BINUS UNIVERSITY INTERNATIONAL

Data Structures and Algorithms

Stacks

Week 3

seraphina@binus.ac.id

BINUS UNIVERSITY INTERNATIONAL

https://1drv.ms/u/s!AkfRgh1gcQzbmyPhuGw5loOJtA23?e=Q3scy9

dsa

BINUS UNIVERSITY INTERNATIONAL

Session Learning Outcomes

Upon completion of this session, students are expected to be able to

- CILO3 Apply appropriate data structures and simple algorithms for solving computing problems.
- CILO4 Design computer programs by applying different data structures and related algorithms

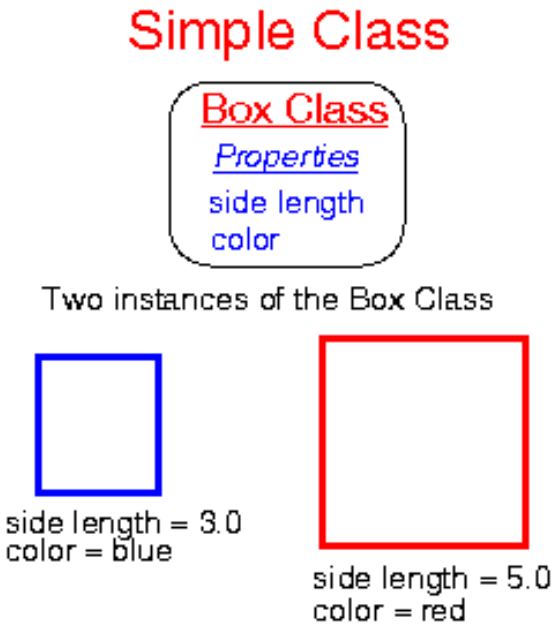
BINUS UNIVERSITY INTERNATIONAL

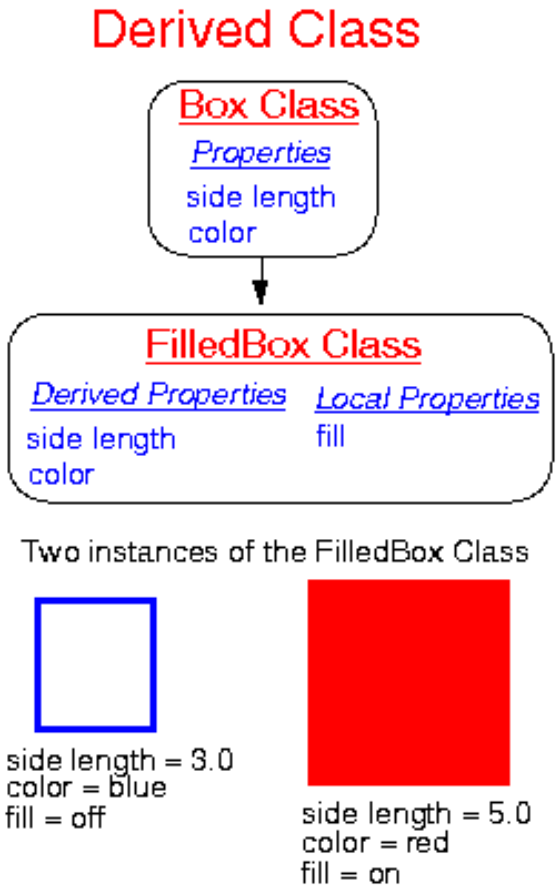
Topics

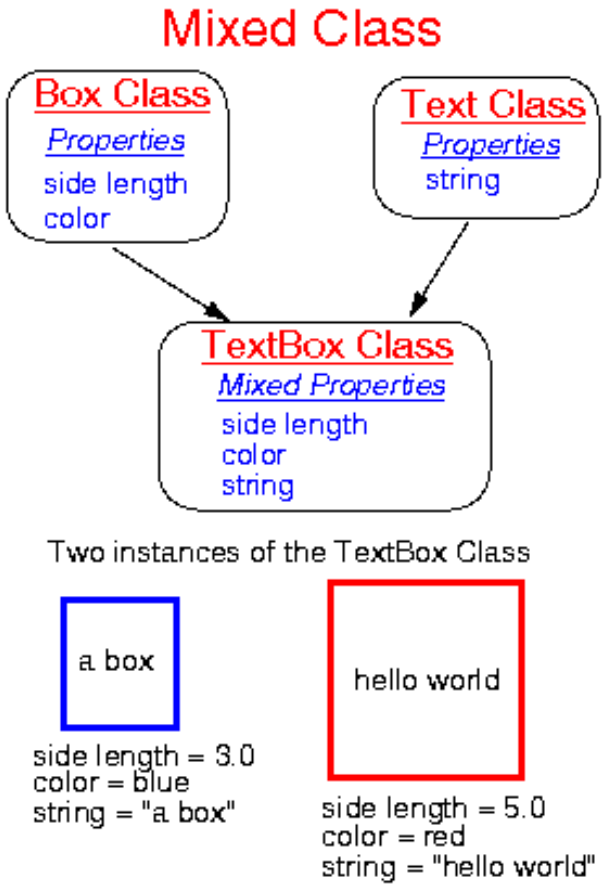
- Review
  - Class, object, method, property
  - Stack
  - Array and Linked List
    Array – more on lab sessions
- Pointer
- Stack

Review

- Class VS Object
  - A class is a “template” / “blueprint” for objects







https://www.ncl.ucar.edu/Document/HLUs/User_Guide/classes/classoview.shtml

Class / Object / Method / Property?

setSpeed

car

drive

orange

sit

mercedes

dalmatian

getFuel

peach

maxspeed

dog

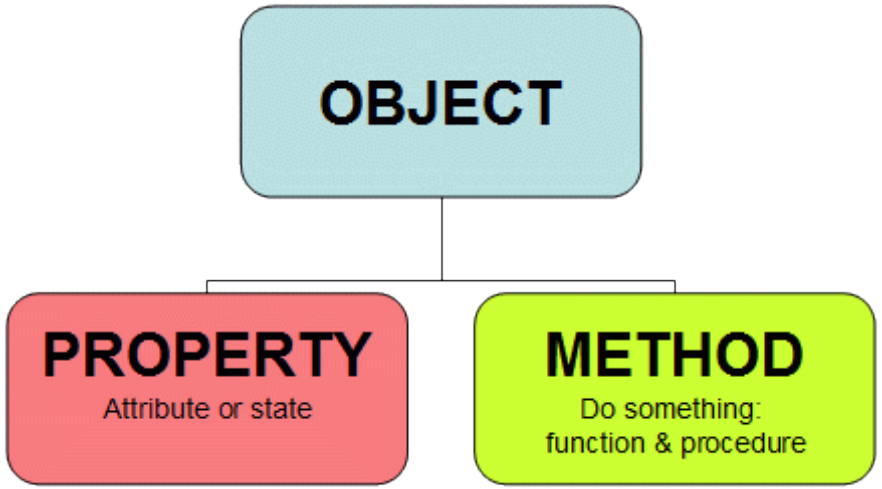
bmw

fuel

eyecolor

shake

fruit



http://www.teachitza.com/delphi/oop.htm

Review

- Stack
  - ✓Initializing array
  - ✓Assign the value(s)
  - ✓Get the value(s)

  (Further implementation on session 3 lab sessions)
- Array VS Linked List (*illustration*)

a[0] = 10 xsd01	a[1] = 20 xsd02	a[2] = 30 xsd03	a[3] = 40 xsd04	a[4] = 50 xsd05	xsd06
xsd07	xsd08	xsd09	xsd10	xsd11	xsd12
xsd13	xsd14	xsd15	xsd16	xsd17	xsd18

Array

a[5] = 10, 20, 30, 40, 50

Problem – not enough "space"

Add more array? Add more indexes (example: add 3 more)?

30 Next* = xsd14 xsd07	xsd08	10 Next* = xds16 xsd09	xsd10	xsd11	xsd12
xsd13	40 Next* = xsd17 xsd14	xsd15	20 Next* = xsd07 xsd16	50 Next* = NULL Xsd17	xsd18

Linked List

Use address (dynamic allocation)

Assign 5 values = 10, 20, 30, 40, 50

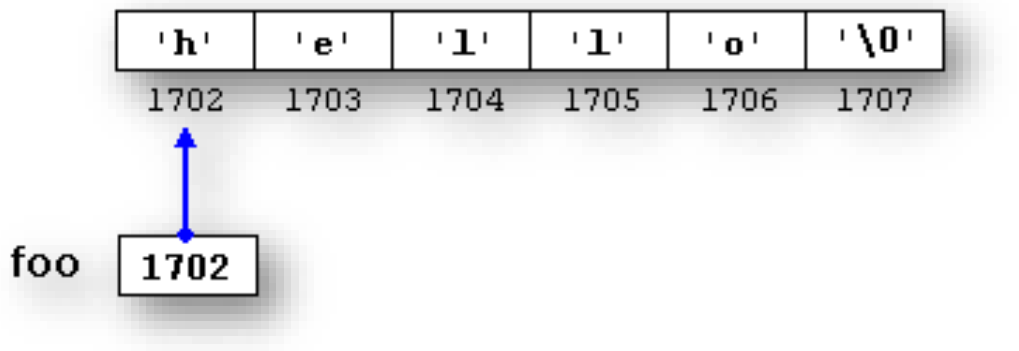
BINUS UNIVERSITY INTERNATIONAL

Pointers and Array Based Lists

PS: pointers are very useful if you want to create a linked list (week 5)

Pointers and Array Based Lists

Pointers – the “teaser” on lab sessions (week 1)



foo is a pointer and contains the value 1702

- The data types in C++ are classified into three categories: simple, structured, and pointers.

- The Pointer Data Type and Pointer Variables
  - The domain, (that is, the values of a pointer data type), consists of addresses (memory locations), a pointer variable is a variable whose content is an address, that is, a memory location.

http://www.cplusplus.com/doc/tutorial/pointers/

BINUS
UNIVERSITY
INTERNATIONAL

Declare and manipulate pointer variables

Declaring Pointer Variables

- The value of a pointer variable is an address.
- That is, the value refers to another memory space.
- The data is typically stored in this memory space.

- In C++, you declare a pointer variable by using the asterisk symbol (*) between the data type and the variable name.

```
dataType *identifier;
```

As an example, consider the following statements:

```
int *p;
char *ch;
```

both p and ch are pointer variables

https://www.ncl.ucar.edu/Document/HLUs/User_Guide/classes/classoview.shtml

Trivia

- declare p to be a pointer variable of type int

```
int *p;
int* p;
int * p;
```

Correct?

https://www.ncl.ucar.edu/Document/HLUs/User_Guide/classes/classoview.shtml

BINUS UNIVERSITY INTERNATIONAL

Question

To avoid confusion, we prefer to attach the character * to the variable name

```
int* p, q;
```

Which is/are the pointer variable(s)?

```
int *p, q;
```

Which is/are the pointer variable(s)?

```
int *p, *q;
```

Which is/are the pointer variable(s)?

BINUS
UNIVERSITY
INTERNATIONAL

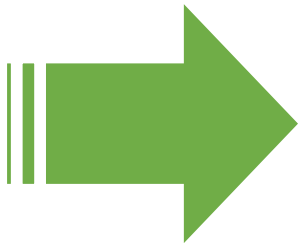
Use address of operator and dereferencing pointer

Address of Operator (&)

- C++ provides two operators—the address of operator (&) and the dereferencing operator (*)—to work with pointers.

- In C++, the ampersand, &, called the address of operator, is a unary operator that returns the address of its operand.

```
int x;
int *p;
```



```
p = &x;
```

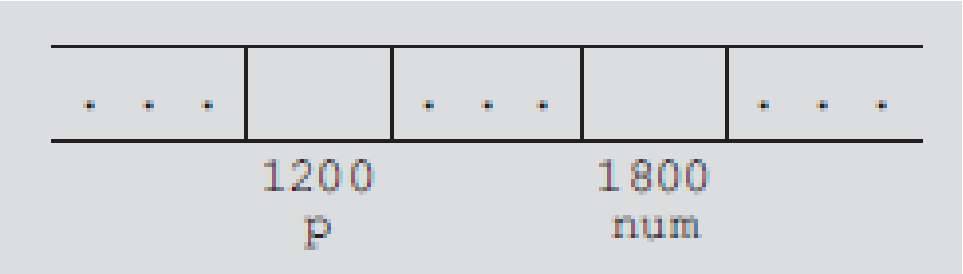
assigns the address of x to p. That is, x and the value of p refer to the same memory location

BINUS UNIVERSITY INTERNATIONAL

Dereferencing Operator (*)

- The previous chapters used the asterisk character, *, as the binary multiplication operator.
- C++ also uses * as a unary operator.

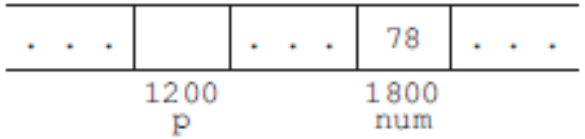
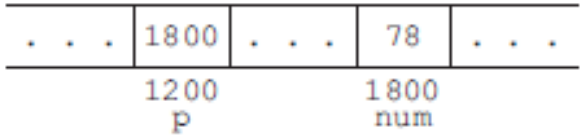
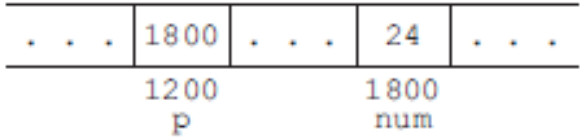
- When *, commonly referred to as the dereferencing operator or indirection operator, is used as a unary operator, * refers to the object to which the operand of the * (that is, the pointer) points.



Variables p and num

- Consider the following statements:

1. num = 78;
2. p = #
3. *p = 24;

After statement	Values of the variables	Explanation
1		The statement `num = 78;` stores 78 into num.
2		The statement `p = &num;` stores the address of num, which is 1800, into p.
3		The statement `*p = 24;` stores 24 into the memory location to which p points. Because the value of p is 1800, statement 3 stores 24 into memory location 1800. Note that the value of num is also changed.

BINUS UNIVERSITY INTERNATIONAL

Dereferencing Operator (*)

Example:

```
int x = 25;
int *p;
p = &x; //store the address of x in p
```

the statement

```
cout << *p << endl;
```

prints the value stored in the memory space to which p points, which is the value of x.

Also, the statement

```
*p = 55;
```

stores 55 in the memory location to which p points—that is, 55 is stored in x.

BINUS UNIVERSITY INTERNATIONAL

Exercise

- Results?

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int *p;
6      int num1 = 5;
7      int num2 = 8;
8      p = &num1; //store the address of num1 into p;
9      cout << "Line 9: &num1 = " << &num1 << ", p = " << p << endl; //Line 9
10     cout << "Line 10: num1 = " << num1  << ", *p = " << *p << endl; //Line 10
11     *p = 10;
12     cout << "Line 12: num1 = " << num1  << ", *p = " << *p << endl << endl; //Line 12
13     p = &num2; //store the address of num2 into p;
14     cout << "Line 14: &num2 = " << &num2 << ", p = " << p << endl; //Line 14
15     cout << "Line 15: num2 = " << num2 << ", *p = " << *p << endl; //Line 15
16     *p = 2 * (*p);
17     cout << "Line 17: num2 = " << num2 << ", *p = " << *p << endl; //Line 17
18     return 0;
19 }
```

int *p;

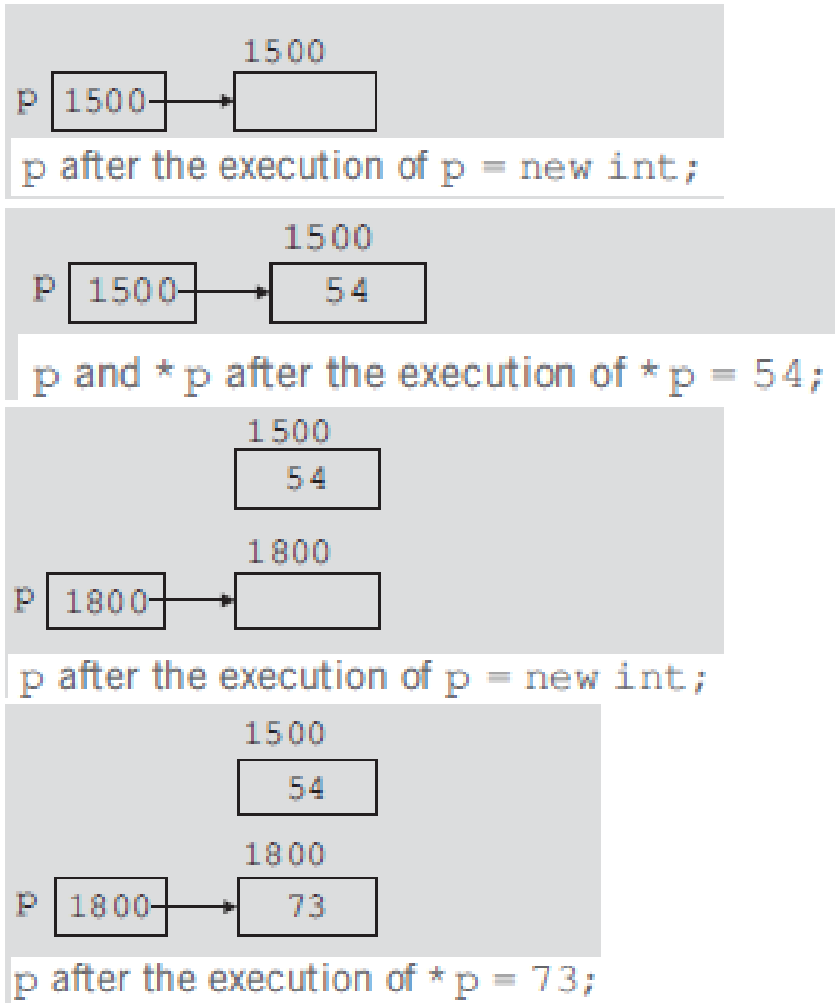
This statement declares p to be a pointer variable of type int. Next, consider the following statements:

p = new int; //Line 1

*p = 54; //Line 2

p = new int; //Line 3

*p = 73; //Line 4



BINUS UNIVERSITY INTERNATIONAL

Operator delete

How to avoid memory leak?
When a dynamic variable is no longer needed, it can be destroyed; that is, its memory can be deallocated

- memory leak = there is an unused memory space that cannot be allocated.

- The syntax to use the operator delete has the following two forms:

```
delete pointerVariable; //to deallocate a single dynamic variable
delete [] pointerVariable; //to deallocate a dynamic array
```

- Thus, given the declarations of the previous section, the statements:

```
delete p;
delete str;
```

- deallocate the memory spaces to which the pointers p and str point.
- Suppose p is a pointer variable, as declared previously. Note that an expression such as

```
delete p;
```

Operations on Pointer Variables

```
int *p;
double *q;
char *chPtr;
```

- Suppose that the size of the memory allocated for an int variable is 4 bytes, a double variable is 8 bytes, and a char variable is 1 byte.
- The statement

```
p++; or p = p + 1;
```

- increments the value of p by 4 bytes because p is a pointer of type int.

```
q++;
chPtr++;
```

- increment the value of q by 8 bytes and the value of chPtr by 1 byte, respectively.

```
p = p + 2;
```

- increments the value of p by 8 bytes.

BINUS
UNIVERSITY
INTERNATIONAL

Define abstract classes

Abstract classes

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm

abstract classes

- A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows –

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Implementation on lab sessions

https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm

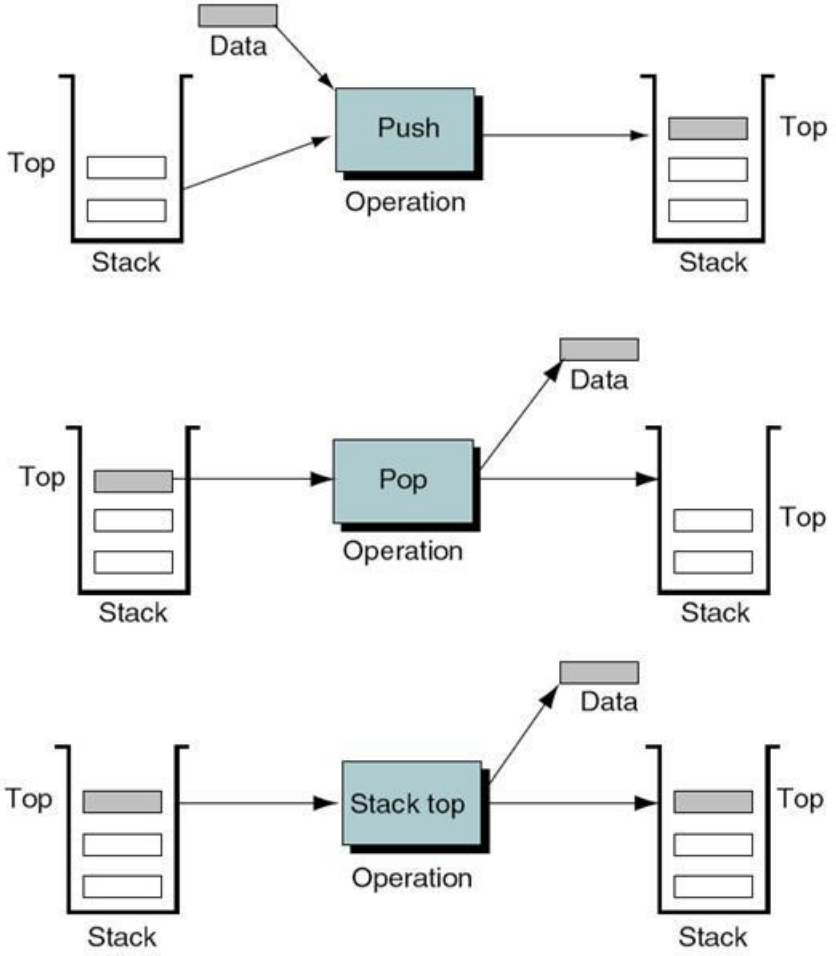
BINUS
UNIVERSITY
INTERNATIONAL

Stack

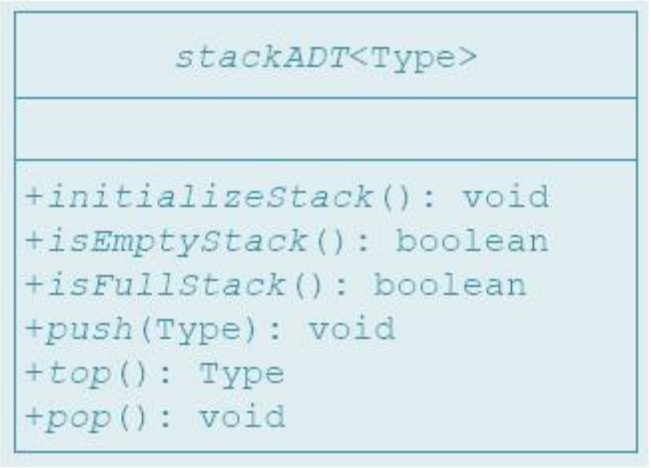
Recall - Stack

- Stack: A data structure in which the elements are added and removed from one end only (top); a Last In First Out (**LIFO**) data structure.

- Operation on Stack
  - **Push**: a new element is **added** to the stack.
  - **Pop**: top element is **removed** from the stack.
  - **top**: top element is **retrieved** of the stack.



A Stack ADT



- **initializeStack**—Initializes the stack to an empty state.
- **isEmptyStack**—Determines whether the stack is empty. If the stack is empty, it returns the value true; otherwise, it returns the value false.
- **isFullStack**—Determines whether the stack is full. If the stack is full, it returns the value true; otherwise, it returns the value false.
- **push**—Adds a new element to the top of the stack. The input to this operation consists of the stack and the new element. Prior to this operation, the stack must exist and must not be full.
- **top**—Returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty.
- **pop**—Removes the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

```
template <class Type>
class stackADT {
    virtual void initializeStack() = 0;
    virtual bool isEmptyStack() const = 0;
    virtual bool isFullStack() const = 0;
    virtual void push(const Type& newItem) = 0;
    virtual Type top() const = 0;
    virtual void pop() = 0;
};
```

Pure virtual function (abstract function) = no need to write any function definition. We have to declare it by assigning 0

Further reading: https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/

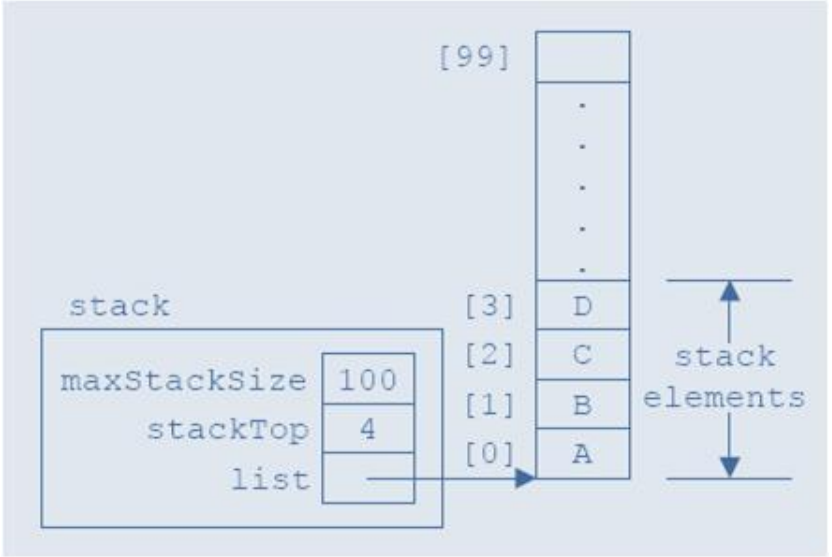
Array Implementation of a Stack

- Because all the elements of a stack are of the same type, you can use an array to implement a stack.
  - The first element of the stack can be put in the first array slot, the second element of the stack in the second array slot, and so on.
  - The top of the stack is the index of the last element added to the stack.
  - To keep track of the top position of the array, we can simply declare another variable, called stackTop.

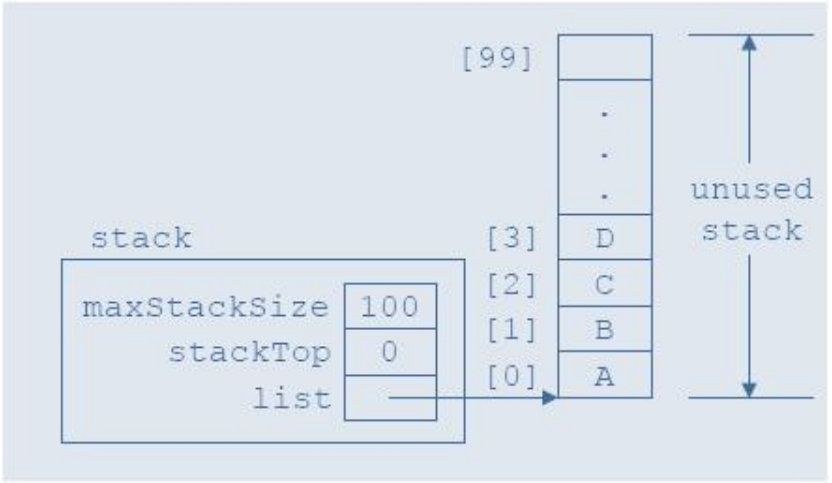
```
template <class Type>
class stackType: public stackADT<Type> {
    public:
        const stackType<Type>& operator=(const stackType<Type>&);
        void initializeStack();
        bool isEmptyStack() const;
        bool isFullStack() const;
        void push(const Type& newItem);
        Type top() const;
        void pop();
        stackType(int stackSize = 100);
        stackType(const stackType<Type>& otherStack);
        ~stackType();
    private:
        int maxStackSize;
        int stackTop;
        Type *list;
        void copyStack(const stackType<Type>& otherStack);
};
```

Array Implementation of a Stack

The pointer **list** contains the **base address of the array** (holding the stack elements)—that is, the address of the first array component



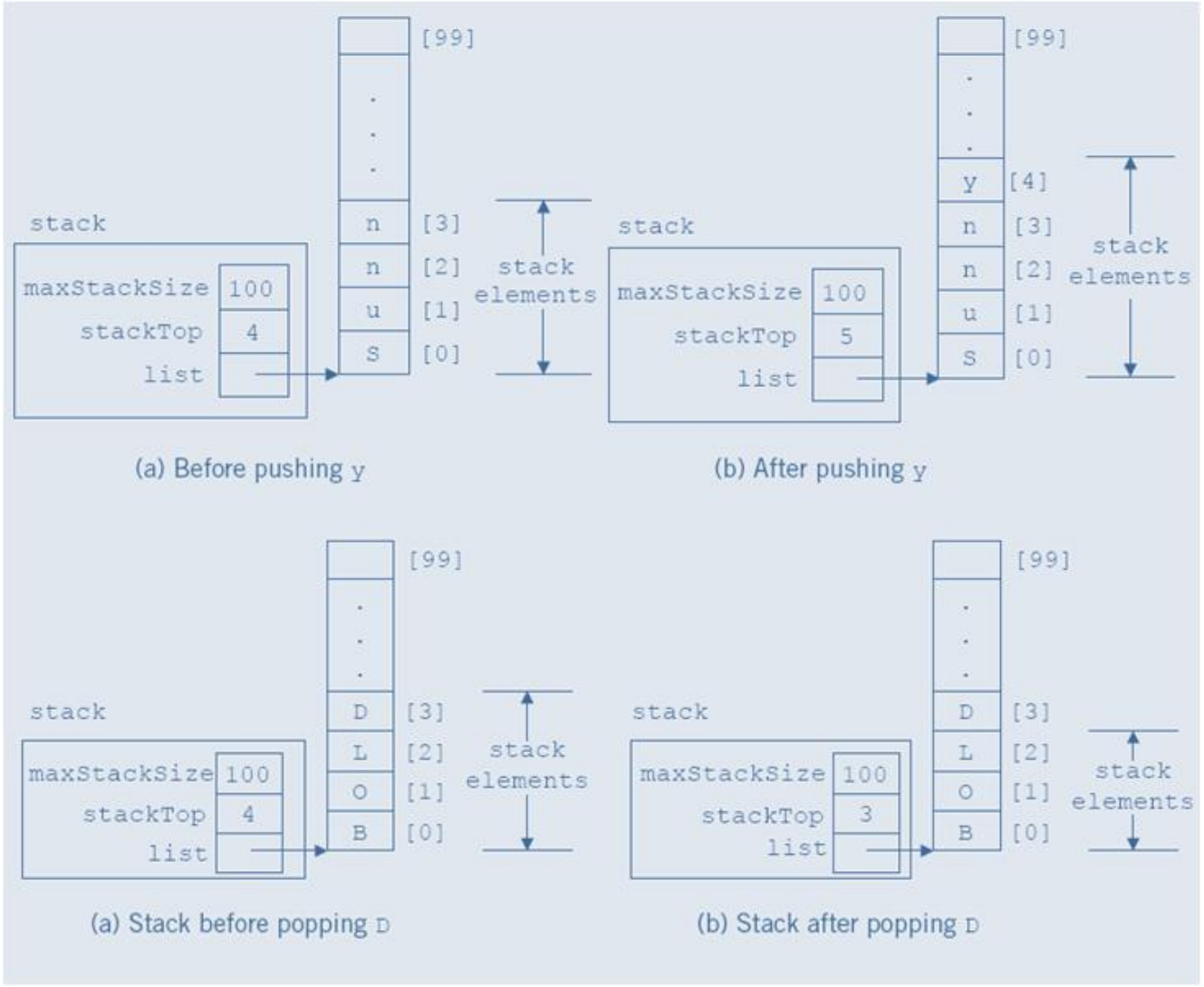
Because the value of stackTop indicates whether the stack is empty, we can simply set stackTop to 0 to initialize the stack.



Array Implementation of a Stack

The **push** operation is as follows:
1. Store the newItem in the array component indicated by stackTop.
2. Increment stackTop

To remove, or pop, an element from the stack, we simply **decrement stackTop by 1**.



Application of Stack

- Reversing a line of text
- Balancing Symbols
- Convert Decimal to Binary
- Infix to Postfix Conversion
- Postfix Expression

On lab session : STL and how to use the stack function

Algorithm Balancing Parenthesis

This algorithm reads a source program and parses it to make sure all opening-closing parentheses are paired.

```
while (more data)
read (character)
if (opening parenthesis)
    push(stack, character)
    else
    if (closing parenthesis)
        if (isEmpty(stack))
            print (Error: closing parenthesis not matched)
        else
            pop(stack)
        end if
    end if
end if
end while
```

BINUS UNIVERSITY INTERNATIONAL

Infix Notation

- The usual notation for writing arithmetic expressions is called **infix notation**, in which the operator is written between the operands.
  - For example, in the expression **a + b**, the operator **+** is between the operands **a** and **b**.
- In infix notation, the operators have precedence.
  - We must evaluate expressions from left to right, and multiplication and division have higher precedence than addition and subtraction.
  - If we want to evaluate the expression in a different order, we must include parentheses.
  - For example, in the expression **a + b * c**,
    - first evaluate * using the operands b and c, and
    - then we evaluate + using the operand a and the result of b * c.

Prefix and Postfix Notation

- Prefix Notation
  - In the early 1920s, the Polish mathematician Jan Lukasiewicz discovered that if operators were written before the operands (**prefix** or **Polish Notation**; for example, **+ a b**), the parentheses can be omitted.
- Postfix Notation
  - In the late 1950s, the Australian philosopher and early computer scientist Charles L. Hamblin proposed a scheme in which the operators follow the operands (**postfix** operators), resulting in the **Reverse Polish Notation (RPN)**, for example **a b +**
- RPN has the advantage that the **operators appear in the order required for computation**.

Infix to Postfix Conversion

operand **operator**

- **A + B * C - D / E** converts to **A B C * + D E / -**

1. Copy operand A to output expression.
2. Push operator + into stack.
3. Copy operand B to output expression.
4. Push operator * into stack. (Priority of * is higher than +)
5. Copy operand C to output expression.
6. Pop operator * and copy to output expression.
7. Pop operator + and copy to output expression.

Precedence Rules:

When a current operator is higher priority than the operator at the top of the stack, push it into the stack. Conversely, if the operator at the top of the stack has a higher priority than the current operator, it is popped and placed in the output expression.

When a current operator with a lower or equal priority forces the top operator to be popped from the stack, check the new top operator. If it is also greater than the current operator, it is popped to the output expression. Consequently, several operators may be popped to the output expression before pushing the new operator into the stack.

	Infix	Stack	Postfix
(a)	A+B*C–D/E		
(b)	+B*C–D/E		A
(c)	B*C–D/E	+	A
(d)	*C–D/E	+	AB
(e)	C–D/E	* +	AB
(f)	–D/E	* +	ABC
(g)	D/E	–	ABC*+
(h)	/E	–	ABC*+D
(i)	E	/ –	ABC*+D
(j)		/ –	ABC*+DE
(k)			ABC*+DE/–

Infix to Postfix Exercises (5 mins)

- Using the algorithm in the preceding slide, convert the following infix to postfix notation:
  - Higher priority: * /
  - Lower priority: + -

Infix expression	Equivalent postfix expression
A + B	
A + B * C	

Step	Infix string	Stack content	Postfix string
(a)			
(b)			
(c)			
....			

Infix to Postfix Exercises (10 mins)

- Using the algorithm in the preceding slide, convert the following infix to postfix notation:
  - Higher priority: * /
  - Lower priority: + -

Infix expression	Equivalent postfix expression
A * B + C	
(A + B) *C	
(A – B) * (C + D)	
(A + B) * (C – D / E) + F	

Postfix Evaluation

- Postfix form
  - Operator appears **after** the operands

Infix	Postfix
(4+3)*5	4 3 + 5 *
4+(3*5)	4 3 5 * +

  - No precedence rules or parentheses!
- Input expression given in postfix form
  - How to evaluate it?

Postfix Evaluation Steps

- Postfix expressions can be evaluated using the following algorithm:
  - Use a **stack,** assume binary operators +,*
  - Input: postfix expression
  - Scan the input
    - If operand,
      - **push** to stack
    - If operator
      - **pop** the stack twice
      - apply operator
      - **push** result back to stack

Thank you

Final project consultation

https://docs.google.com/spreadsheets/d/16CMSU7_3qU8NjYr-1wBx3KMNojSjOtwphqARWofclvM/edit?usp=sharing

BINUS
UNIVERSITY
INTERNATIONAL

FYI

BINUS UNIVERSITY INTERNATIONAL

Pointers and Classes

```
string *str;
str = new string;
*str = "Sunny Day";
```

The syntax for accessing a class (struct) member using the operator -> is as follows:

pointerVariableName->classMemberName

Thus, the expression

(*str).length()

is equivalent to the expression

str->length()

Classes and Pointers: Some Peculiarities

Because a class can have pointer member variables, this section discusses some peculiarities of such classes. To facilitate the discussion, we use the following class:

```
class pointerDataClass
{
public:
     .
     .
     .

private:
     int x;
     int lenP;
     int *p;
};
```

Also consider the following statements. (See Figure 3-20.)

```
pointerDataClass objectOne;
pointerDataClass objectTwo;
```

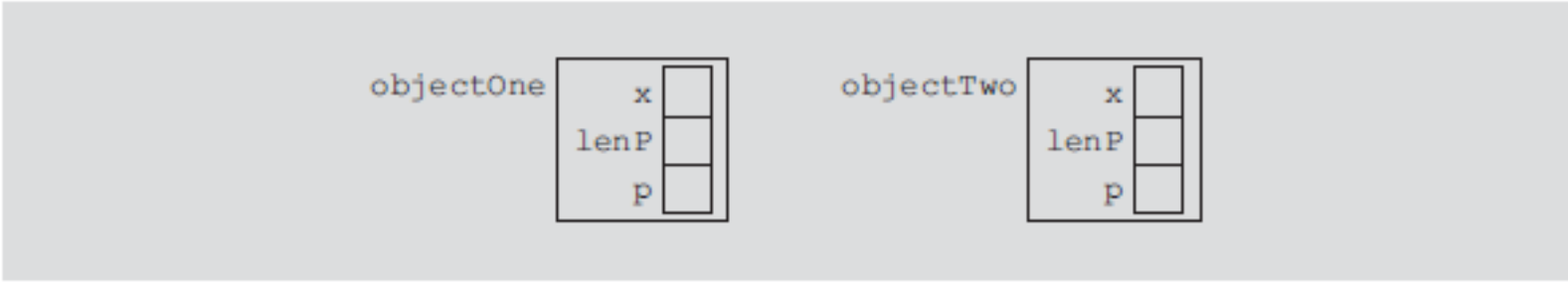


FIGURE 3-20 Objects `objectOne` and `objectTwo`

BINUS UNIVERSITY INTERNATIONAL

Classes and Virtual Destructors

One thing recommended for classes with pointer member variables is that these classes should have the destructor. The destructor is automatically executed when the class object goes out of scope. Thus, if the object creates dynamic objects, the destructor can be designed to deallocate the storage for them. If a derived class object is passed to a formal parameter of the base class type, the destructor of the base class executes regardless of whether the derived class object is passed by reference or by value. Logically, however, the destructor of the derived class should be executed when the derived class object goes out of scope.

To correct this problem, the destructor of the base class must be virtual. The **virtual destructor** of a base class automatically makes the destructor of a derived class virtual. When a derived class object is passed to a formal parameter of the base class type, then when the object goes out of scope, the destructor of the derived class executes. After executing the destructor of the derived class, the destructor of the base class executes.
Therefore, when the derived class object is destroyed, the base class part (that is, the members inherited from the base class) of the derived class object is also destroyed.

BINUS
UNIVERSITY
INTERNATIONAL

Use dynamic arrays
Explore dynamic arrays and lists

BINUS UNIVERSITY INTERNATIONAL

Dynamic Arrays

- An array created during the execution of a program is called a dynamic array.
- To create a dynamic array, we use the second form of the new operator.

The statement

```
int *p;
```

declares `p` to be a pointer variable of type `int`. The statement

```
p = new int[10];
```

allocates 10 contiguous memory locations, each of type `int`, and stores the address of the first memory location into `p`. In other words, the operator `new` creates an array of 10 components of type `int`, it returns the base address of the array, and the assignment operator stores the base address of the array into `p`. Thus, the statement

```
*p = 25;
```

stores `25` into the first memory location, and the statements

```
p++;        //p points to the next array component
*p = 35;
```

store `35` into the second memory location. Thus, by using the increment and decrement operations, you can access the components of the array. Of course, after performing a few increment operations, it is possible to lose track of the first array component. C++ allows us to use array notation to access these memory locations. For example, the statements

```
p[0] = 25;
p[1] = 35;
```

store `25` and `35` into the first and second array components, respectively. That is, `p[0]` refers to the first array component, `p[1]` refers to the second array component, and so on. In general, `p[i]` refers to the `(i + 1)`th array component. After the preceding statements execute, `p` still points to the first array component.

Constant Pointer

```
int list[5];
list[0] = 25;
list[2] = 78;
```