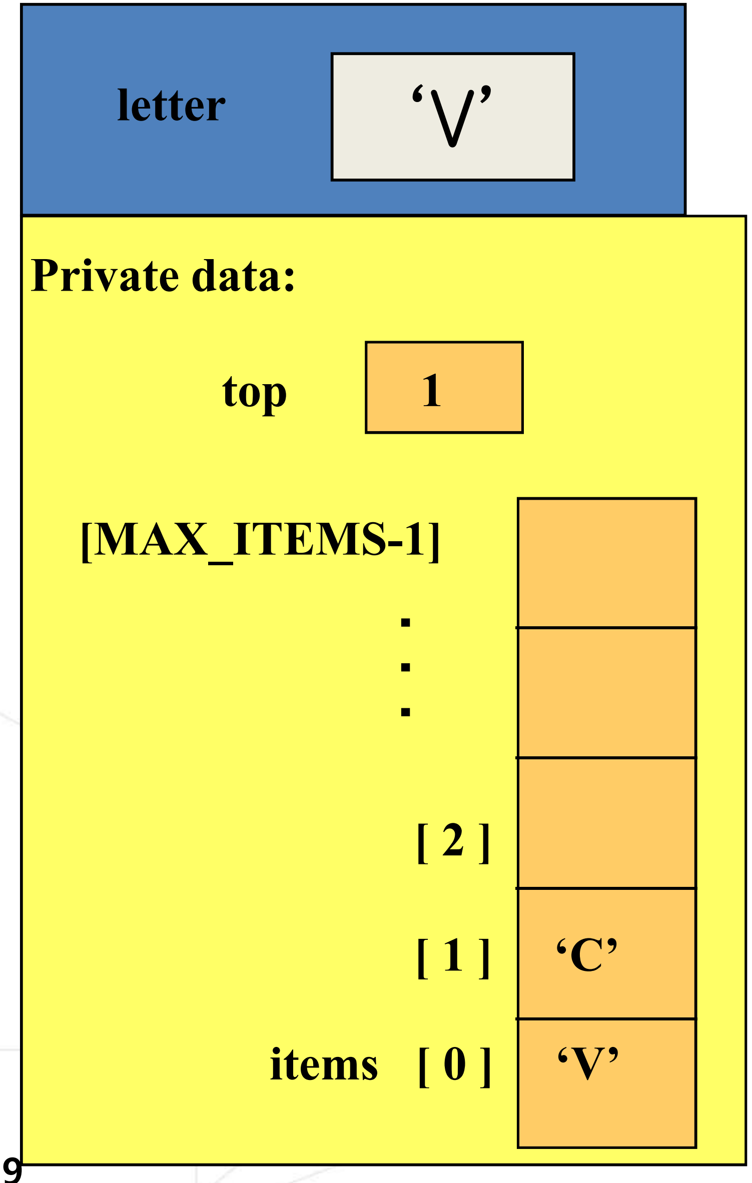
```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push('K');
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘V’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

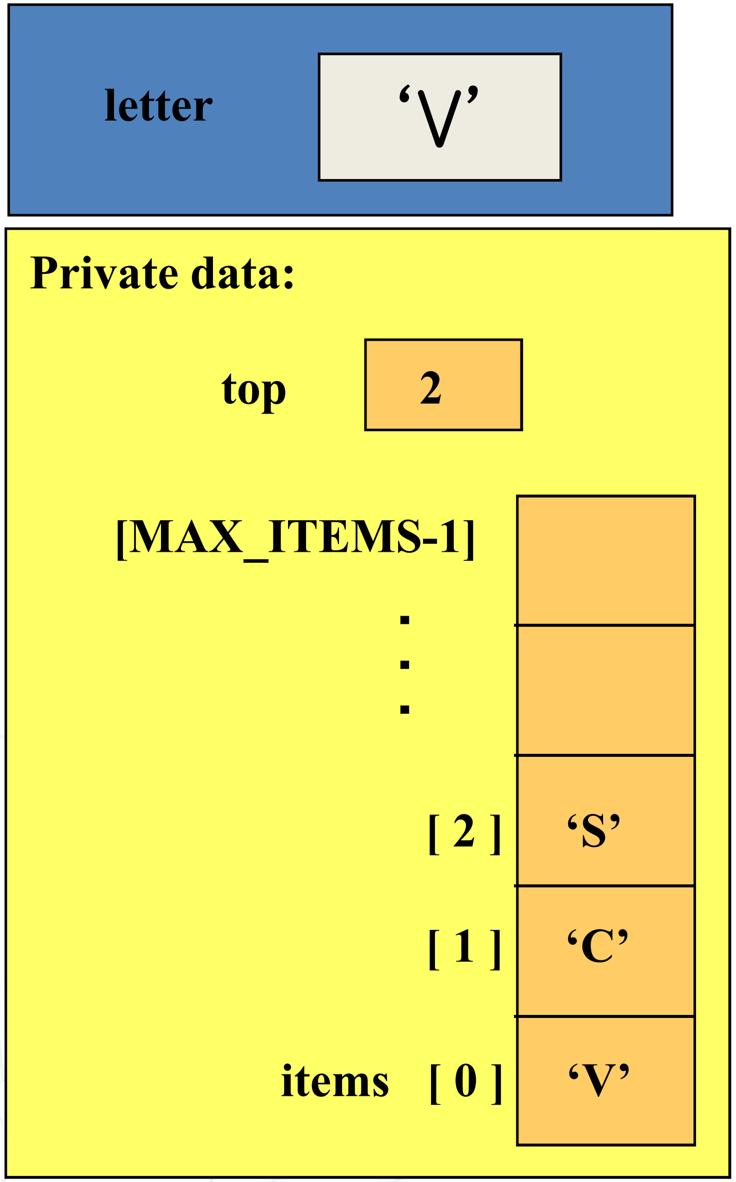
[2]

[1] ‘C’

items [0] ‘V’

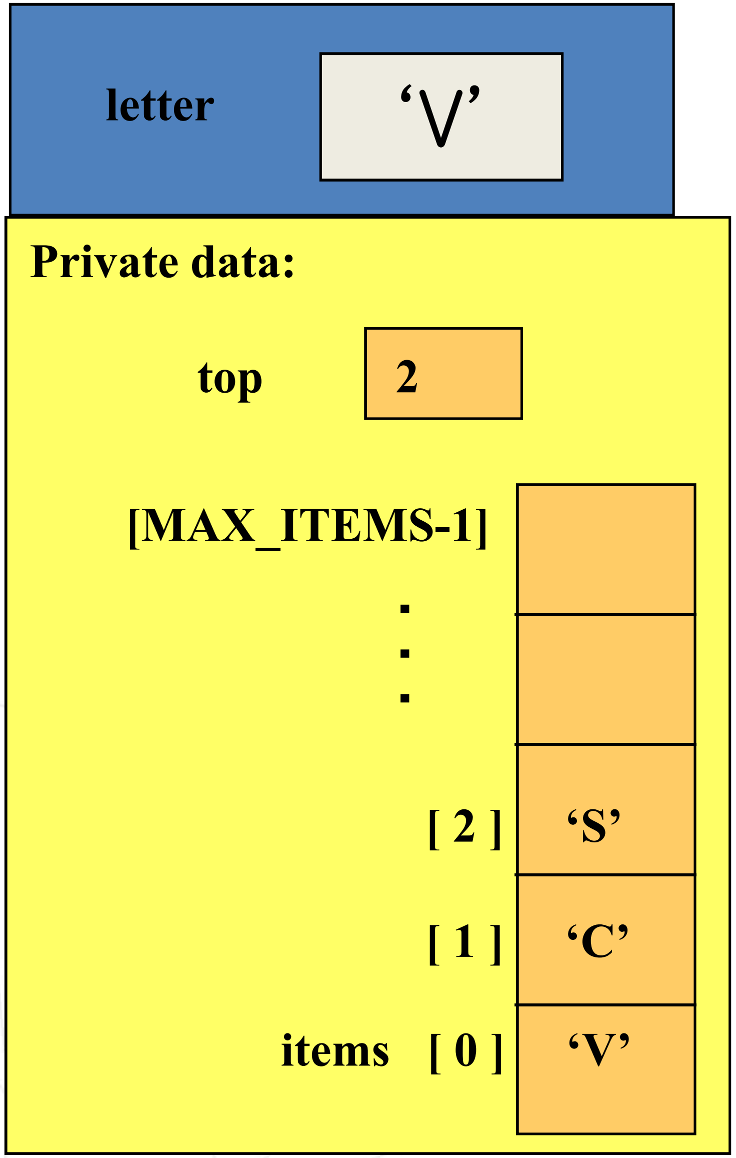
```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
       charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
       charStack.Pop(letter);
```

Tracing Client Code



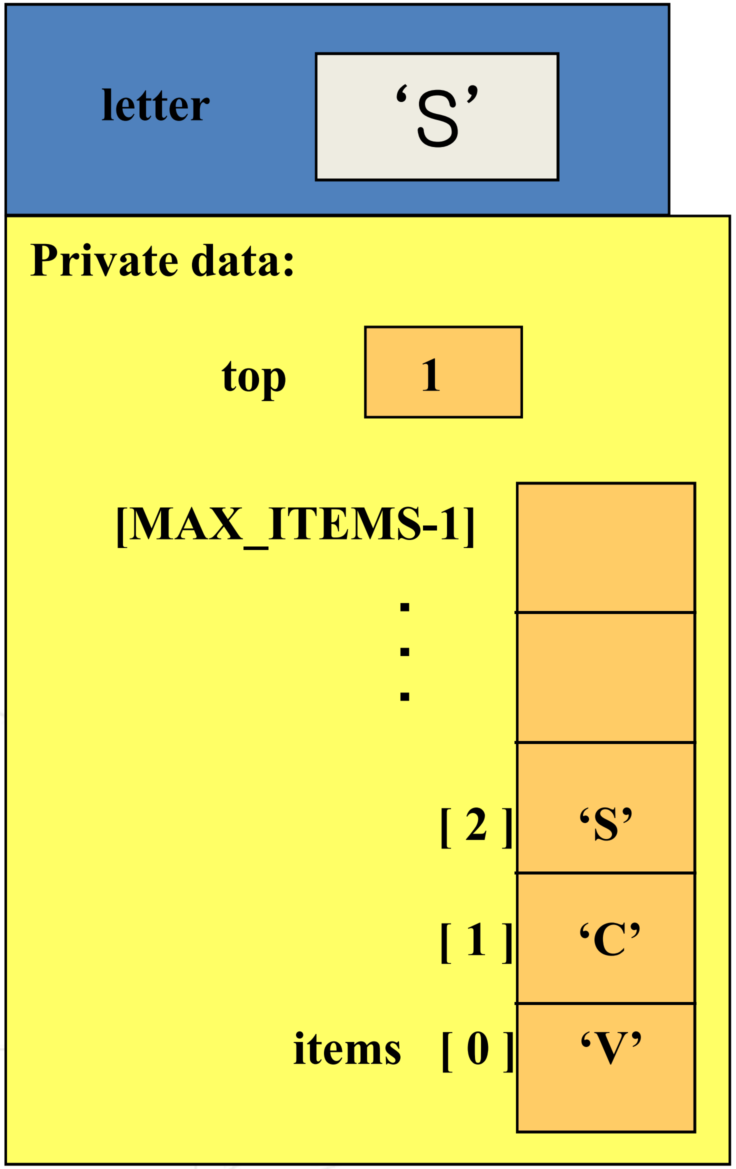
```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push('K');
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



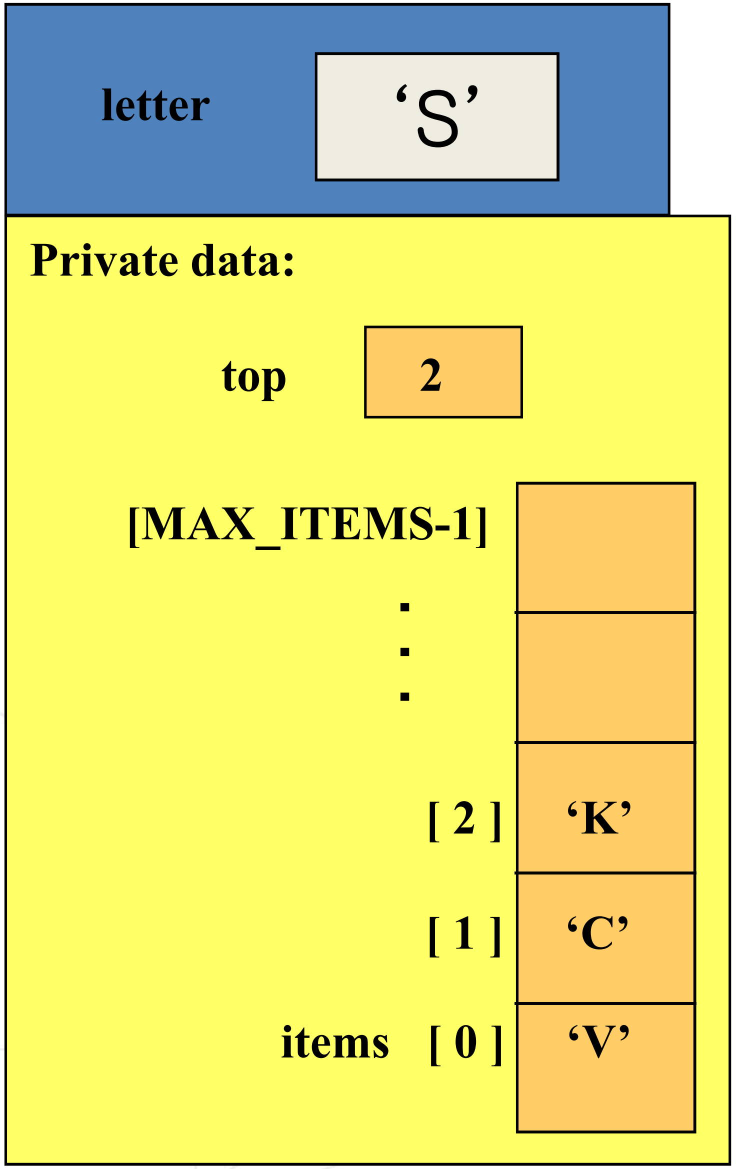
```
char   letter = 'V';
StackType   charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push('K');
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



```
char   letter = 'V';
StackType   charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push('K');
while  (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘S’

Private data:

top 2

[MAX_ITEMS-1]

. . .

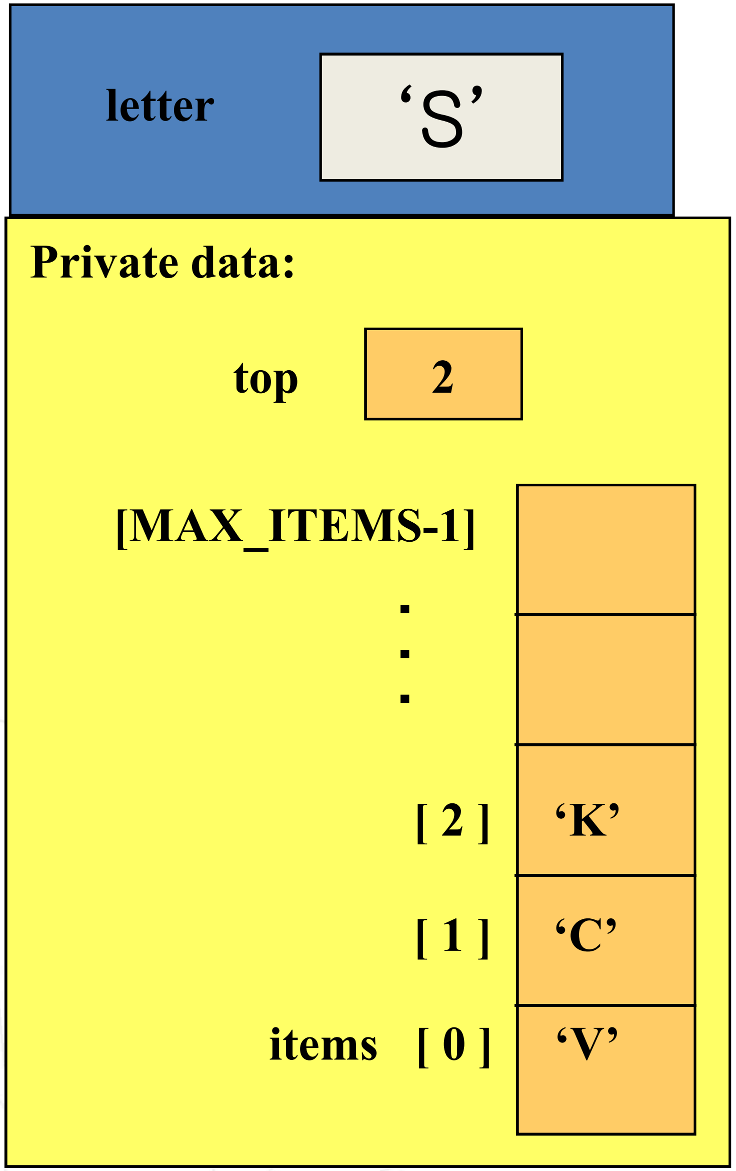
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘S’

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

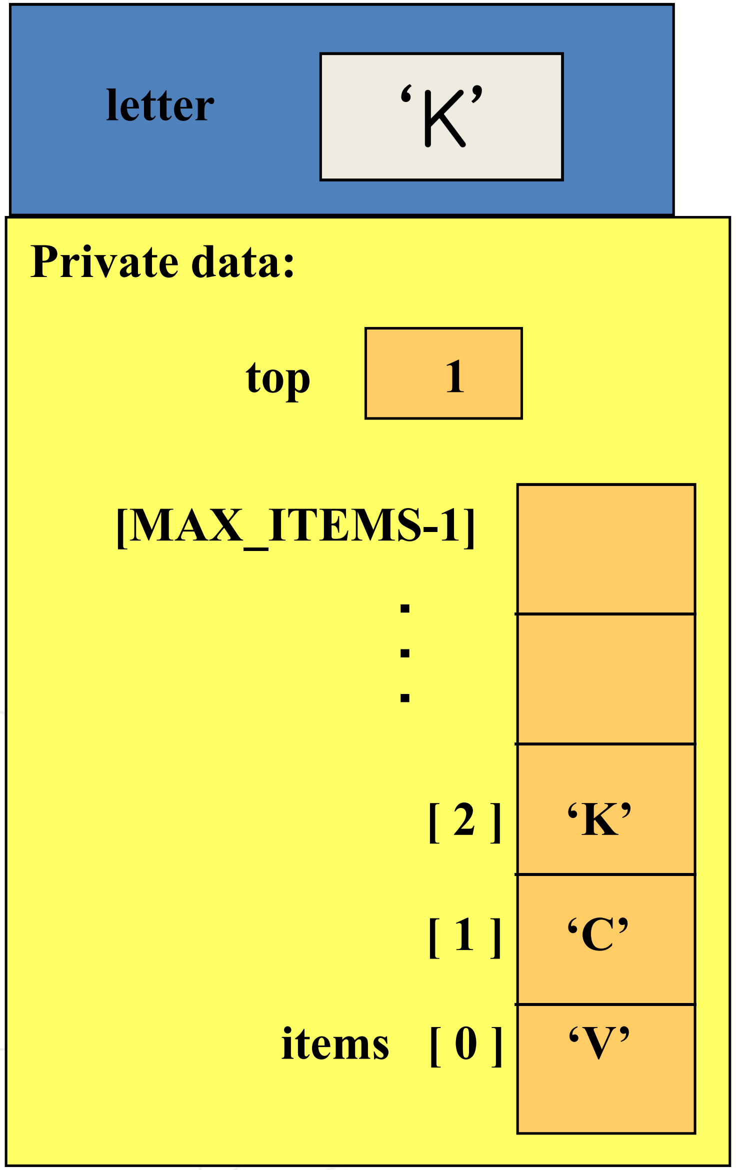
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

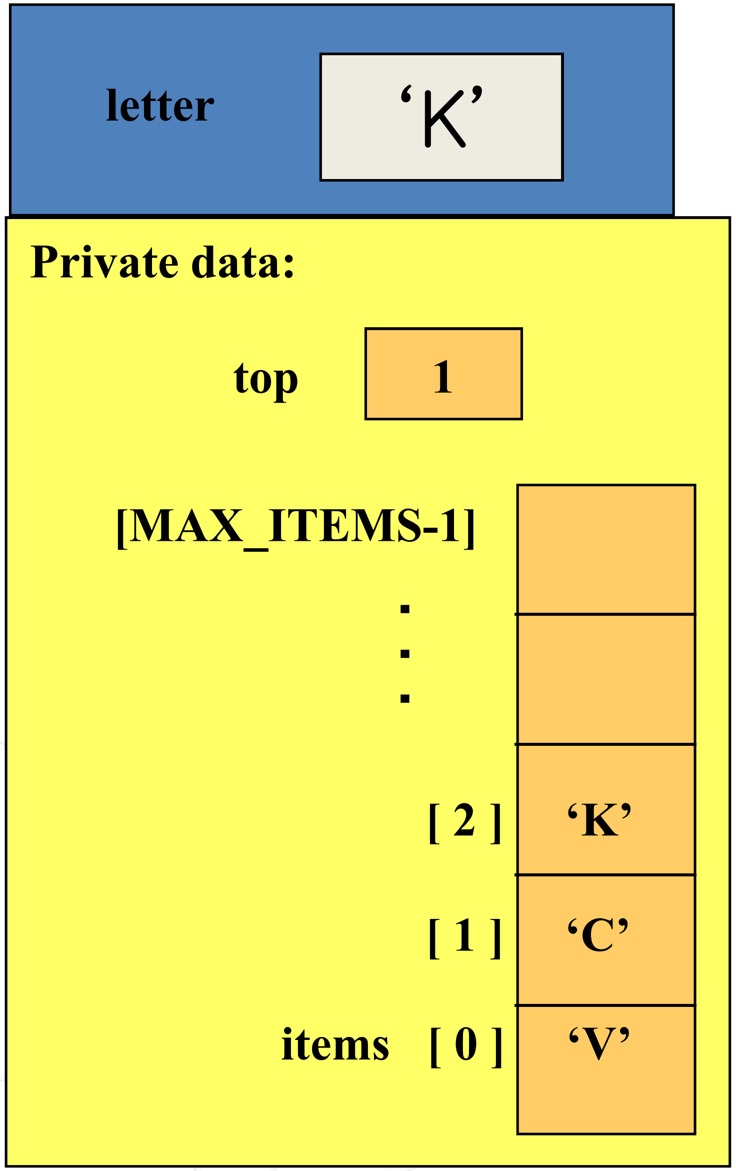
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

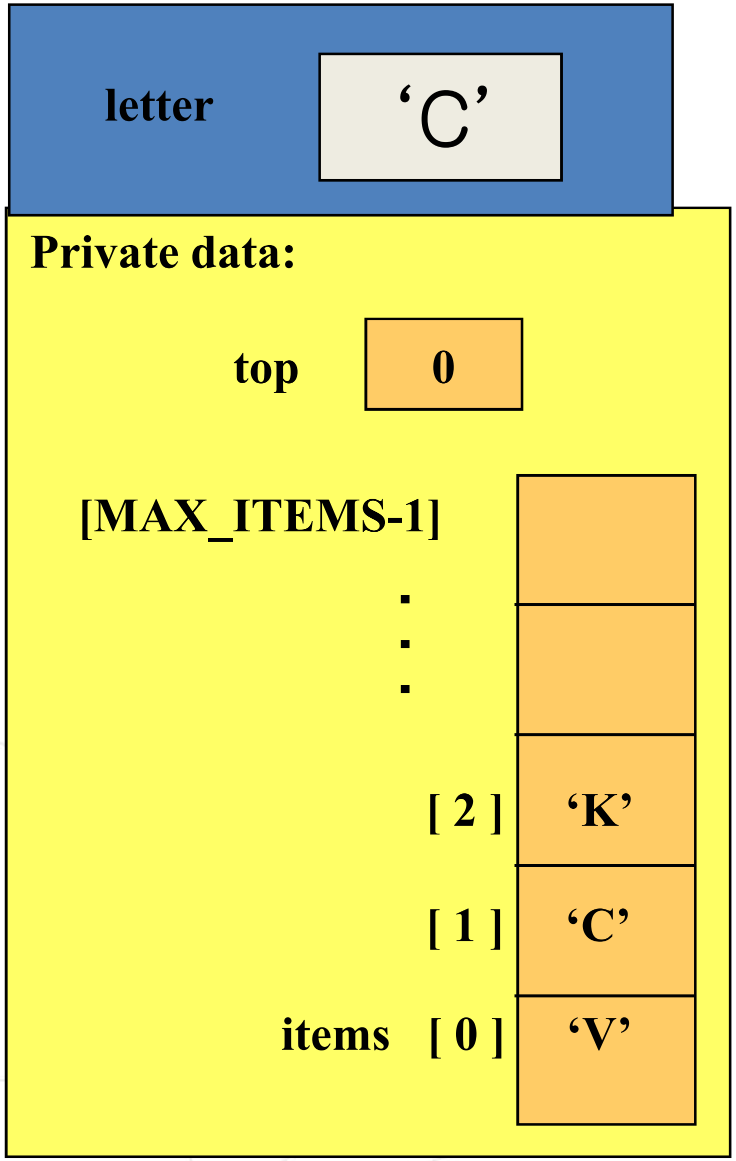
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



letter ‘C’

Private data:

top 0

[MAX_ITEMS-1]

.
.
.

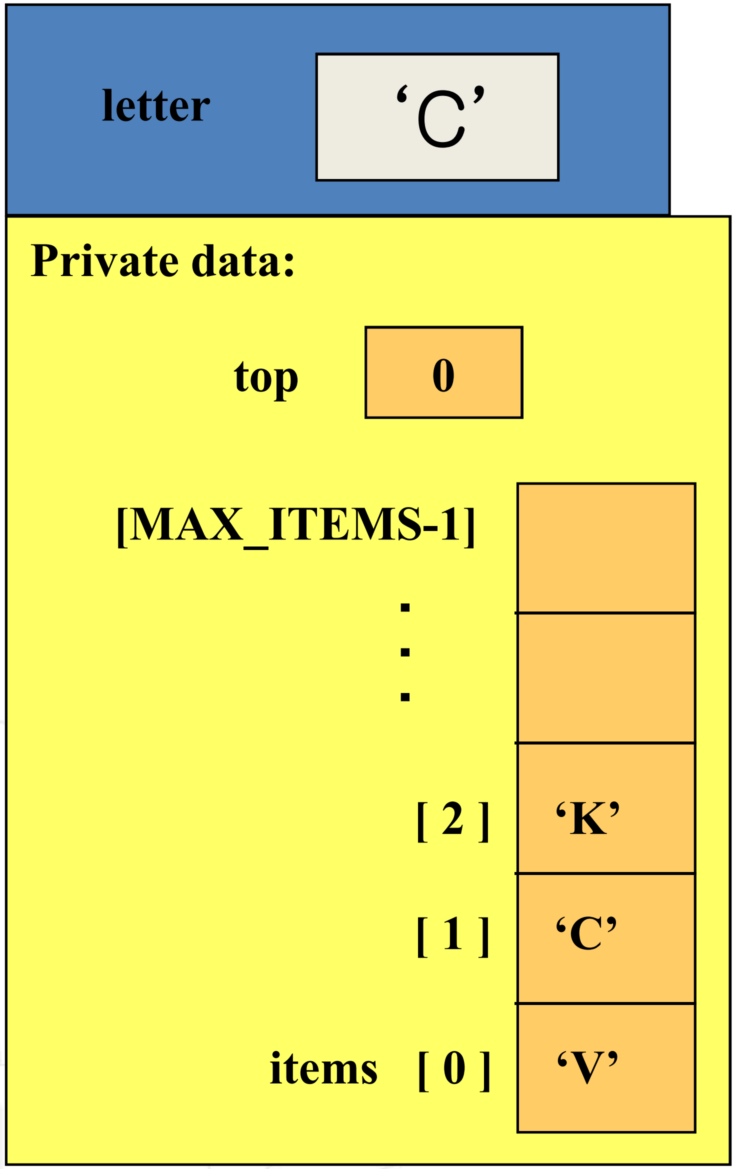
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
       charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
       charStack.Pop(letter);
```

Tracing Client Code



letter ‘C’

Private data:

top 0

[MAX_ITEMS-1]

.
.
.

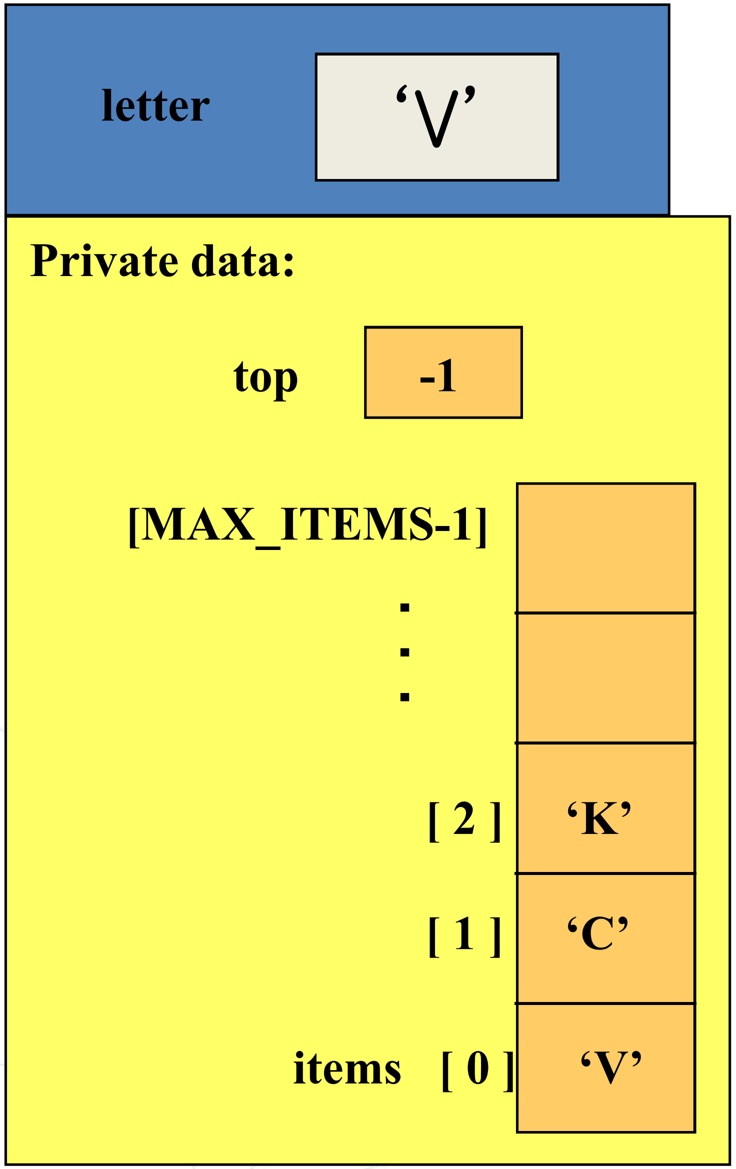
[2] ‘K’

[1] ‘C’

items [0] ‘V’

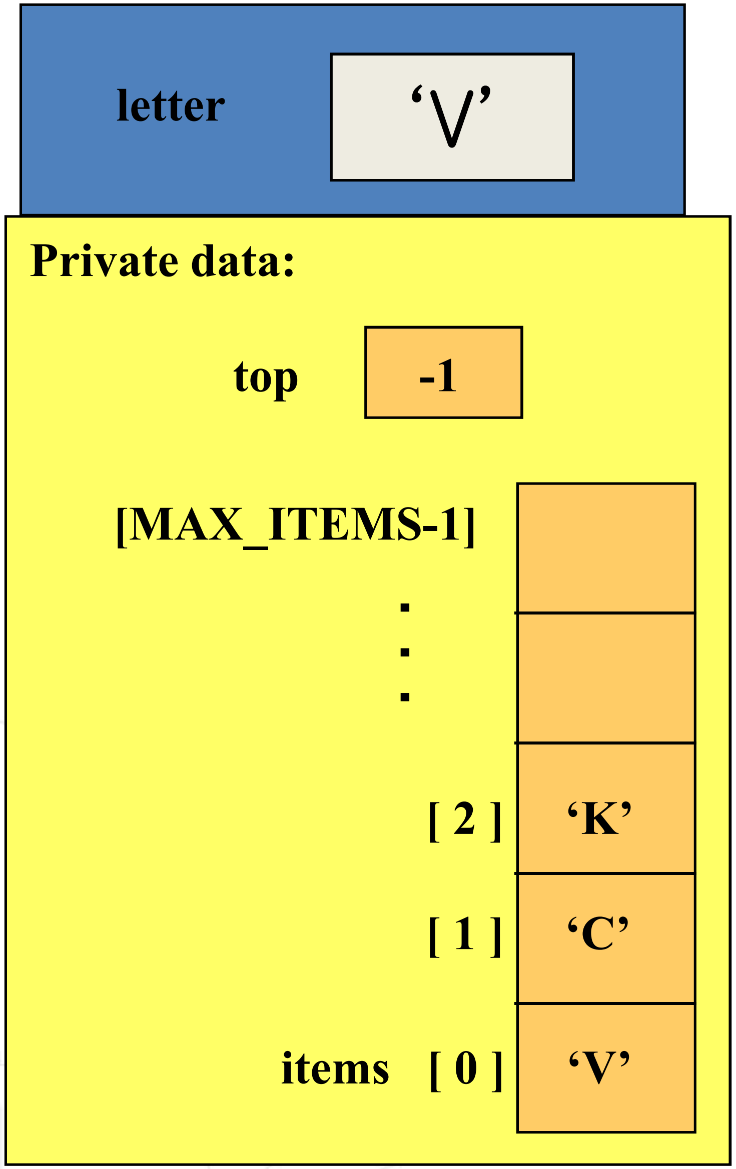
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

Tracing Client Code



```
char   letter = 'V';
StackType   charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push('K');
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

End of Trace



letter ‘V’

Private data:

top -1

[MAX_ITEMS-1]	
. . .	
[2]	‘K’
[1]	‘C’
items [0]	‘V’

```
char   letter = ‘V’;
StackType   charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
        charStack.Pop(letter);
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
        charStack.Pop(letter);
```

What is a Class Template?

- A class template allows the compiler to generate multiple versions of a class type by using type parameters.

- The formal parameter appears in the class template definition, and the actual parameter appears in the client code. Both are enclosed in pointed brackets, < >.

```
//  Laboratory 4, Class declaration...  listarr.h
...
template < class LE >
class List {
  public:
    List ( int maxNumber = defMaxListSize );
   ~List ();
   void insert ( const LE &newElement );
   void remove ();
  ...
   LE getCursor () const;   // Return element
  ..
    void moveToNth ( int n );                   // InLab 2
   int find ( const LE &searchElement );       // InLab 3
  private:
    // Data members
    int maxSize, size, cursor;
    LE *element;   // Array containing the list elements
};
```

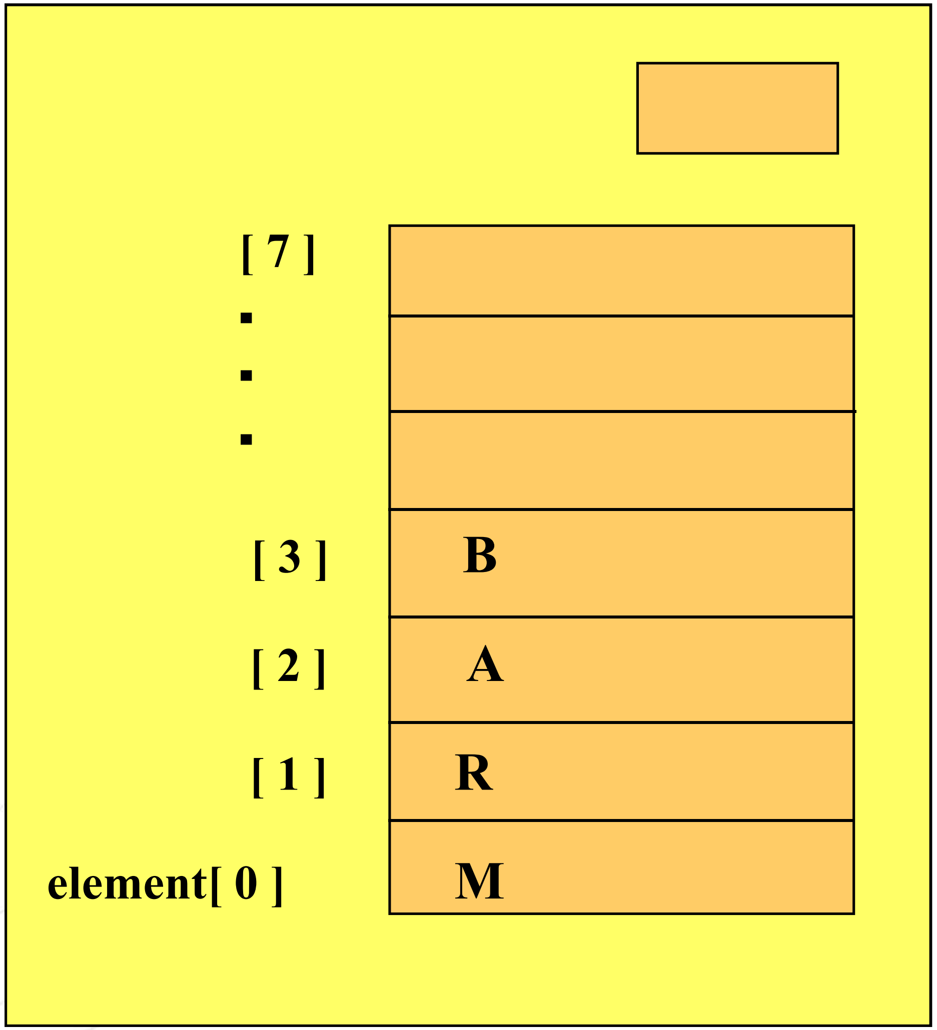
```
//  Laboratory 4, Class implementation ... listarr.cpp
#include <assert.t>
#include "listarr.h"
//--------------------------------------------
template < class LE >
List<LE> :: List ( int maxNumber )
// Creates an empty list. Allocates enough memory for
  maxNumber
// elements (defaults to defMaxListSize).
   : maxSize(maxNumber),size(0),cursor(-1)
{
    element = new LE [ maxSize ];
    assert ( element != 0 );
}
//-----------------------------------------------
template < class LE >
List<LE> :: ~List () // Frees the memory used by a list.
{
    delete [] element;
}
//-----------------------------------------------
  ...
```

//Laboratory 4, Client code: test4.cpp (modified)

```
#include <iostream.h>
#include "listarr.cpp"

void main()
{
   List<char> testList_char(8);   // Test list
  List<int> testList_int(10);   // Test list
   char testElement_char;        // List element
   int  testElement_int;         // List element
    ...
}
```

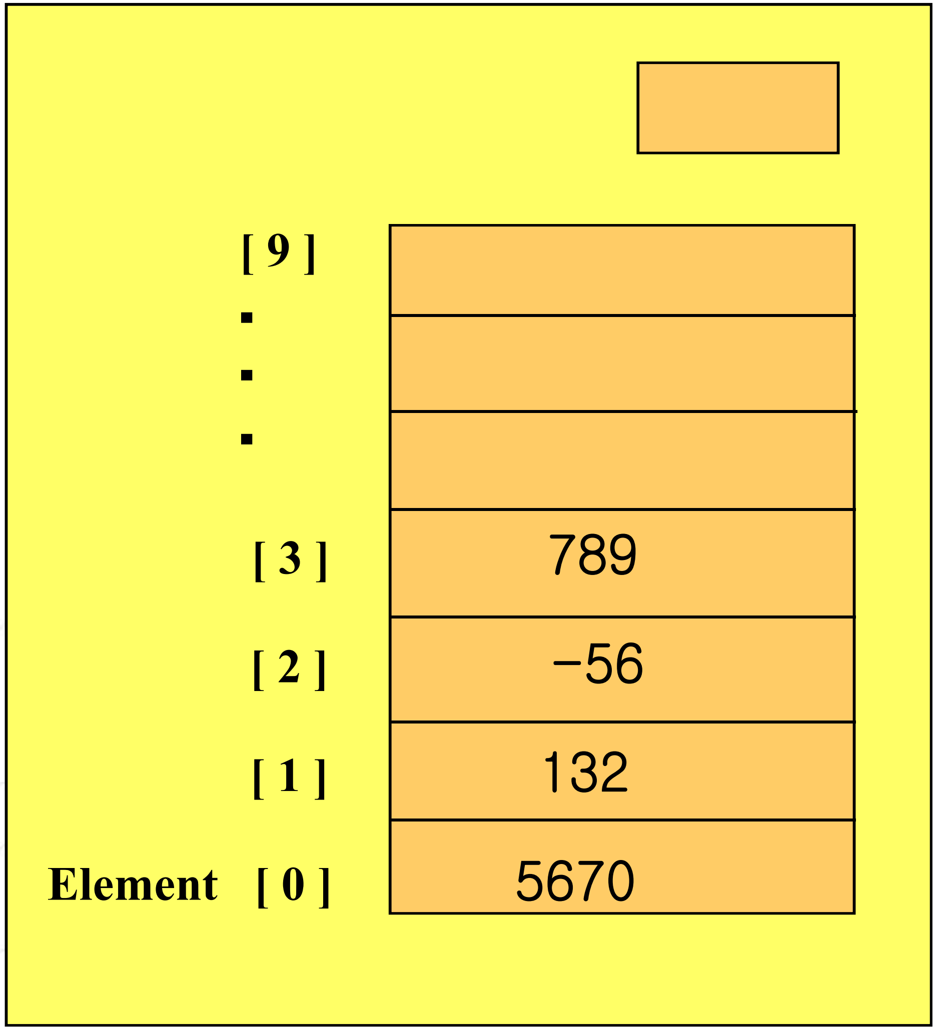
List<char> testList_char(8);

ACTUAL PARAMETER



List<int>testList_int(10);

ACTUAL PARAMETER



[9]

▪

▪

▪

[3] 789

[2] -56

[1] 132

Element [0] 5670

Using class templates

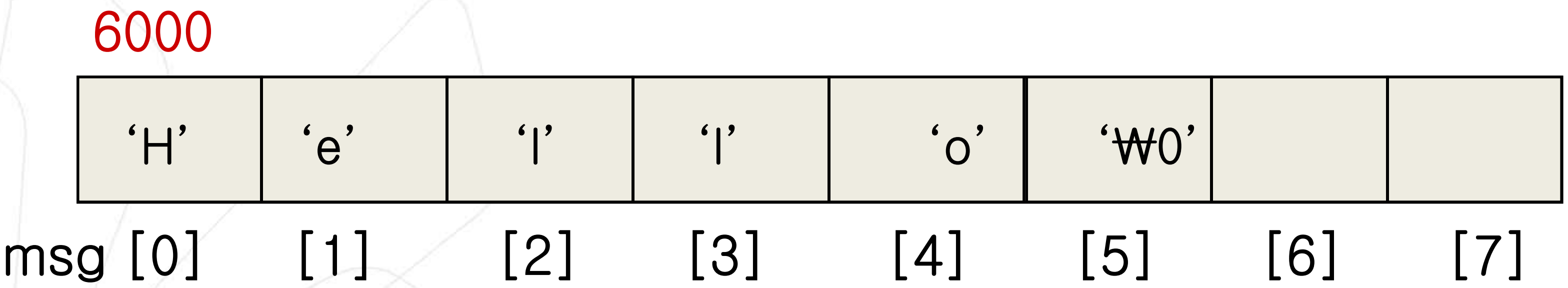
- The actual parameter to the template is a data type. Any type can be used, either built-in or user-defined.

Pointer Types

Recall that …

char msg [8];

msg is the base address of the array. We say msg is a pointer because its value is an address. It is a pointer constant because the value of msg itself cannot be changed by assignment. It “points” to the memory location of a `char`.



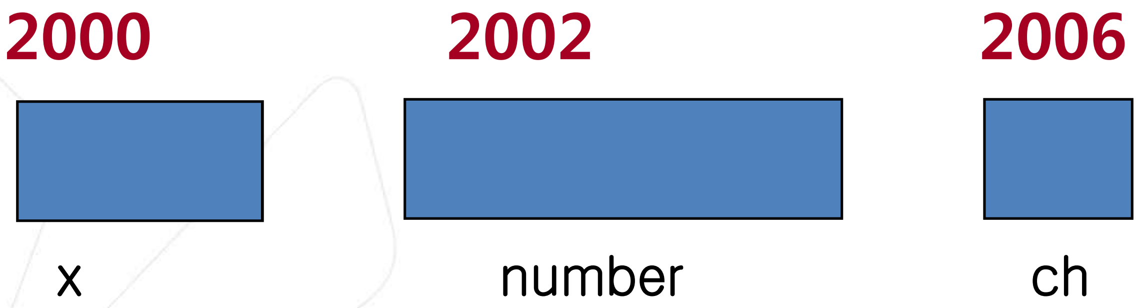
6000

‘H’	‘e’	‘l’	‘l’	‘o’	‘₩0’		
msg [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable.

```
int      x;
float    number;
char     ch;
```



Obtaining Memory Addresses

- The address of a non-array variable can be obtained by using the address-of operator &.

```
int      x;
float    number;
char     ch;

cout << “Address of x is “ << &x << endl;

cout << “Address of number is “ << &number << endl;

cout << “Address of ch is “ << &ch << endl;
```

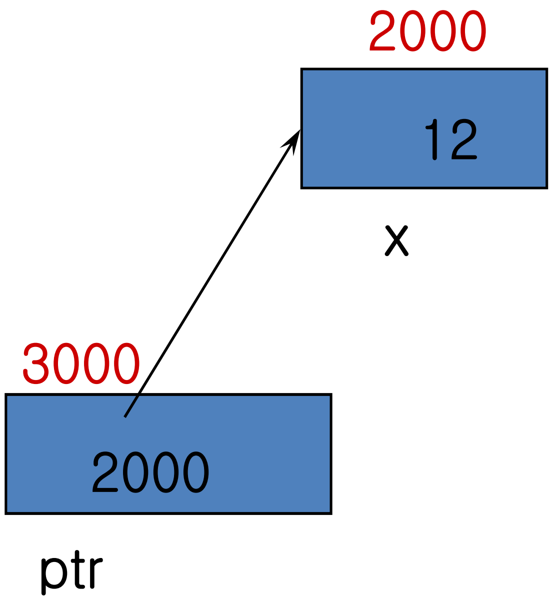
What is a pointer variable?

- A pointer variable is a variable whose value is the address of a location in memory.
- To declare a pointer variable, you must specify the type of value that the pointer will point to.   For example,

```
int*    ptr; // ptr will hold the address of an int

char*   q;   // q will hold the address of a char
```

Using a pointer variable

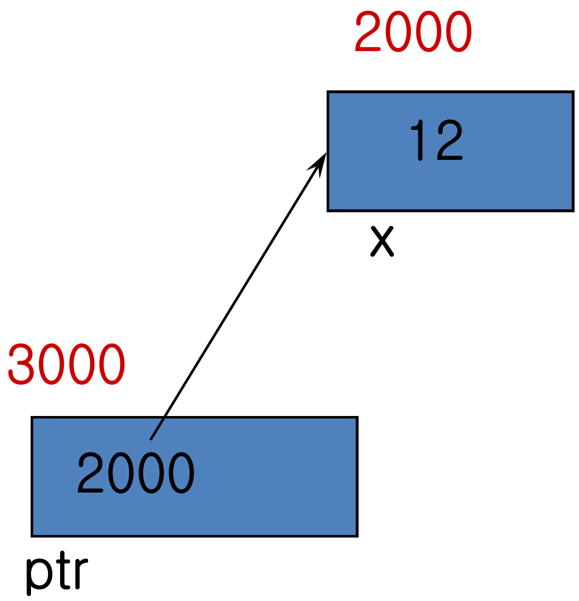
```
int  x;
x = 12;


int*  ptr;
ptr = &x;
```



NOTE: Because ptr holds the address of x, we say that ptr "points to" x

Unary operator * is the deference operator

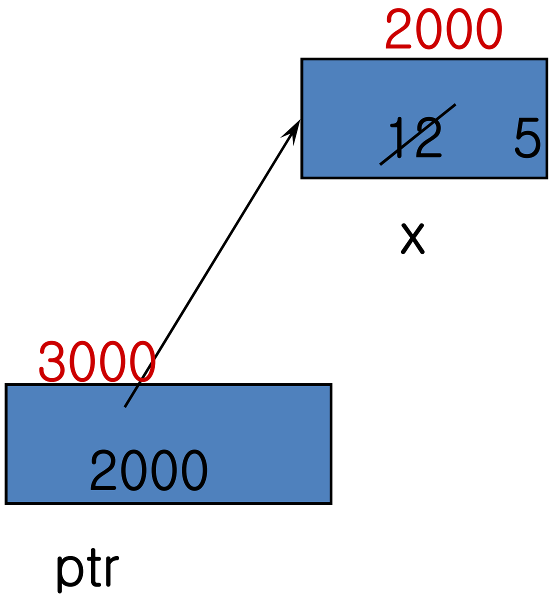
```
int   x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



NOTE: The value pointed to by ptr is denoted by *ptr

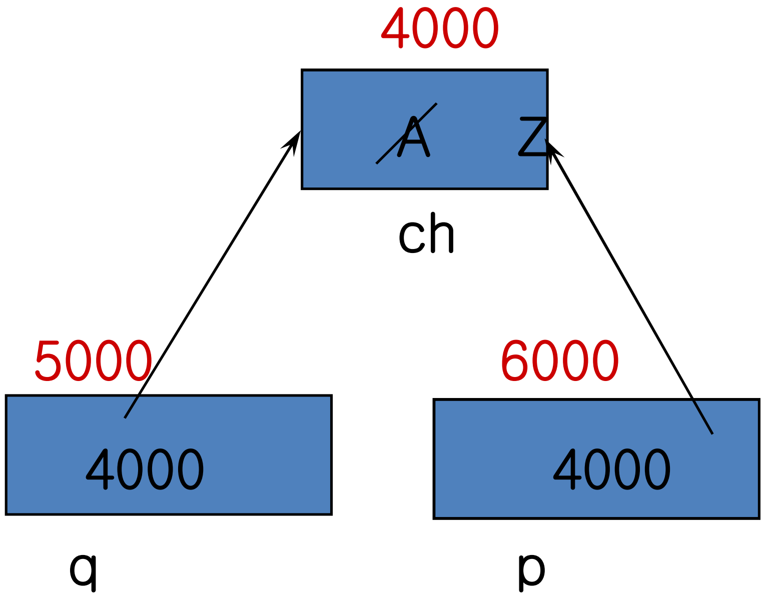
Using the dereference operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;    // changes the value
             // at adddress ptr to 5
```

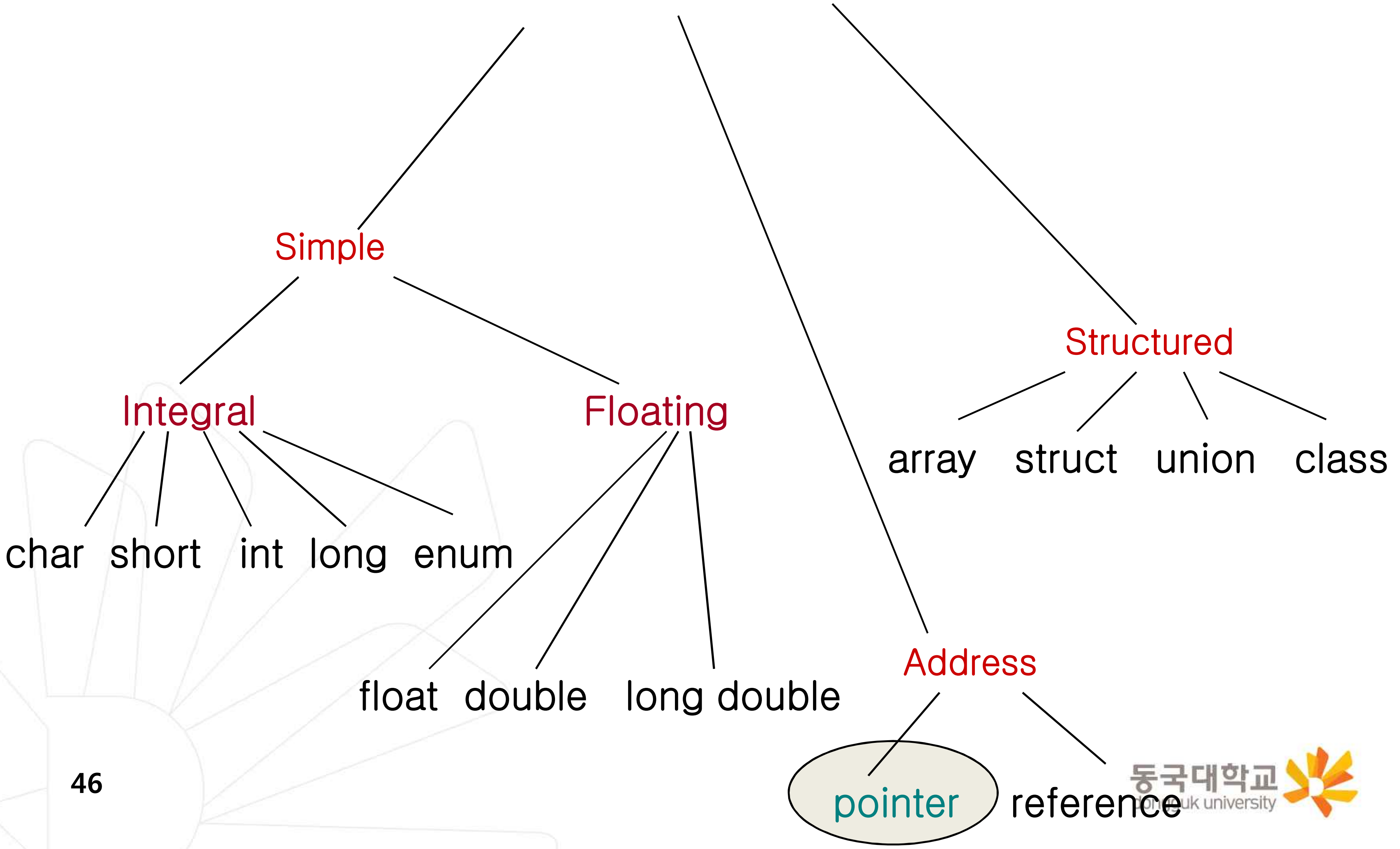


Another Example

```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;    // the right side has value 4000
         // now p and q both point to ch
```



C++ Data Types


Simple
Integral
Floating
char
short
int
long
enum
float
double
long double
Address
pointer
reference
Structured
array
struct
union
class

The NULL Pointer

- There is a pointer constant 0 called the “null pointer” denoted by NULL in stddef.h

- But NULL is not memory address 0.

NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer’s job to check for this.

```
while (ptr != NULL) {
   . . .               // ok to use *ptr here
 }
```

Allocation of memory

STATIC ALLOCATION	DYNAMIC ALLOCATION
Static allocation is the allocation of memory space at compile time.	Dynamic allocation is the allocation of memory space at run time by using operator new.

3 Kinds of Program Data

- **STATIC DATA**: memory allocation exists throughout execution of program.

  ```
  static long SeedValue;
  ```

- **AUTOMATIC DATA**: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function.

- **DYNAMIC DATA**: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using unary operators `new` and `delete`

Using operator `new`

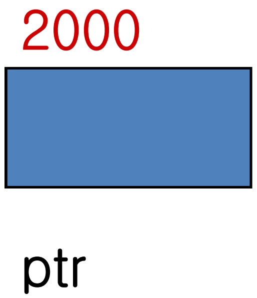
If memory is available in an area called the free store (or heap), operator new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.

Otherwise, the null pointer 0 is returned.

The dynamically allocated object exists until the delete operator destroys it.

Dynamically Allocated Data

```
char*  ptr = 0;

ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
```

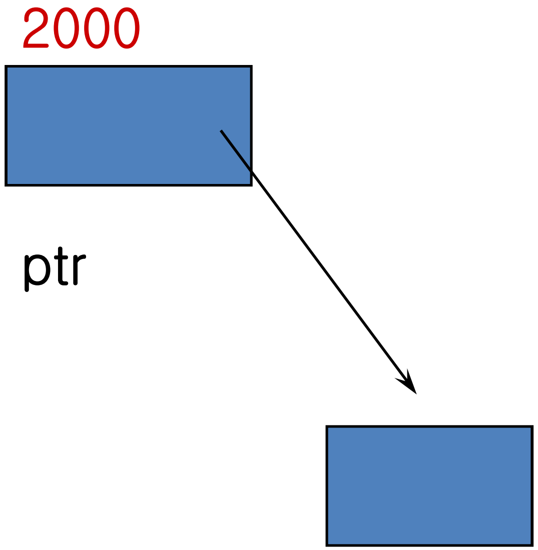


2000

ptr

Dynamically Allocated Data

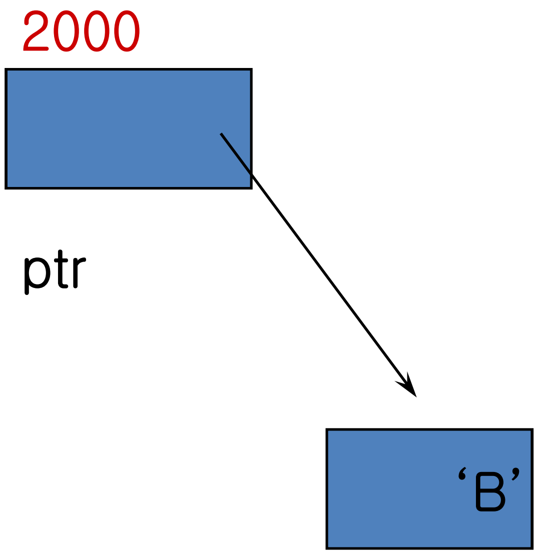
```
char*  ptr;

ptr = new char;
*ptr = 'B';

cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

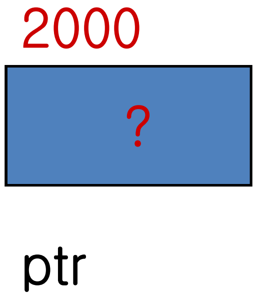
```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;


ptr = new char;

*ptr = ‘B’;

cout  <<  *ptr;

delete  ptr;
```



2000

?

ptr

NOTE: Delete deallocates the memory pointed to by ptr.

Using operator `delete`

- The object or array currently pointed to by the pointer is deallocated, and the pointer is considered unassigned. The memory is returned to the free store.

- Square brackets are used with delete to deallocate a dynamically allocated array of classes.

Some C++ pointer operations

Precedence

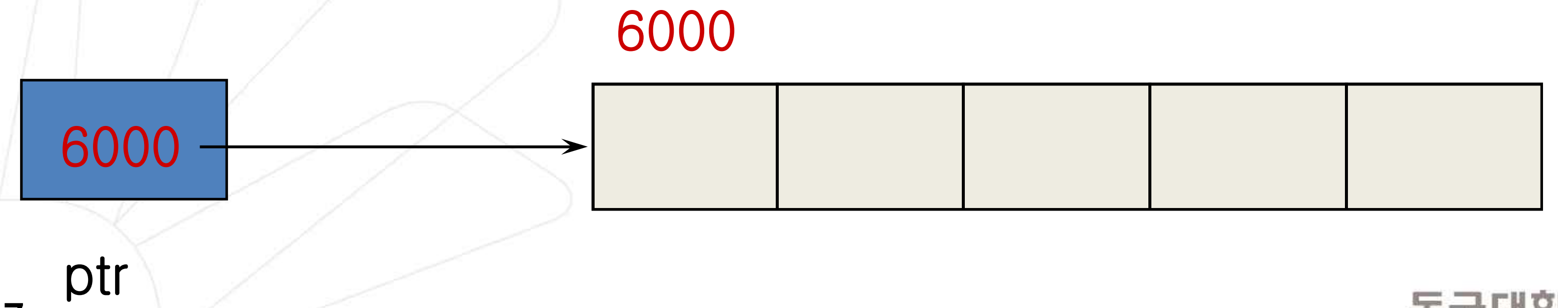
Higher

->	Select member of class pointed to
Unary: ++ -- ! * new delete	Increment, Decrement, NOT, Dereference, Allocate, Deallocate
+ -	Add Subtract
< <= > >=	Relational operators
== !=	Tests for equality, inequality
=	Assignment

Lower

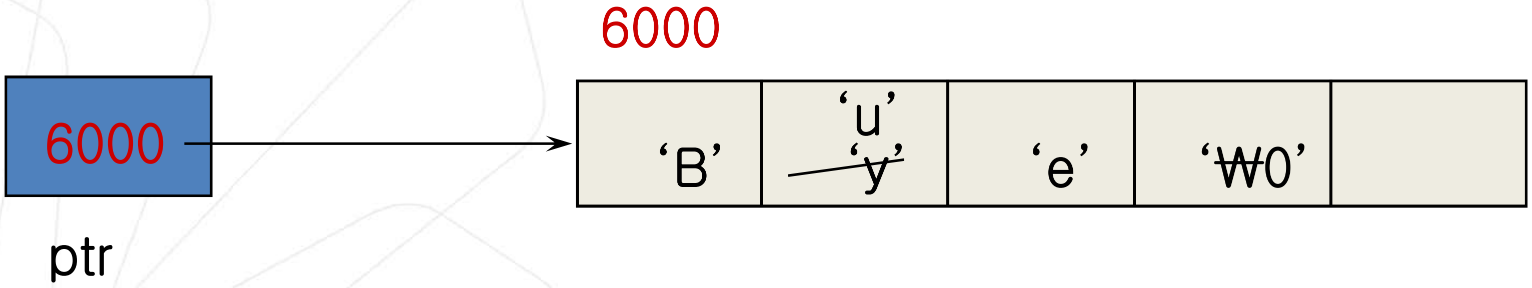
Dynamic Array Allocation

```
char  *ptr;      // ptr is a pointer variable that
                  //  can hold the address of a char

ptr  =  new  char[ 5 ];
             // dynamically, during run time, allocates
             // memory for 5 characters and places into
             //  the contents of ptr their beginning address
```



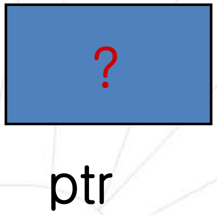
Dynamic Array Allocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, "Bye" );

ptr[ 1 ] = 'u';     // a pointer can be subscripted

cout  << ptr[2] ;
```



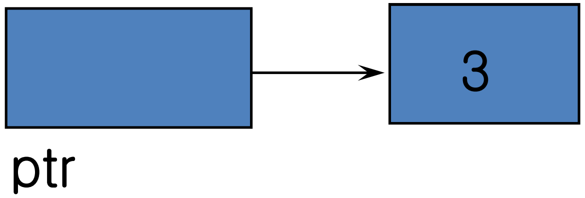
Dynamic Array Deallocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 1 ] = ‘u’;

delete  [ ] ptr; // deallocates array pointed to by ptr
              // ptr itself is not deallocated, but
              // the value of ptr is considered nassigned
```

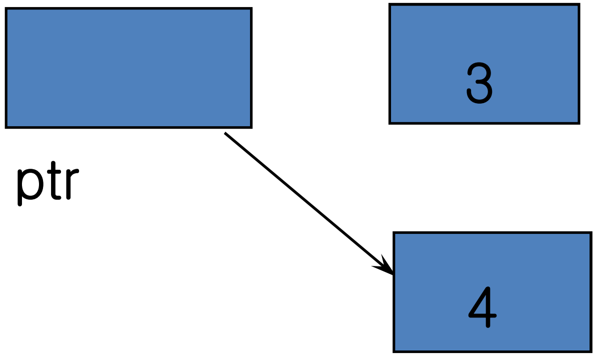


What happens here?

```
int* ptr = new int;
*ptr = 3;
```



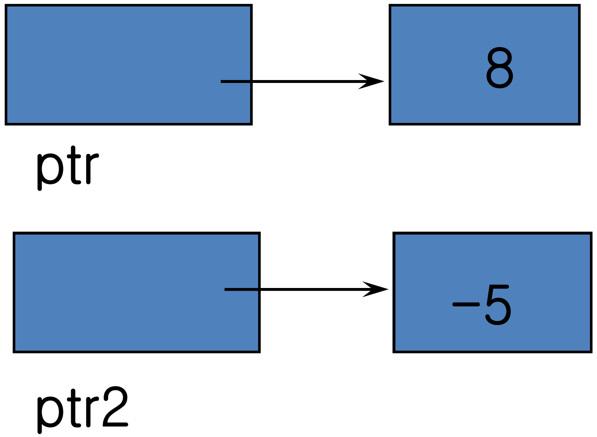
```
ptr = new int;      // changes value of ptr
*ptr = 4;
```



Memory Leak

- A memory leak occurs when dynamic memory (that was created using operator `new`) has been left without a pointer to it by the programmer, and so is inaccessible.

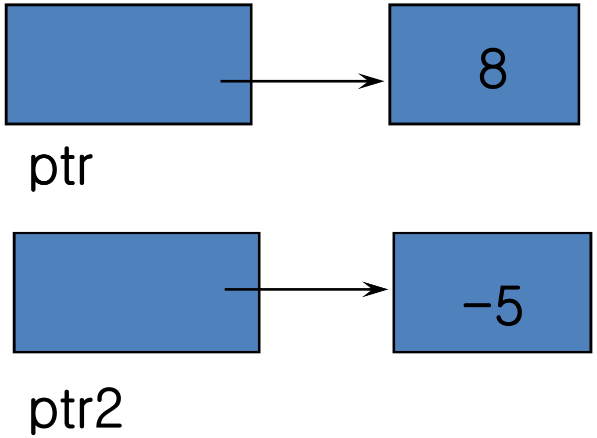
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



How else can an object become inaccessible?

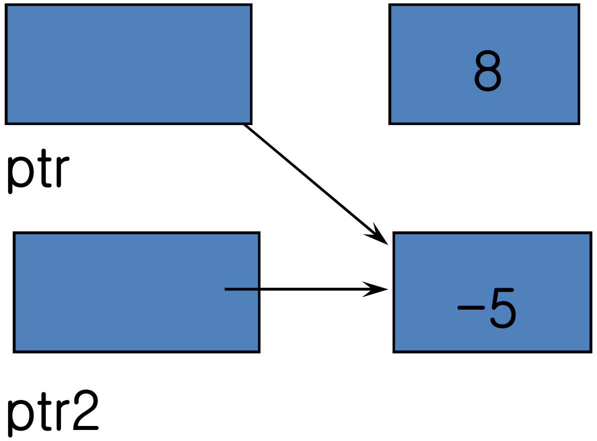
Causing a Memory Leak

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



```
ptr = ptr2;
```

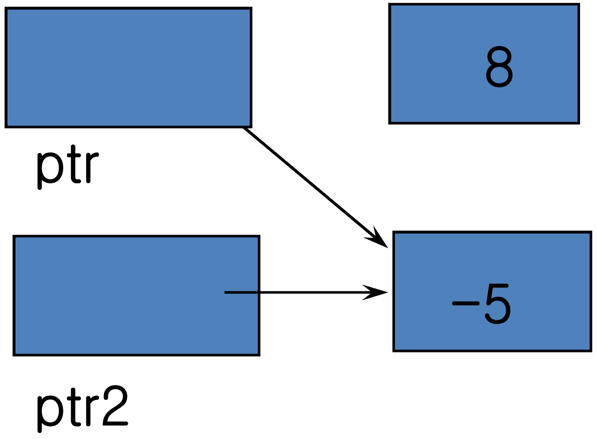
// here the 8 becomes inaccessible



A Dangling Pointer

- occurs when two pointers point to the same object and delete is applied to one of them.

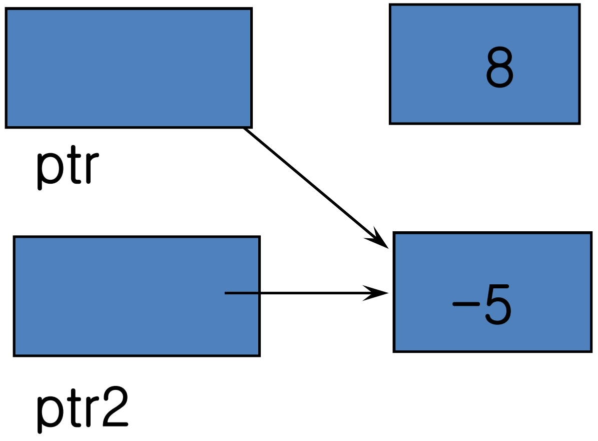
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



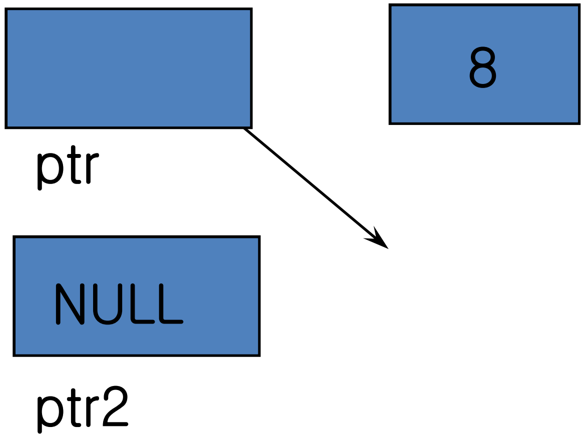
FOR EXAMPLE,

Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



```
delete ptr2;     // ptr is left dangling
ptr2 = NULL;
```


8
ptr
NULL
ptr2