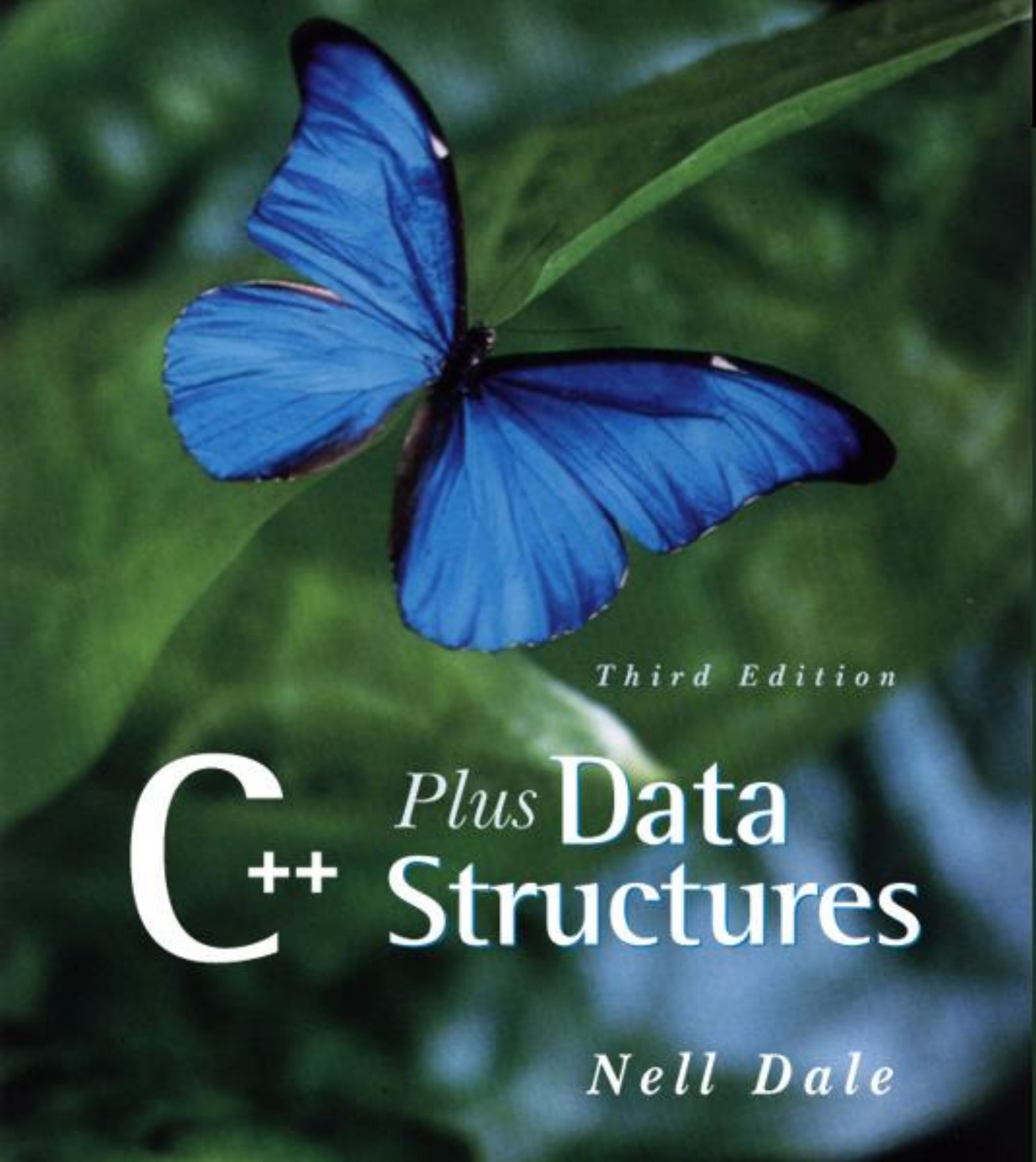
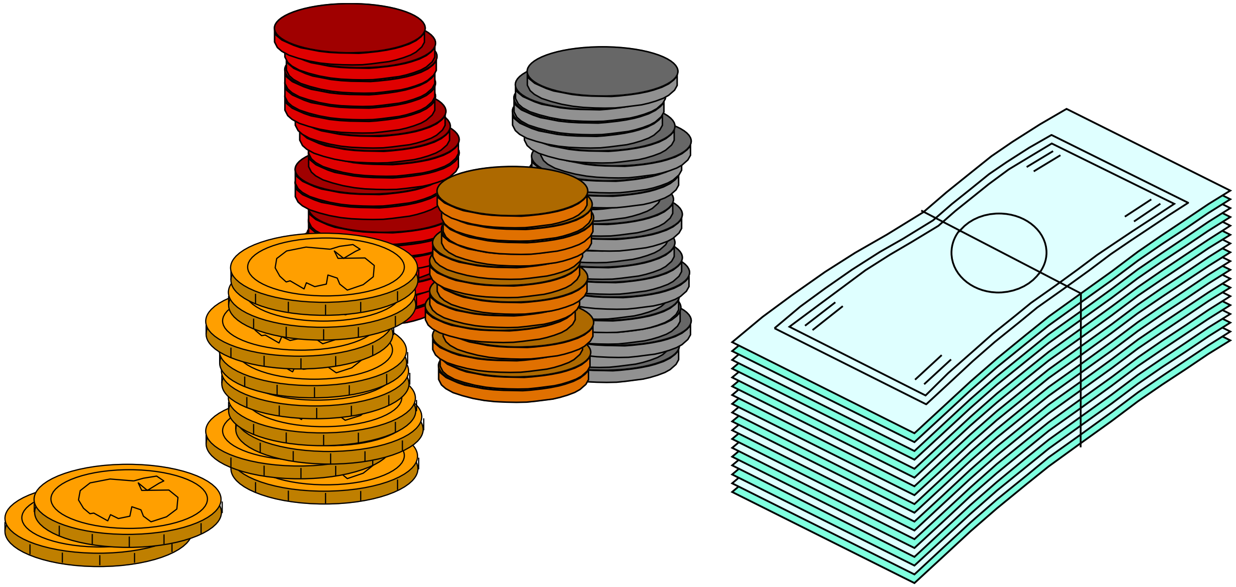


Chapter

4

ADTs Stack and Queue

Stacks of Coins and Bills



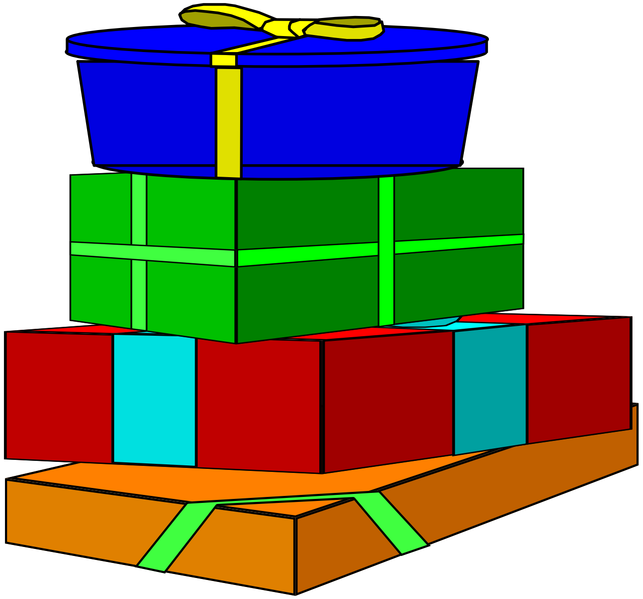
What is a Stack?

- ***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.

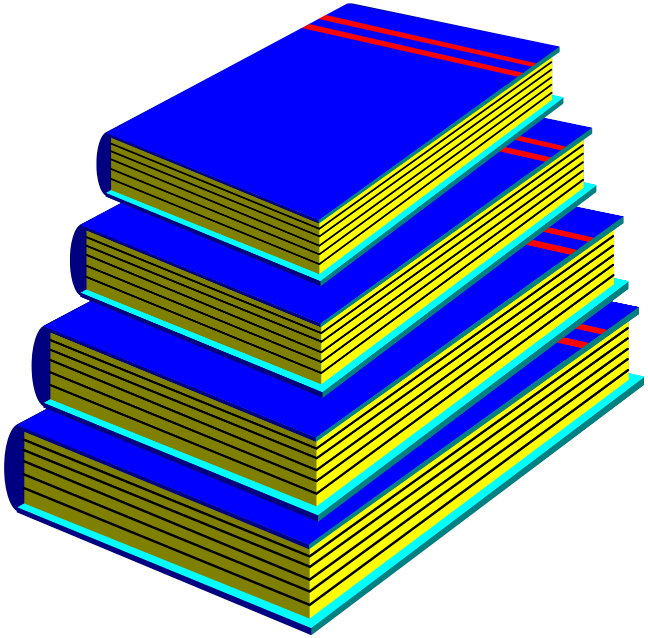
- A stack is a LIFO “last in, first out” structure.

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)** -- Throws exception if stack is full; otherwise adds newItem to the top of the stack.
- **Pop** -- Throws exception if stack is empty; otherwise removes the item at the top of the stack.
- **ItemType Top** -- Throws exception if stack is empty; otherwise returns a copy of the top item

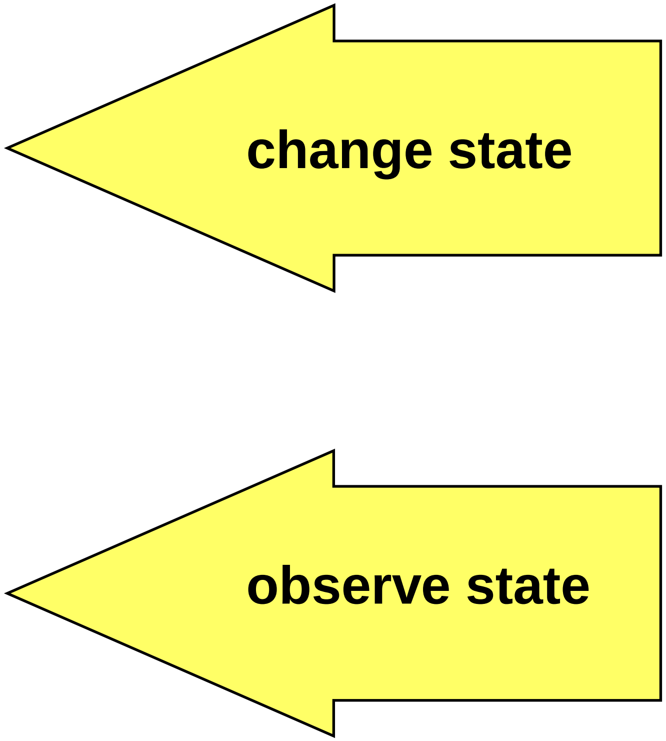
ADT Stack Operations

Transformers

- Push
- Pop

Observers

- IsEmpty
- IsFull
- IsFull



```
// Class specification for Stack ADT in file StackType.h

class FullStack                // Exception class thrown by
                               // Push when stack is full
{};
class EmptyStack               // Exception class thrown by
                              // Pop and Top when stack is empty
{};

#include "ItemType.h"

class StackType
{
public:
   StackType( );
   // Class constructor.
   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.
   // Post: If (stack is full), FullStack exception is thrown;
  //        otherwise, newItem is at the top of the stack.
  void Pop();
   // Function: Removes top item from the stack.
   // Pre: Stack has been initialized.
   // Post: If (stack is empty), EmptyStack exception is thrown;
  //        otherwise, top element has been removed from stack.
  ItemType Top();
   // Function: Returns a copy of top item on the stack.
   // Pre: Stack has been initialized.
   // Post: If (stack is empty), EmptyStack exception is thrown;
   //       otherwise, top element has been removed from stack.
private:
   int top;
   ItemType  items[MAX_ITEMS];
};
```

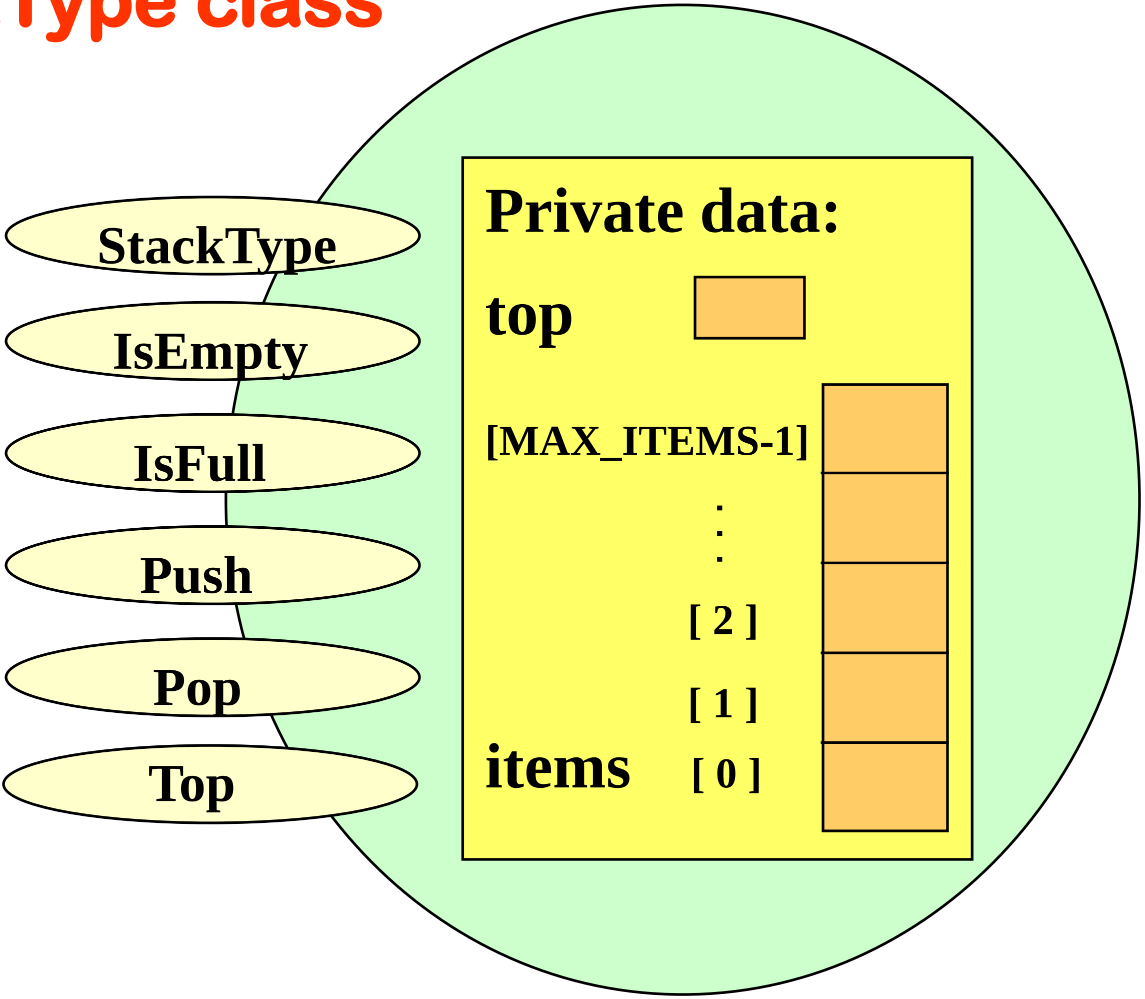
```
// File: StackType.cpp

#include "StackType.h"
#include <iostream>
StackType::StackType( )
{
    top = -1;
}
bool StackType::IsEmpty() const
{
    return(top = = -1);
}

bool StackType::IsFull() const
{
    return (top = = MAX_ITEMS-1);
}
```

```
void StackType::Push(ItemType newItem)
{
   if( IsFull() )
      throw FullStack():
   top++;
   items[top] = newItem;
}

void StackType::Pop()
{
   if( IsEmpty() )
     throw EmptyStack();
    top--;
}

ItemType StackType::Top()
{
   if (IsEmpty())
     throw EmptyStack();
  return items[top];
}
```

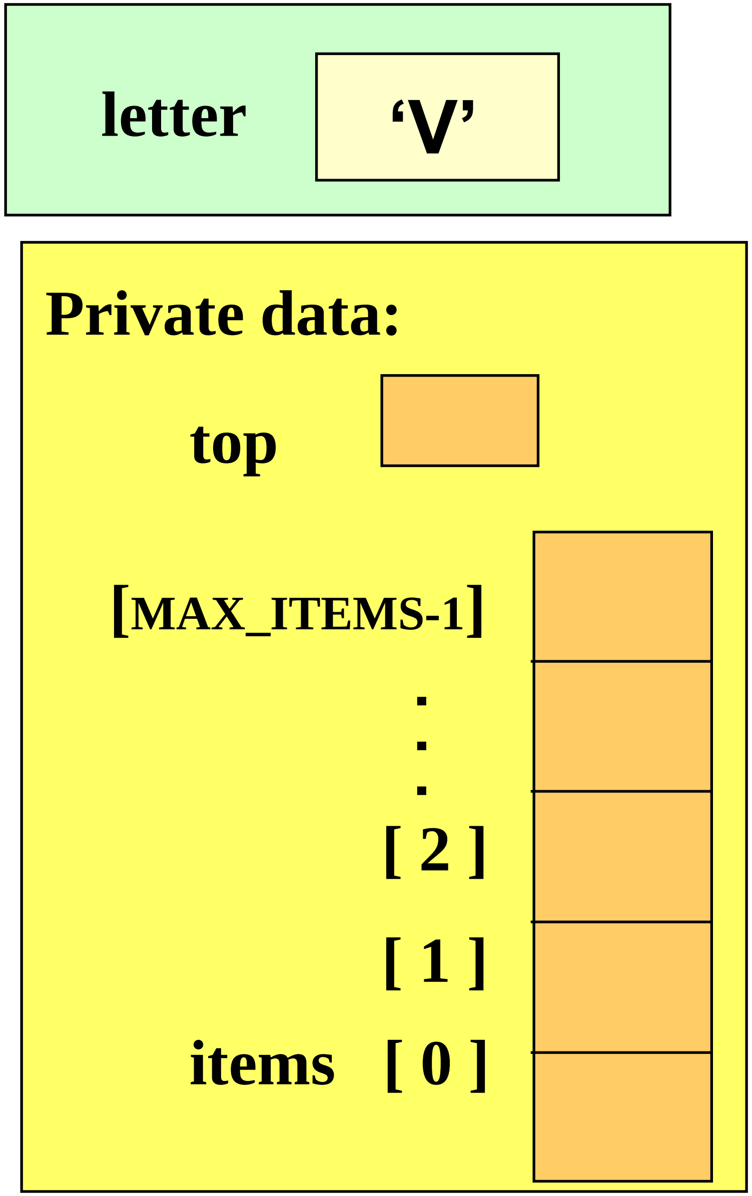
Class Interface Diagram

(Memory reversed to better illustrate concept)

StackType class



Tracing Client Code



letter ‘V’

Private data:

top

[MAX_ITEMS-1]

.
.
.

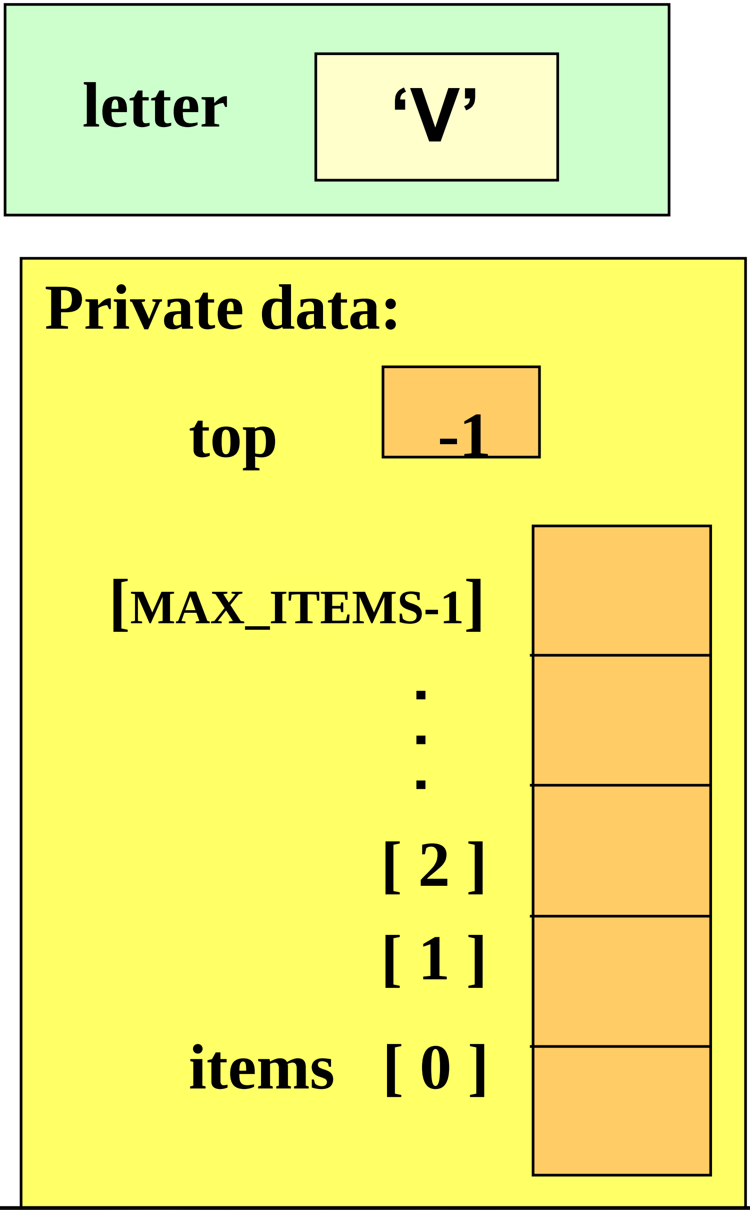
[2]

[1]

items [0]

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘V’

Private data:

top -1

[MAX_ITEMS-1]

.
.
.

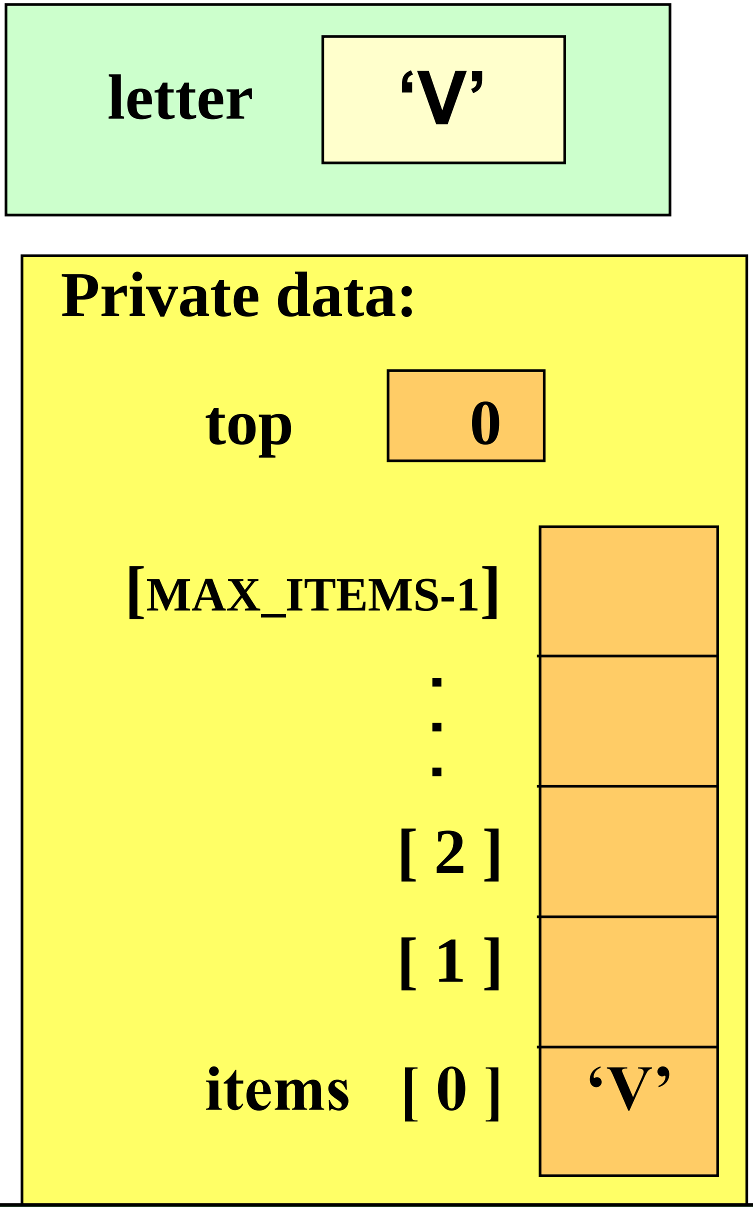
[2]

[1]

items [0]

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code


letter ‘V’

Private data:

top 0

[MAX_ITEMS-1]

.
.
.

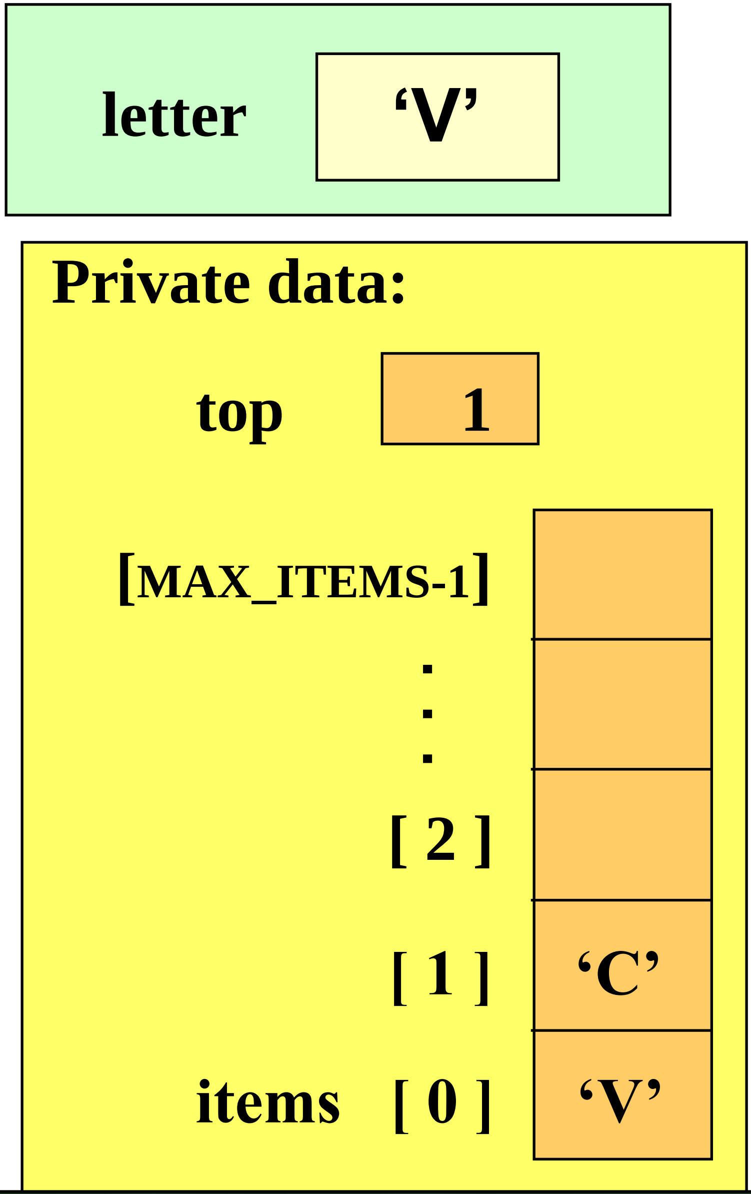
[2]

[1]

items [0] ‘V’

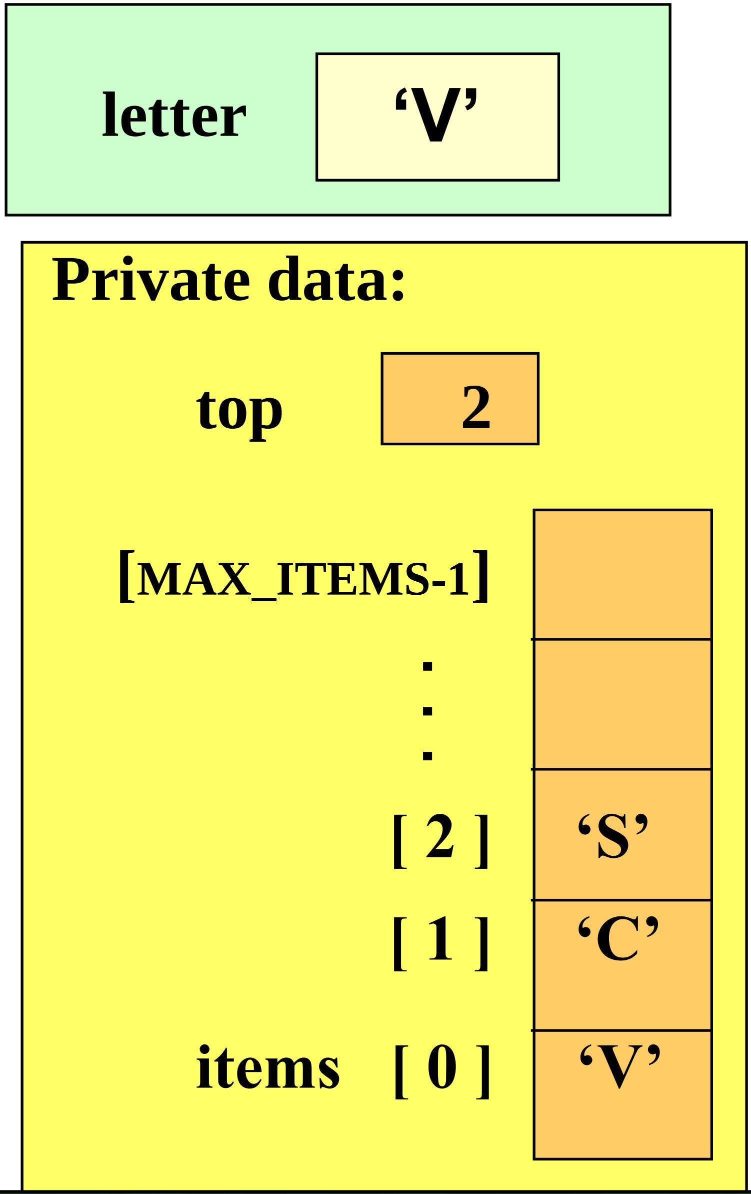
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘V’

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

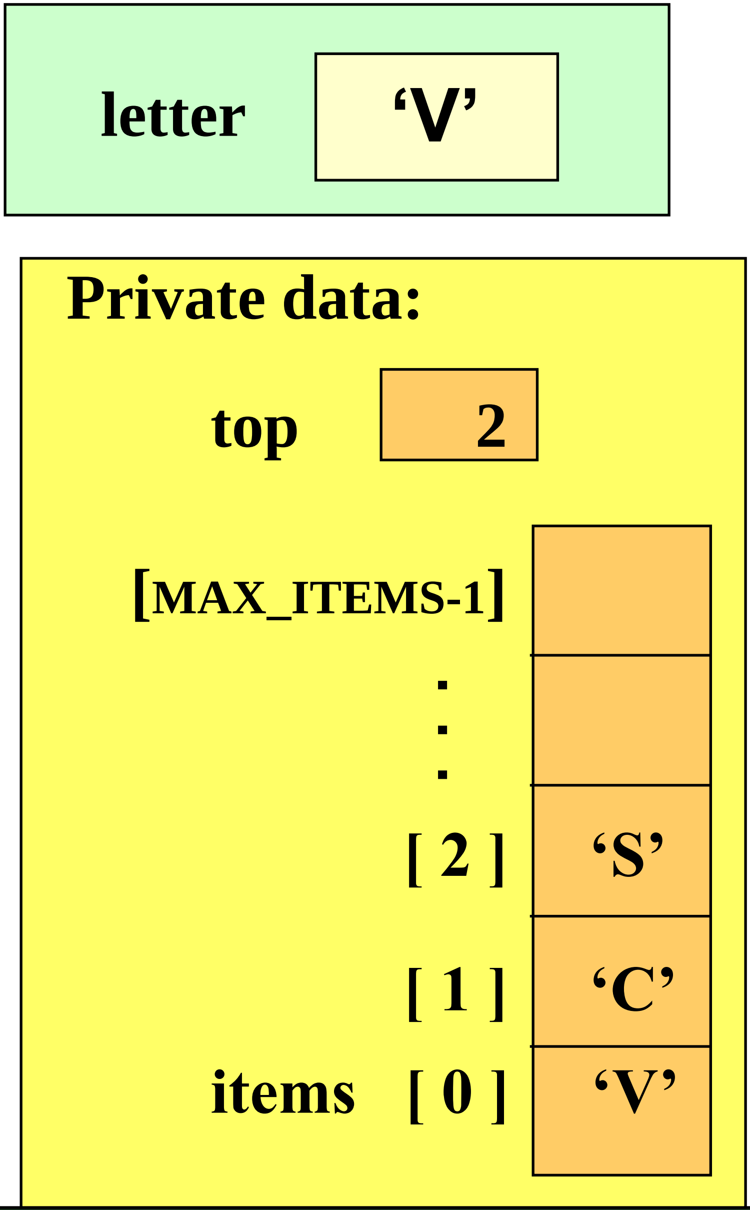
[2] ‘S’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter 'V'

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

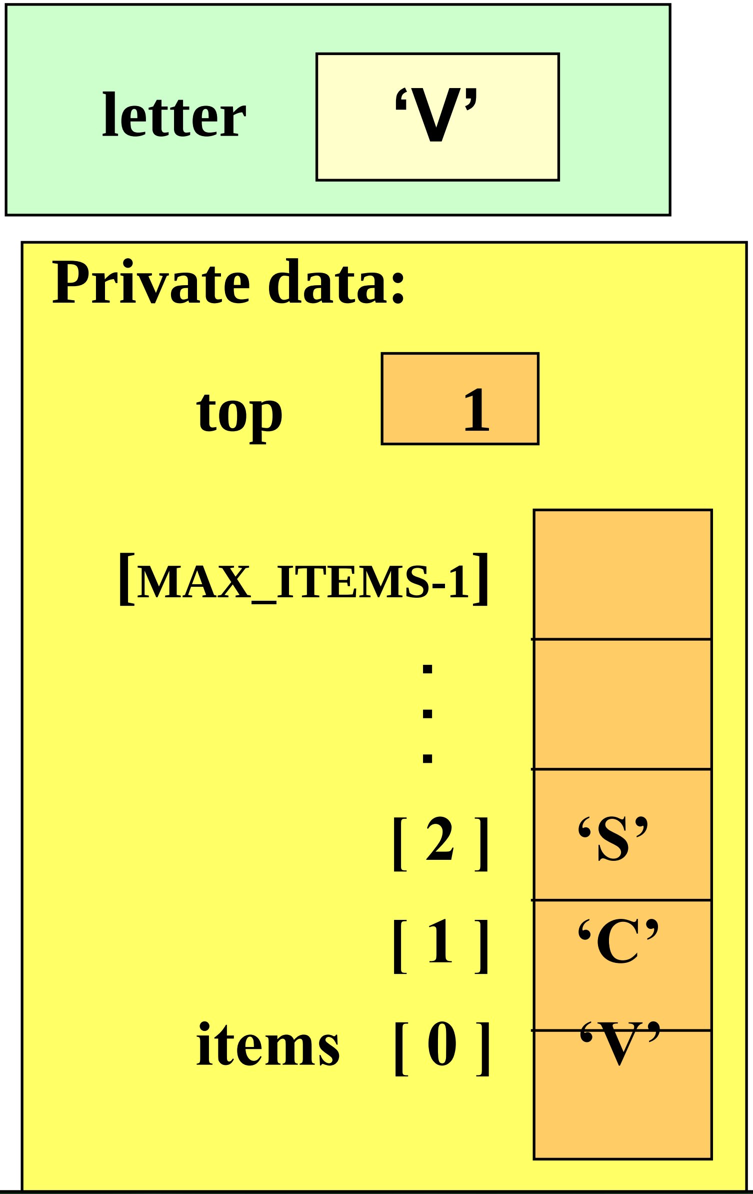
[2] 'S'

[1] 'C'

items [0] 'V'

```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘V’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

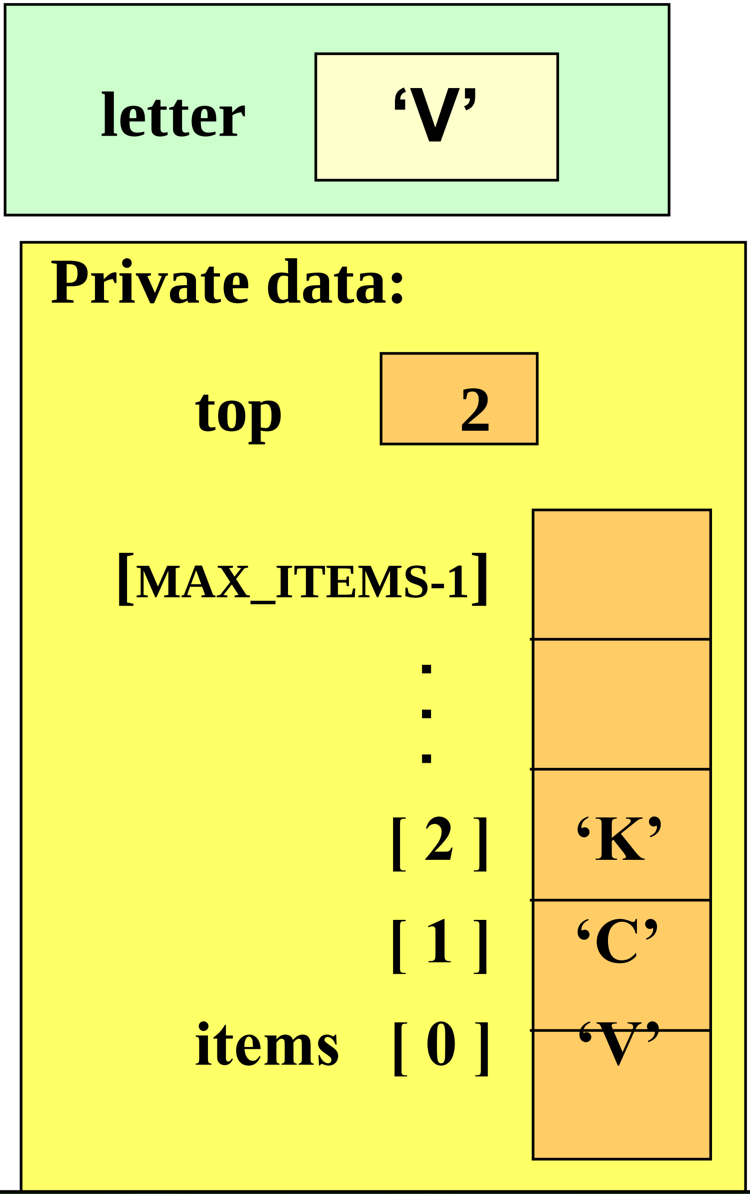
[2] ‘S’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘V’

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

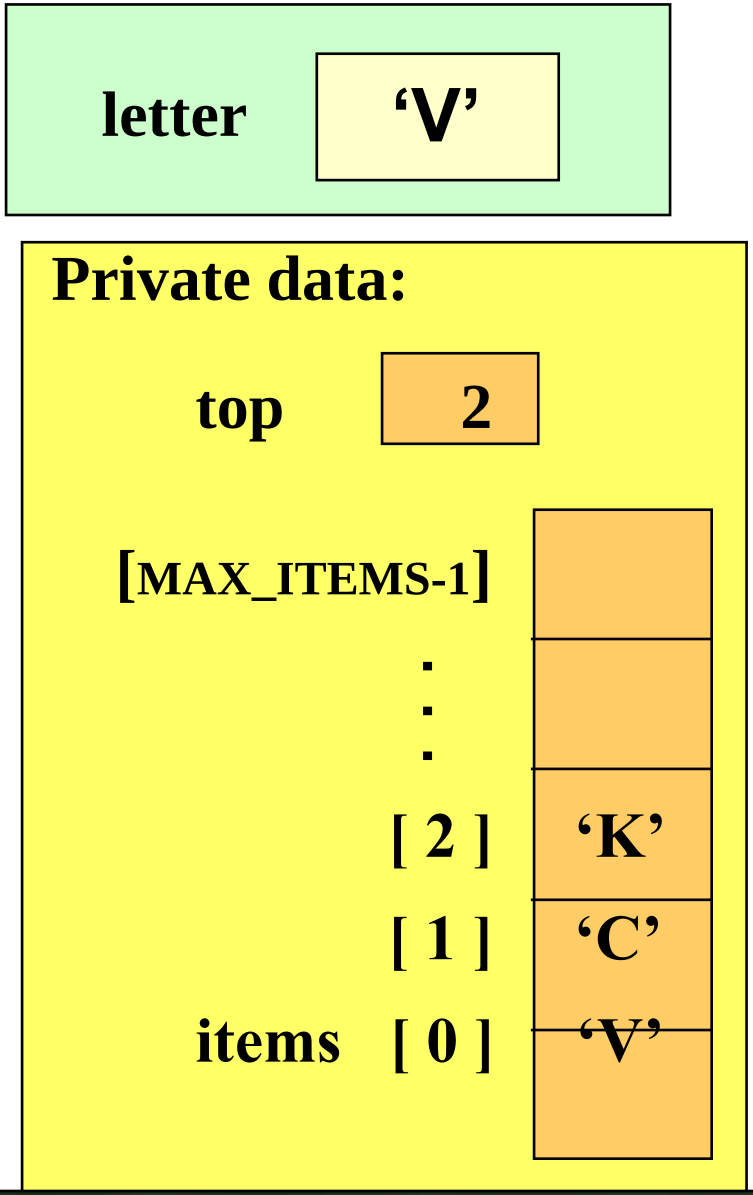
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘V’

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

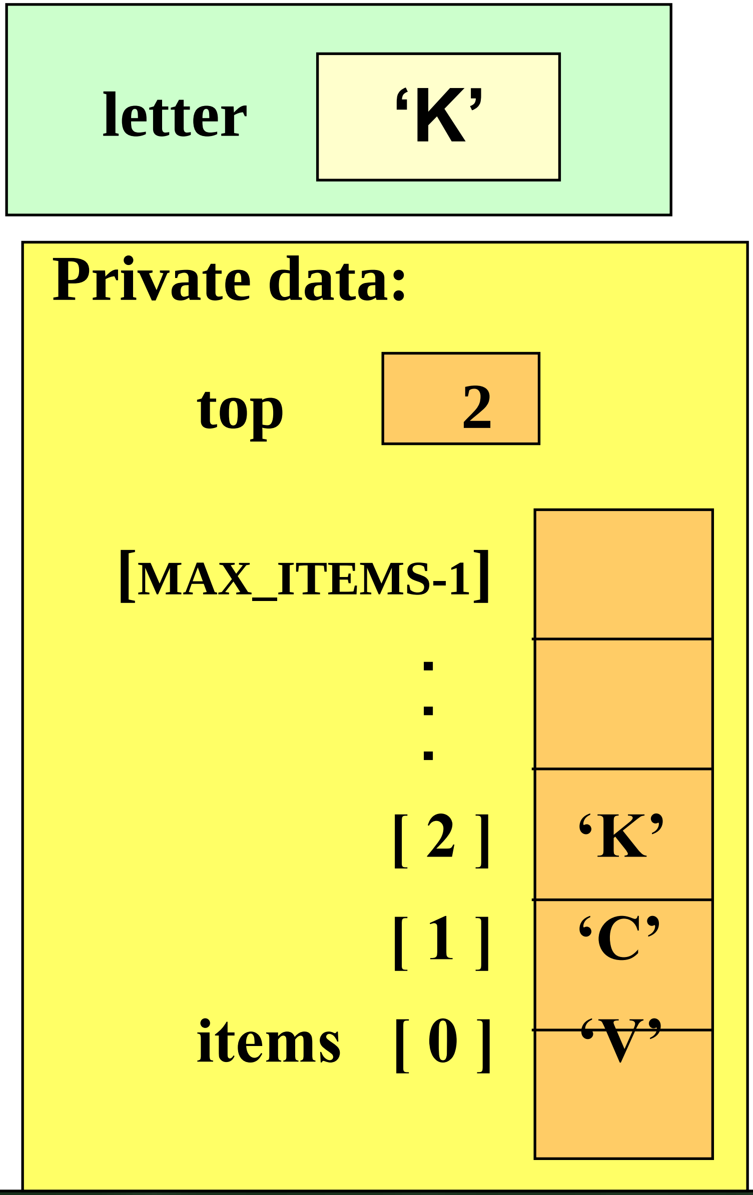
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘K’

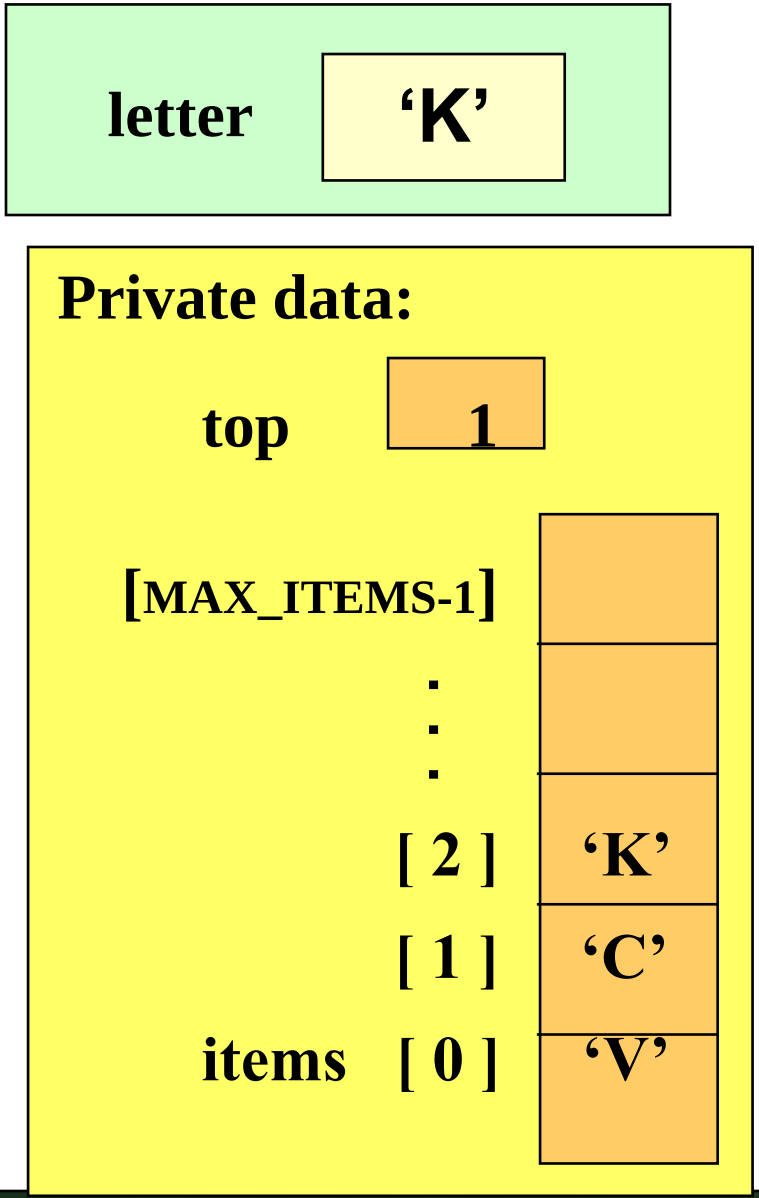
Private data:

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( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

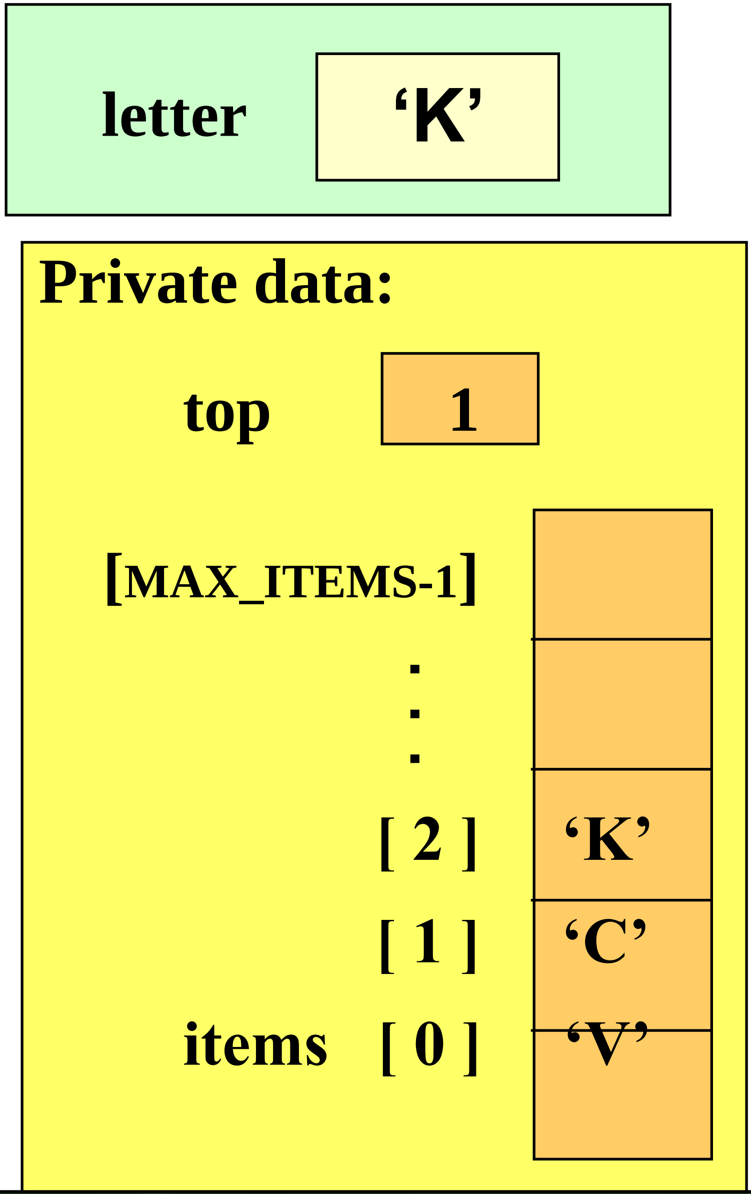
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

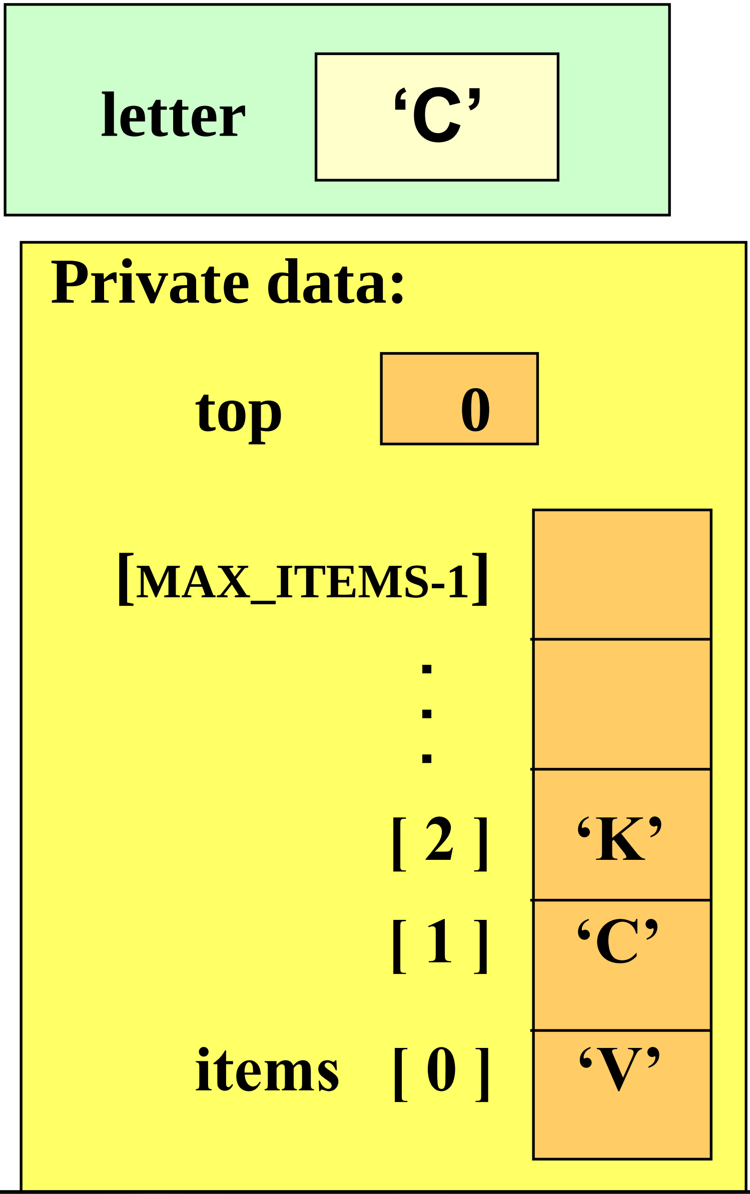
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘C’

Private data:

top 0

[MAX_ITEMS-1]

.
.
.

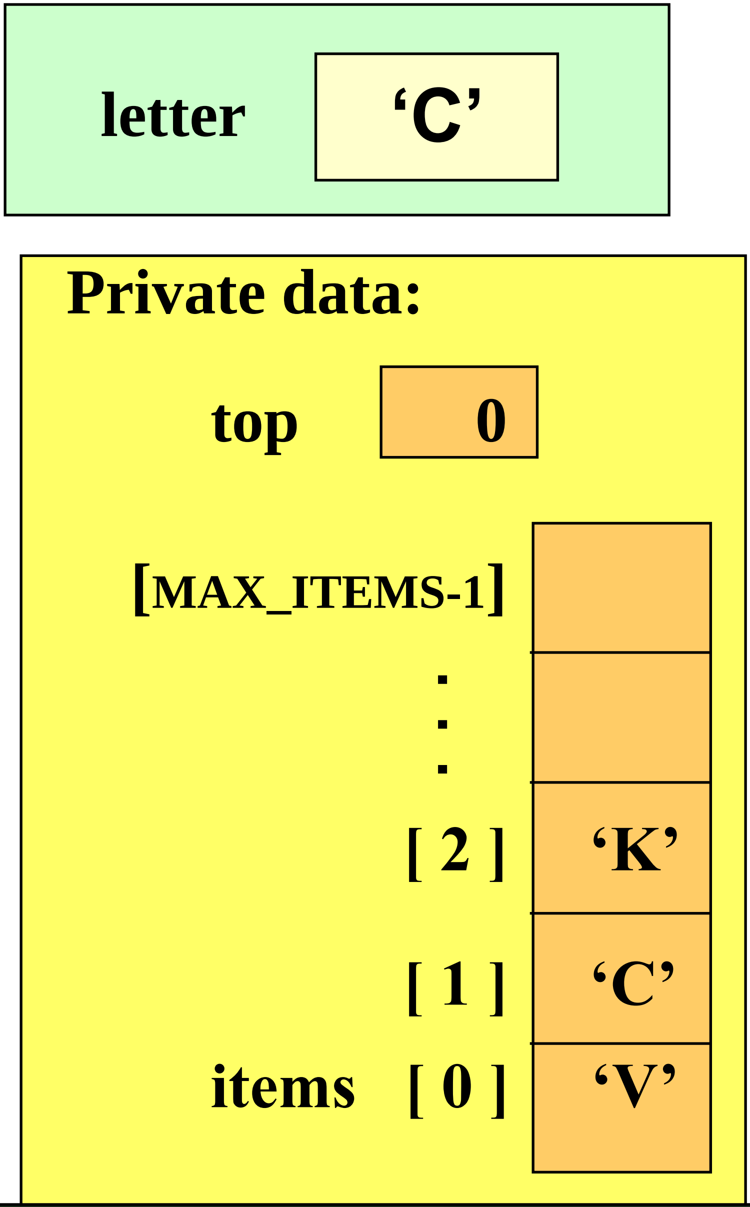
[2] ‘K’

[1] ‘C’

items [0] ‘V’

```
char    letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



letter ‘C’

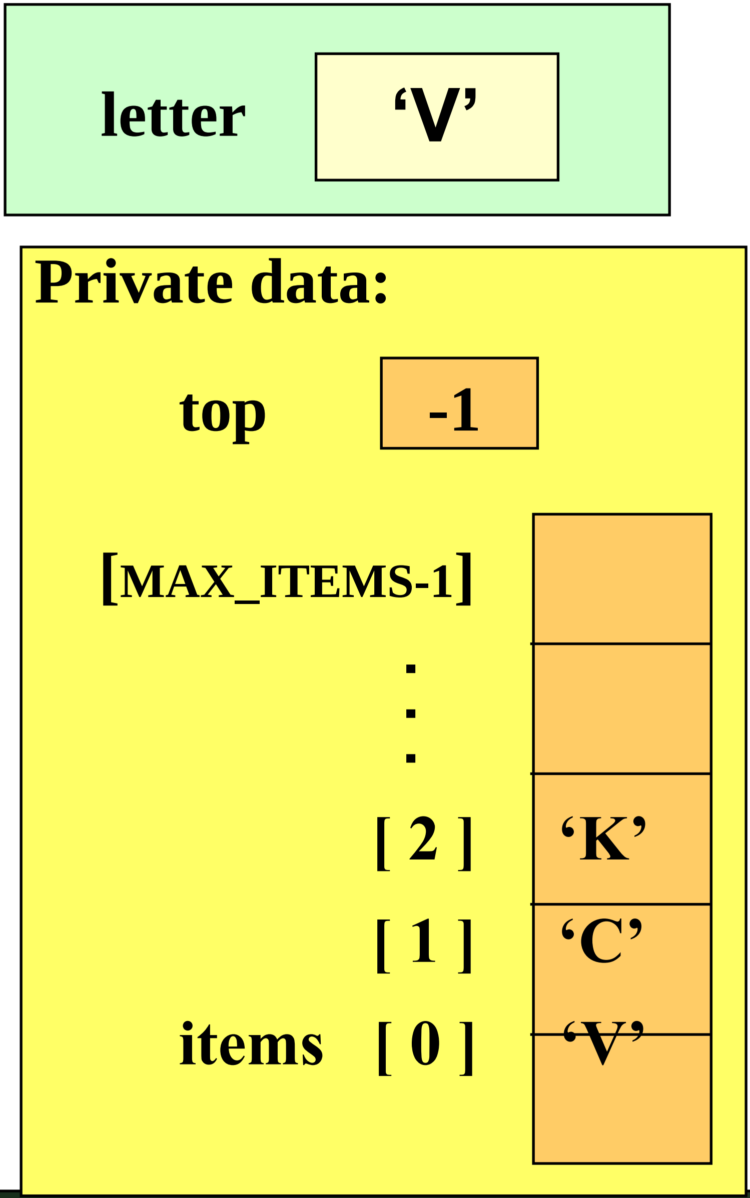
Private data:

top 0

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

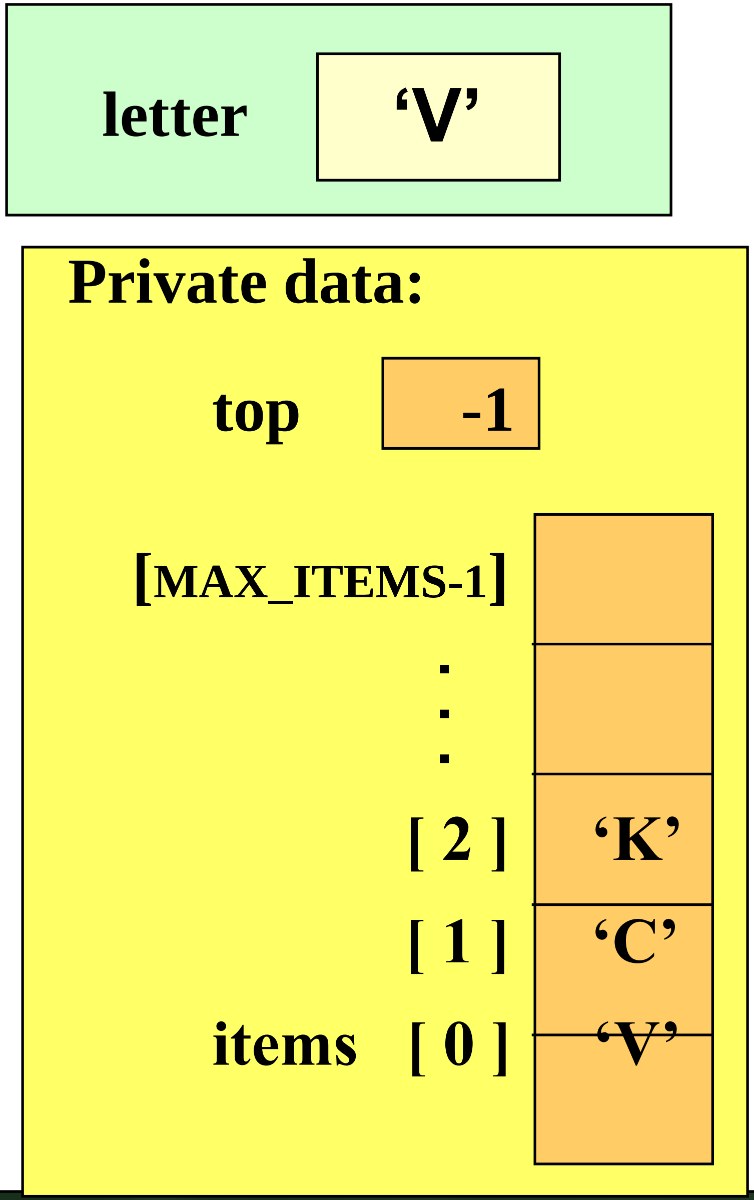
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

End of Trace



letter ‘V’

Private data:

top -1

[MAX_ITEMS-1]

.
.
.

[2] ‘K’

[1] ‘C’

items [0] ‘V’

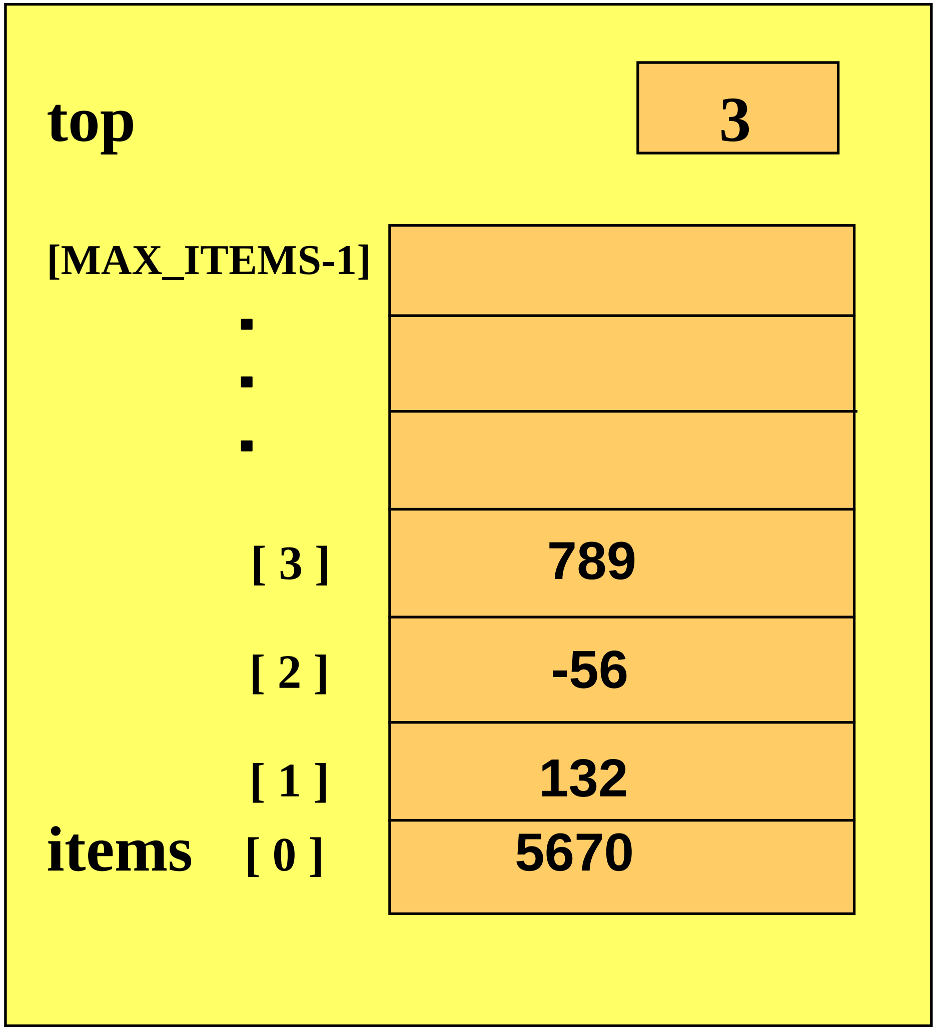
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

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, < >.

ACTUAL PARAMETER

```
StackType<int> numStack;
```



top 3

[MAX_ITEMS-1]

.
.
.

[3] 789

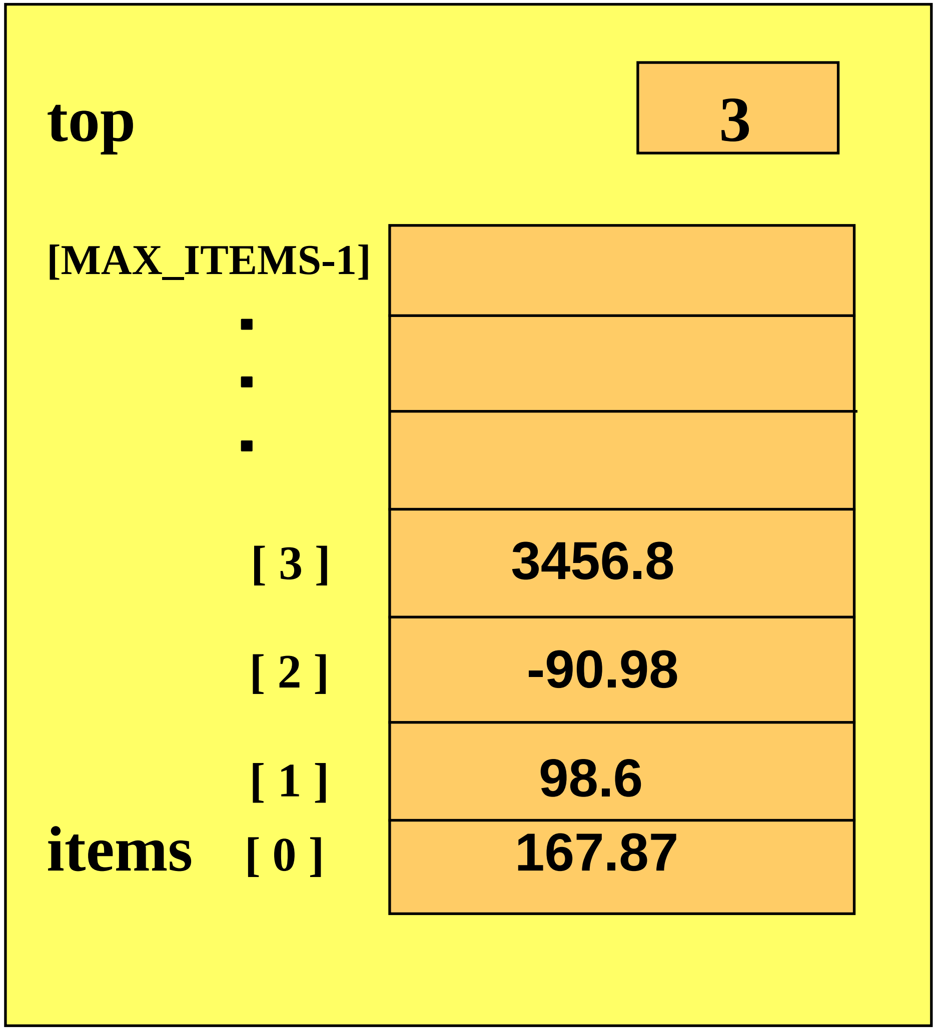
[2] -56

[1] 132

items [0] 5670

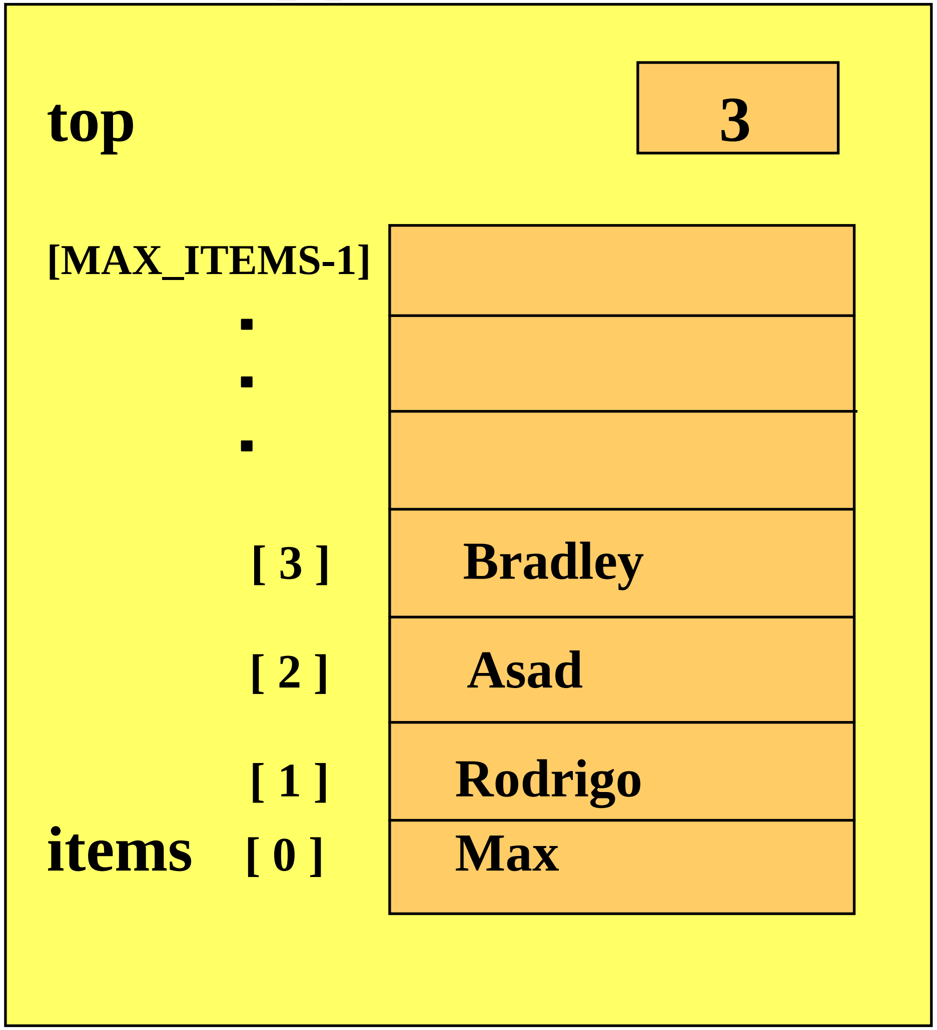
ACTUAL PARAMETER

```
StackType<float> numStack;
```



ACTUAL PARAMETER

```
StackType<StrType> numStack;
```



top 3

[MAX_ITEMS-1]

.
.
.

[3] Bradley

[2] Asad

[1] Rodrigo

items [0] Max

```
//--------------------------------------------------------
// CLASS TEMPLATE DEFINITION
//--------------------------------------------------------
#include "ItemType.h"      // for MAX_ITEMS and ItemTyp

template<class ItemType>   // formal parameter list
class StackType
{
public:
   StackType( );
   bool IsEmpty( ) const;
   bool IsFull( ) const;
   void Push( ItemType item );
   void Pop( ItemType&  item );
   ItemType Top( );
private:
   int        top;
   ItemType   items[MAX_ITEMS];
};
```

```
//--------------------------------------------------------
// SAMPLE CLASS MEMBER FUNCTIONS
//--------------------------------------------------------

template<class ItemType>   // formal parameter list
StackType<ItemType>::StackType( )
{
   top = -1;
}
```

Notice that the class name is StackType<ItemType>

```
template<class ItemType>   // formal parameter list
void StackType<ItemType>::Push ( ItemType newItem )
{
   if (IsFull())
     throw FullStack();
    top++;
   items[top] = newItem;   // STATIC ARRAY IMPLEMENTATION
}
```

Using class templates

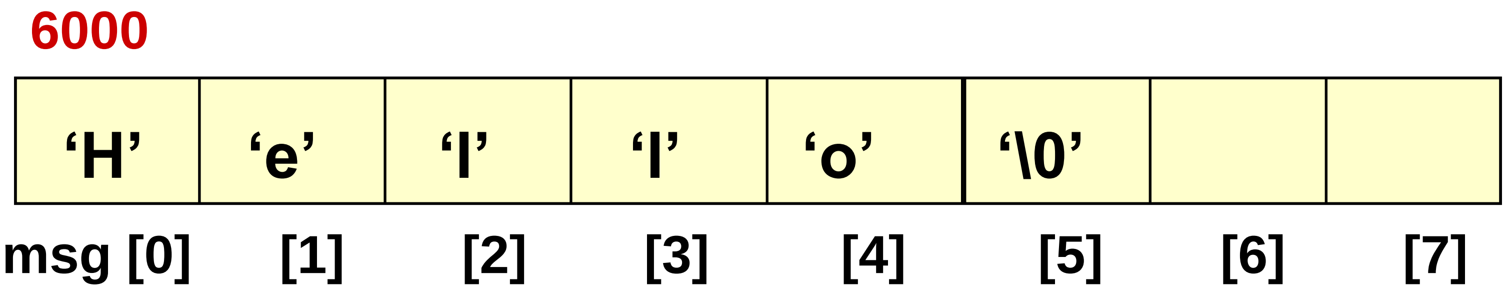
- The actual parameter to the template is a data type. Any type can be used, either built-in or user-defined.

- When creating class template
  - Put .h and .cpp in same file or
  - Have .h include .cpp file

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`.



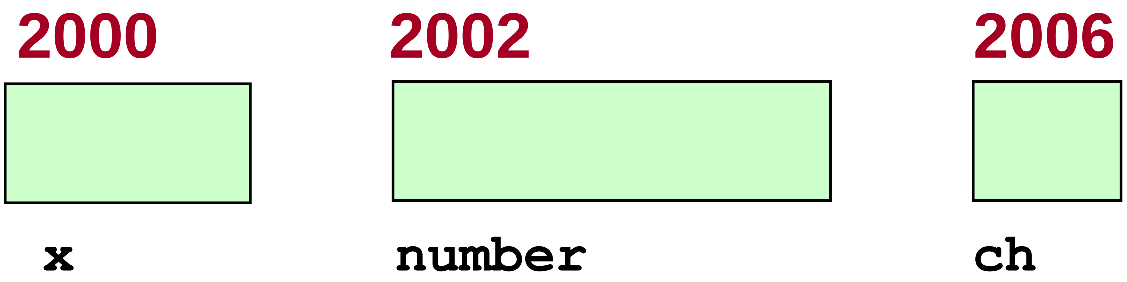
6000

‘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. For example:

```
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 &.

```
using namespace std;
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;
```

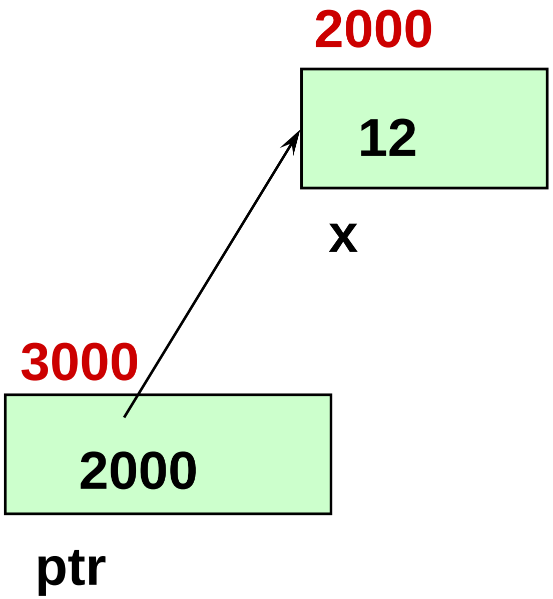
What is a pointer variable?

- 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;
```



2000

12

x

3000

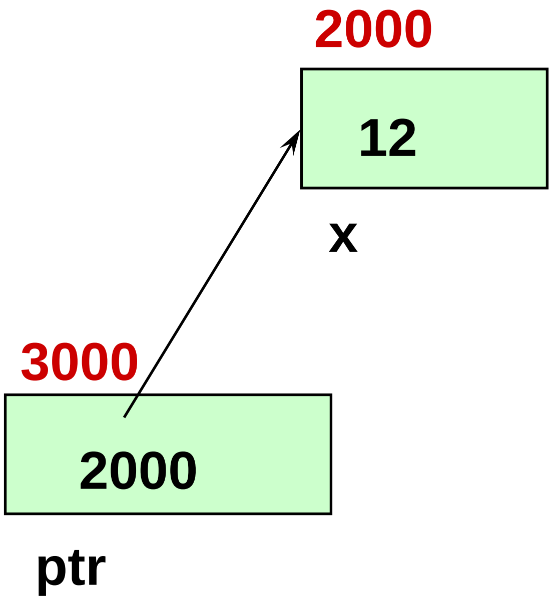
2000

ptr

NOTE: Because `ptr` holds the address of `x`, we say that `ptr` “points to” `x`

Unary operator * is the deference (indirection) operator

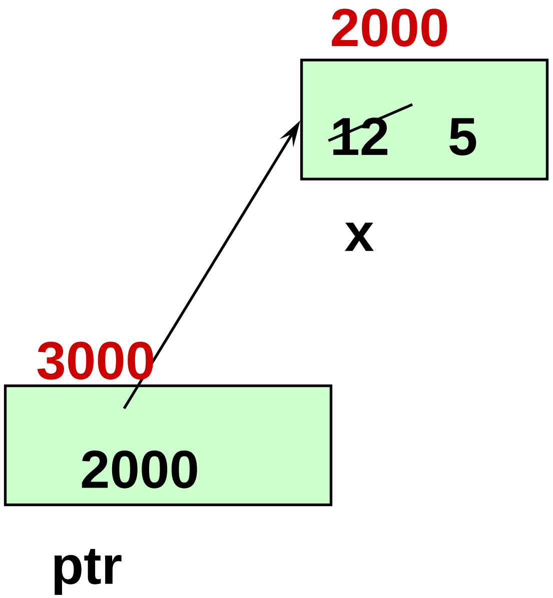
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

std::cout  <<  *ptr;
```



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

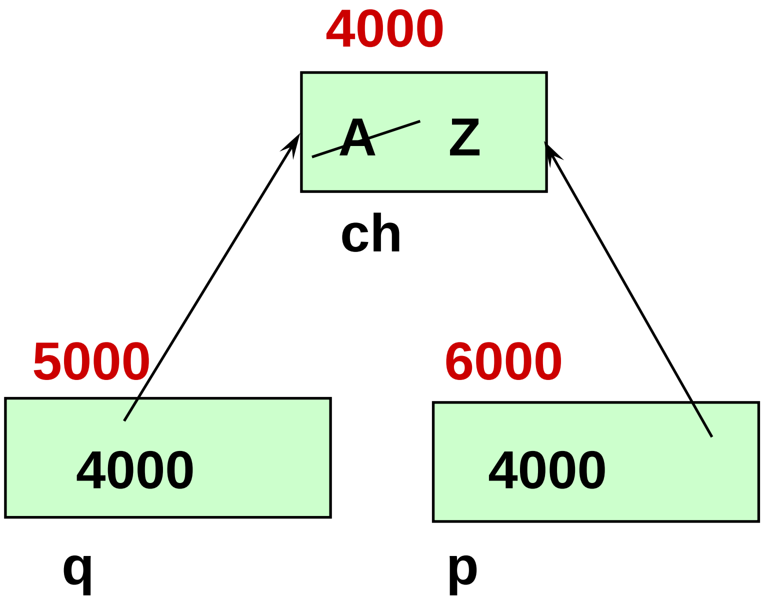
Using the dereference operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

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

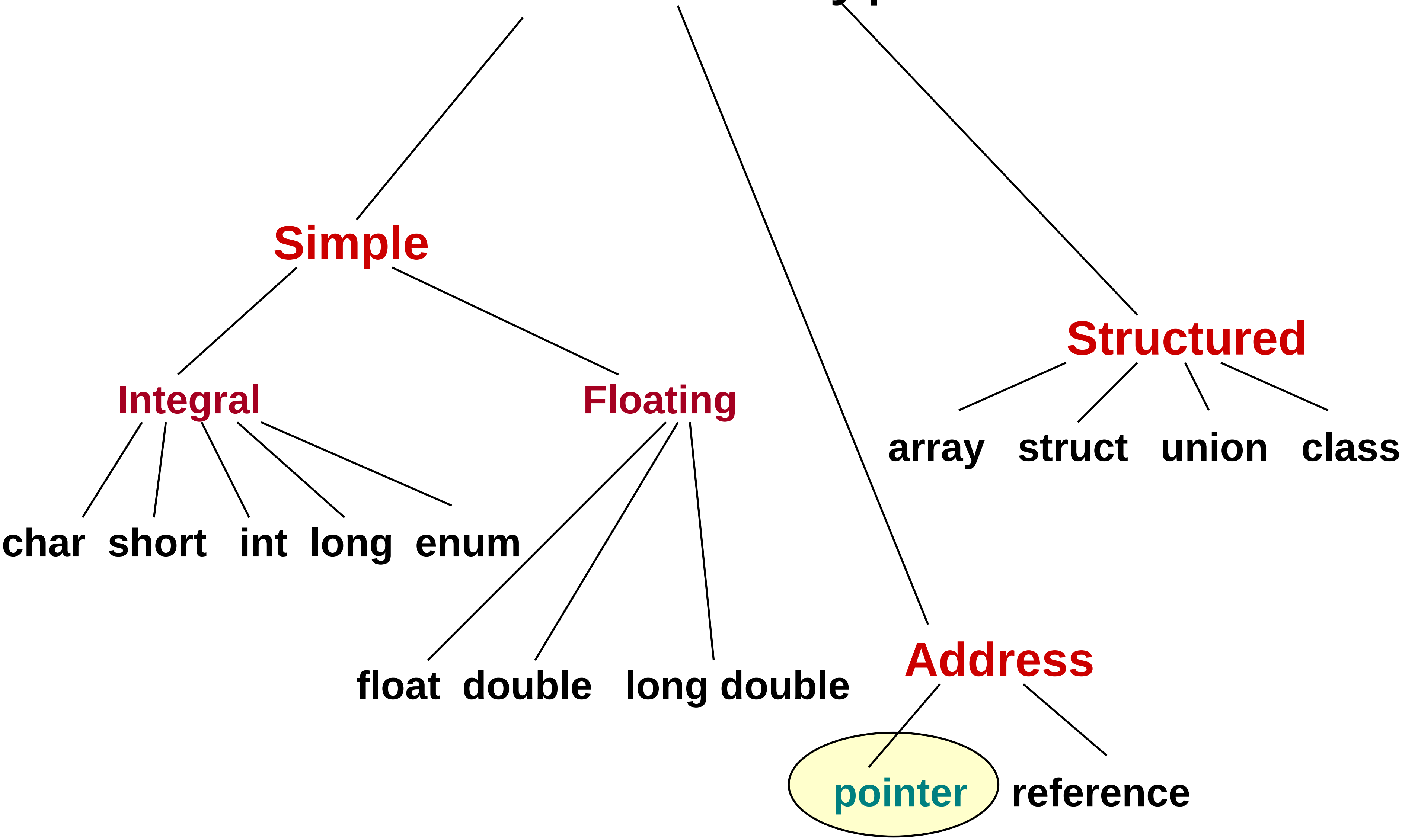


Another Example

```
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

char short int long enum

Floating

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 cstddef.

But NULL is not memory address 0.

NULL allows a pointer to point nothing

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

Static allocation is the allocation of memory space at compile time.

DYNAMIC ALLOCATION

Dynamic allocation is the allocation of memory space at run time by using operator new.

AUTOMATIC ALLOCATION도 있음

3 Kinds of Lifetime for 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`**

* **The *lifetime* of a variable is the time during program execution when the variable has storage assigned to it.**

Using operator `new`

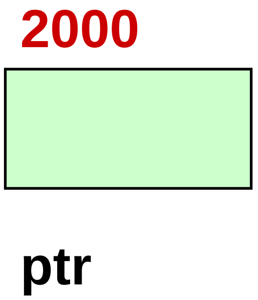
If memory is available in an area called the free store (or heap), operator new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.

Otherwise, the null pointer 0 is returned.

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

Dynamically Allocated Data

```
char*  ptr;

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

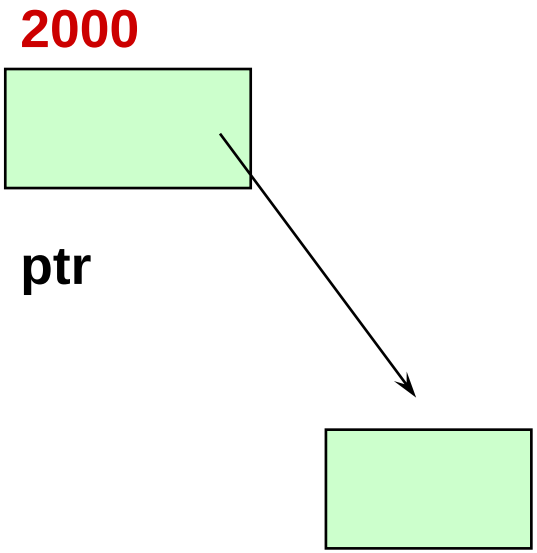


2000

ptr

Dynamically Allocated Data

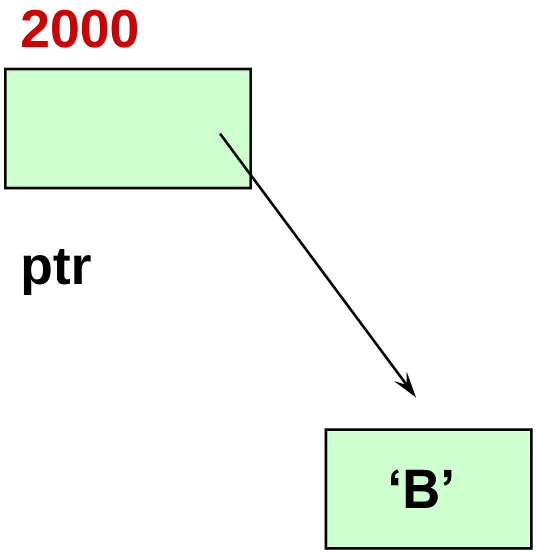
```
char*  ptr;

ptr = new char;
*ptr = ‘B’;

std::cout << *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

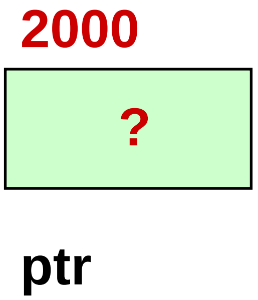
```
char*  ptr;


ptr = new char;
*ptr = ‘B’;

std::cout << *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;


ptr = new char;

*ptr = ‘B’;

std::cout << *ptr;

delete  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

```
MoneyType* moneyPtr = new MoneyType;
moneyPtr->dollars = 3245;
(*moneyPtr).cents = 33;  // NO *moneyPtr.cents = 33;
```

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
Lower	=	Assignment