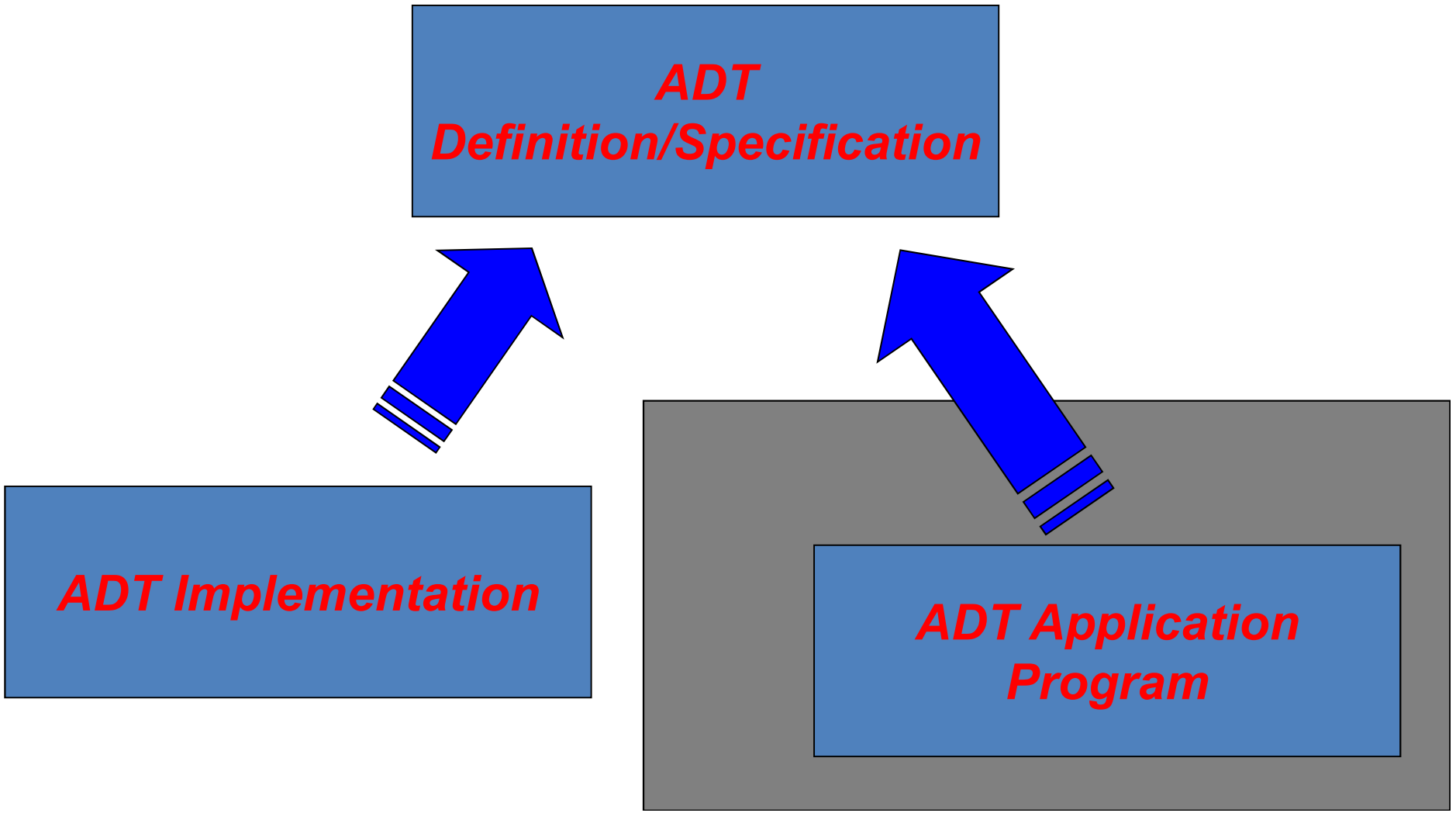
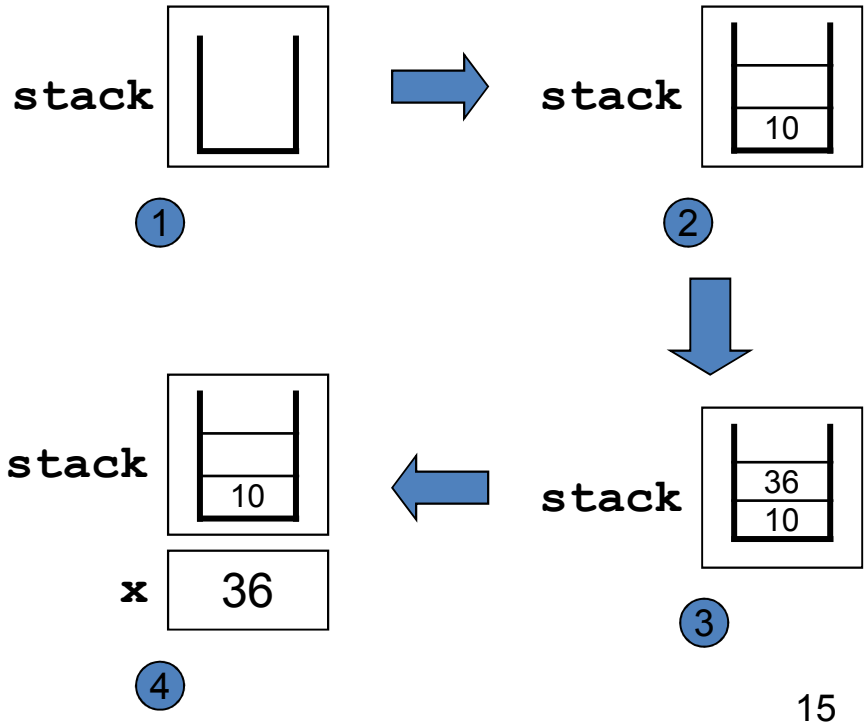
Stack as an ADT



Using the Stack ADT

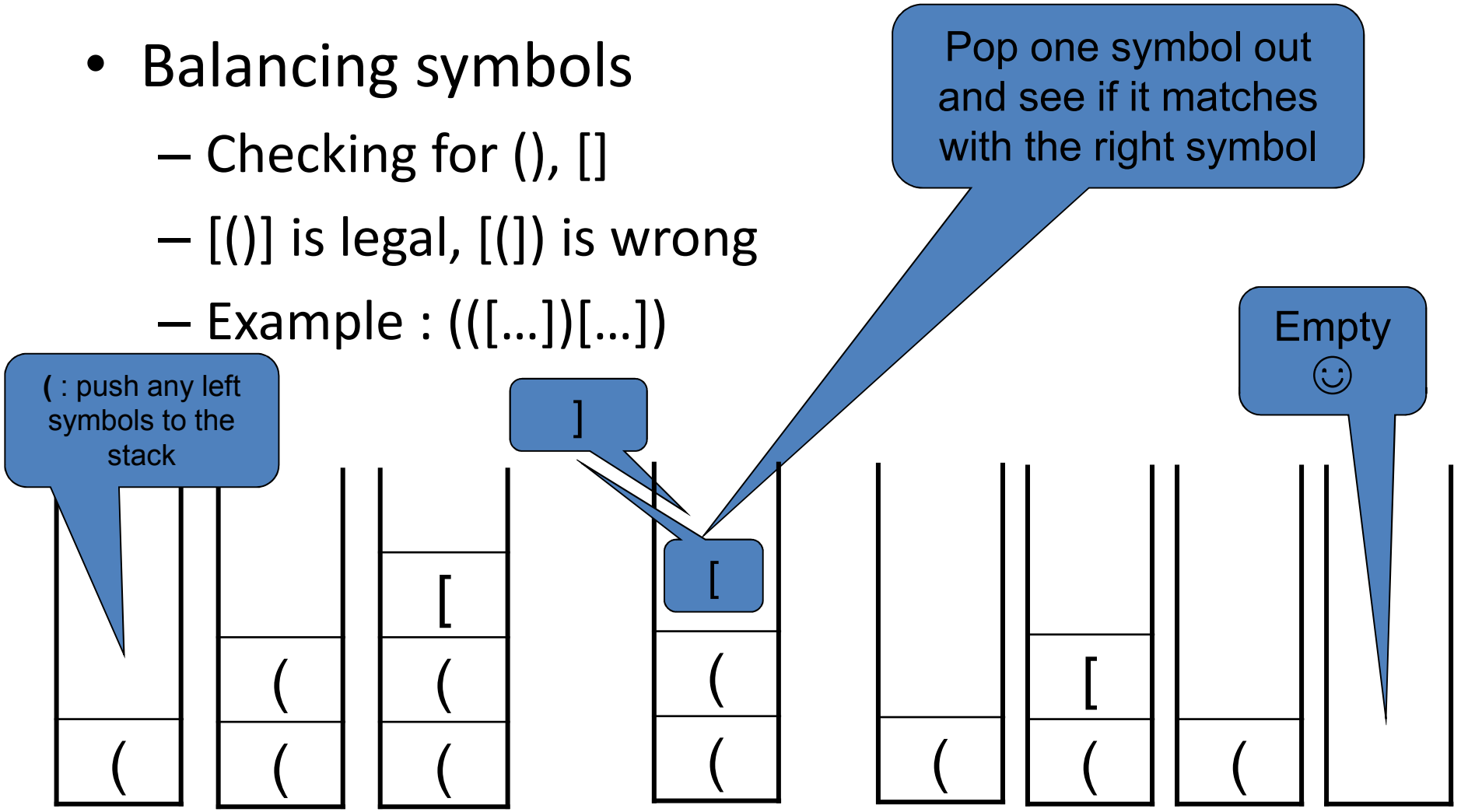
- Remember. To use an ADT, we do not even need to know its implementation.
- We just have to know what the data type *conceptually* is.

```
stackADT stack;
int x;
1  stack = EmptyStack();
2  Push(stack, 10);
3  Push(stack, 36);
4  x = Pop(stack);
.....
```



A Stack Application : balancing symbols

- Balancing symbols
  - Checking for (), []
  - [()] is legal, [(]) is wrong
  - Example : (([…])[…])



Balancing symbols

- Create an empty stack
- Read characters until end of file
  - If the character is an opening symbol, push it onto the stack
  - If the character is a closing symbol
    - If the stack is empty → error
    - Otherwise, pop the stack.
      - If the symbol popped is not the corresponding opening symbol → error
  - At the end of file
    - If stack is not empty → error

A Stack Application: Pocket Calculators

- Suppose we want to calculate

  50 * 15 + 38 / 19

- We want to enter the operations in the following special order.

  50 15 * 38 19 / +

- An expression in this form is called ***reverse Polish notation*** (RPN) or postfix.

a - b - c	a b - c -
a – (b – c)	a b c - -
(a * b) * c	a b * c *
a * (b * c)	a b c* *
a+b*c + (d*e+f) *g	a b c * + d e * f + g * +

Evaluating an RPN Expression

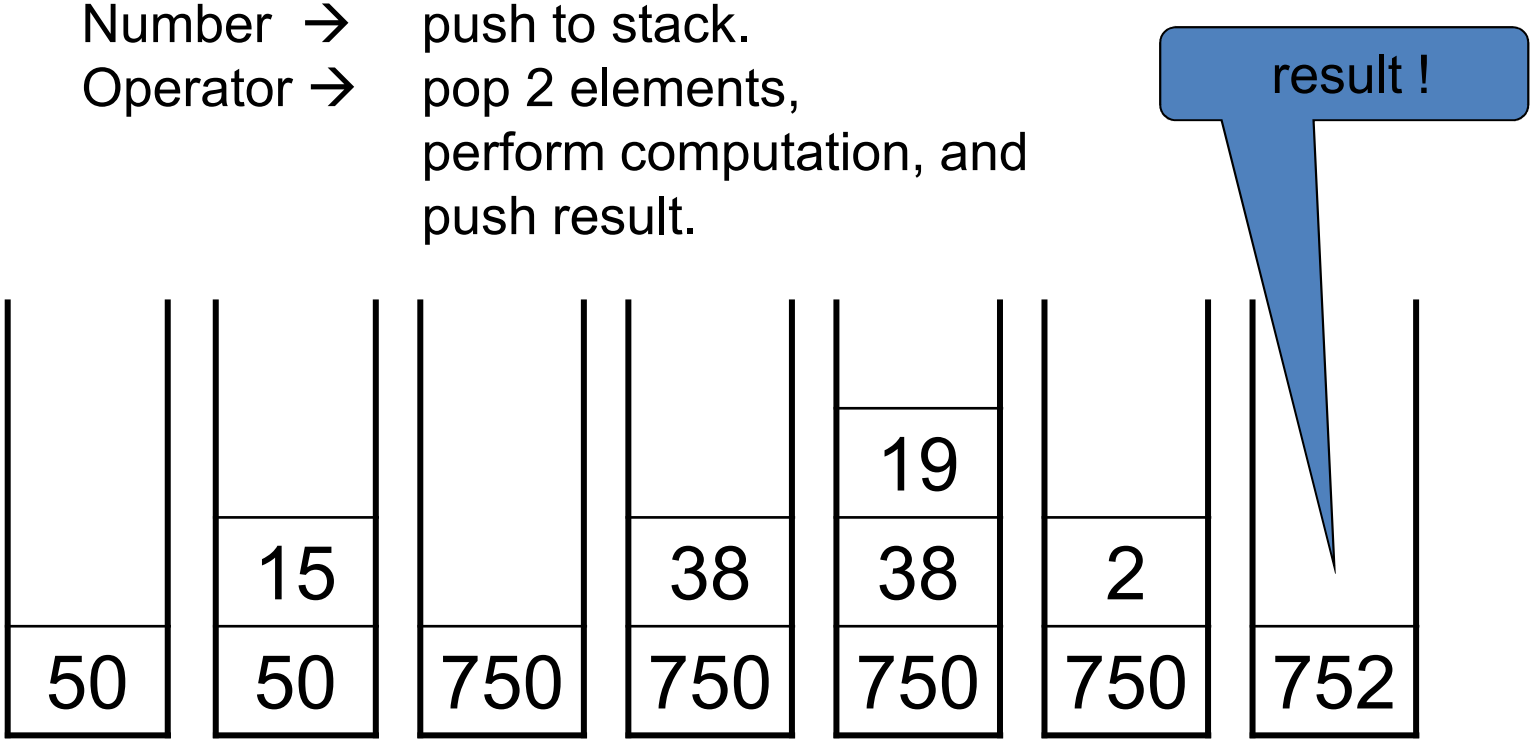
50 15 * 38 19 / + (50 * 15 + 38 / 19)

```
50↵
15↵
*↵
38↵
19↵
/↵
+↵
```

For each line of input:

Number → push to stack.

Operator → pop 2 elements, perform computation, and push result.



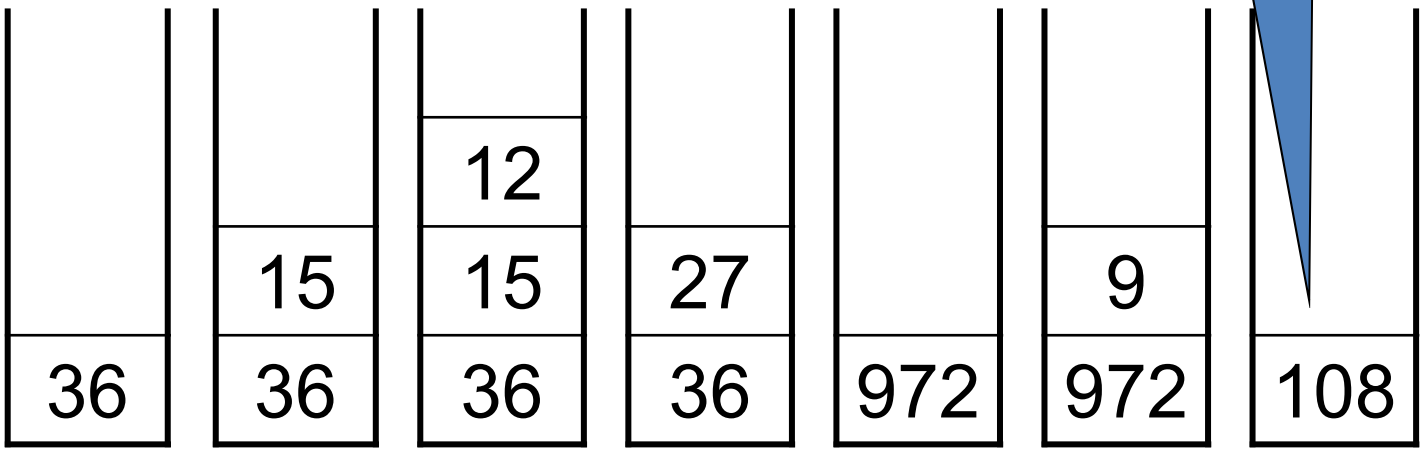
Evaluating an RPN Expression

36 15 12 + * 9 / (36 * (15 + 12) / 9)

```
36↵
15↵
12↵
+↵
*↵
9↵
/↵
```

For each line of input:

- Number → push to stack.
- Operator → pop 2 elements, perform computation, and push result.



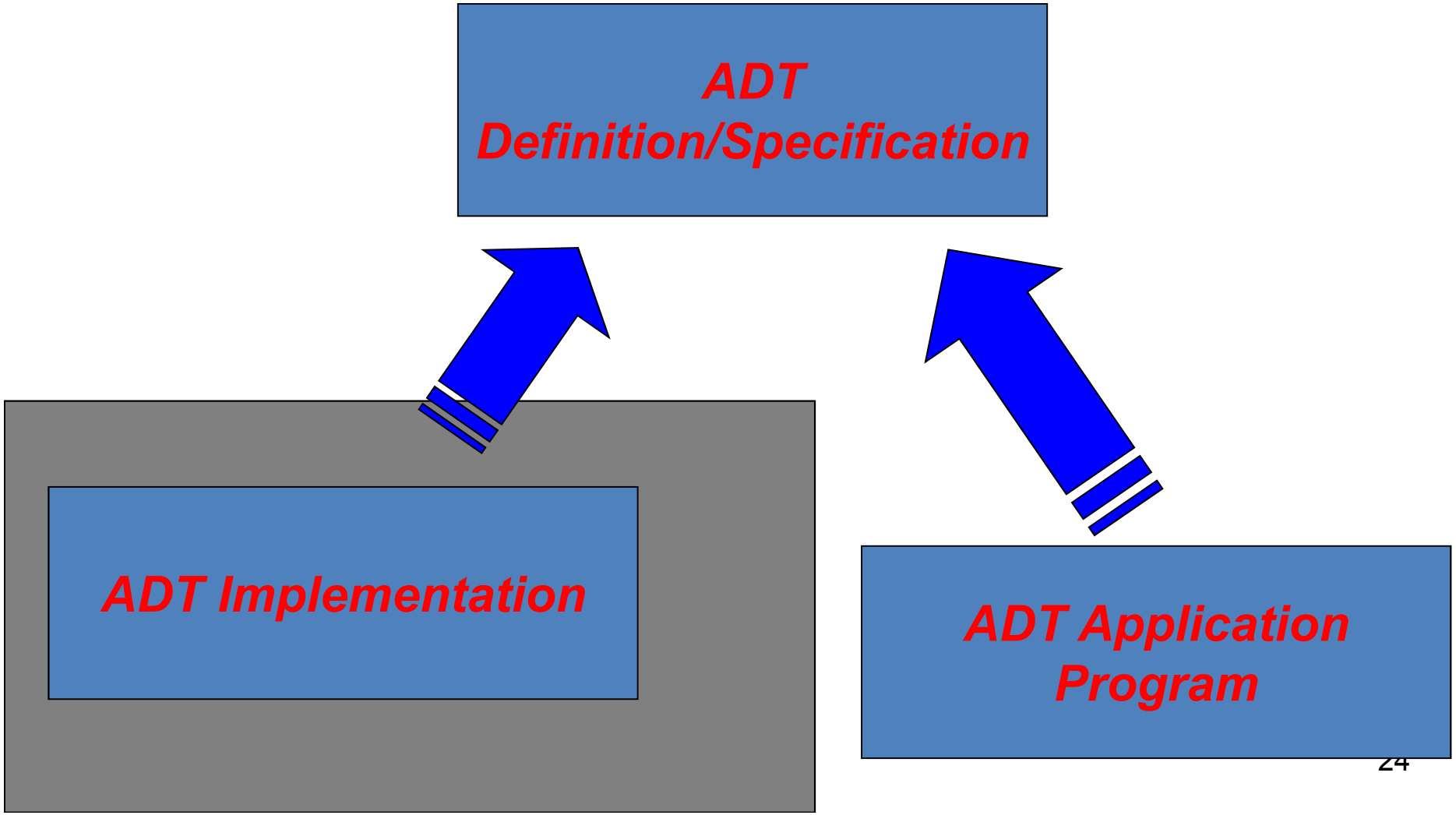
rpncalc.c

```
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>

void ApplyOperator(char c, stackADT stack);

int main() {
   char line[80];
   stackADT operandStack;
   operandStack = EmptyStack();
   do {
      scanf("%s", line);
      if (line[0] == '+' || line[0] == '-' ||
          line[0] == '*' || line[0] == '/') // operator
         ApplyOperator(line[0], operandStack);
      else if (line[0] != '=')                // number
         Push(operandStack, atoi(line));
   } while (line[0] != '=');  // '=' means end of input
   printf("%d\n", Pop(operandStack));
   return 0;
}
```

rpncalc.c (continue)

```
void ApplyOperator(char c, stackADT stack) {
   int x, y;

   // pop 2 elements
   y = Pop(stack);
   x = Pop(stack);

   // perform computation and push result
   if (c == '+')
      Push(stack, x + y);
   else if (c == '-')
      Push(stack, x - y);
   else if (c == '*')
      Push(stack, x * y);
   else if (c == '/')
      Push(stack, x / y);
}
```

Stack as an ADT



Implementing the Stack ADT

- To implement the stack ADT, we need to
  - choose a representation for a stack;
  - implement *all* the functions defined in stack.h.

stack.h

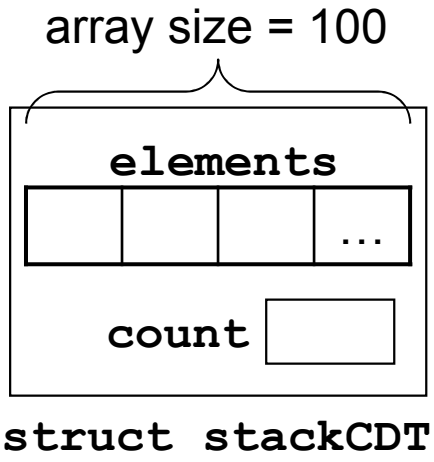
```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

- We shall place the implementation in the file stack.c.

Stack *Implementation* (Ver 1.0)

- The first implementation is to use an array to represent a stack.

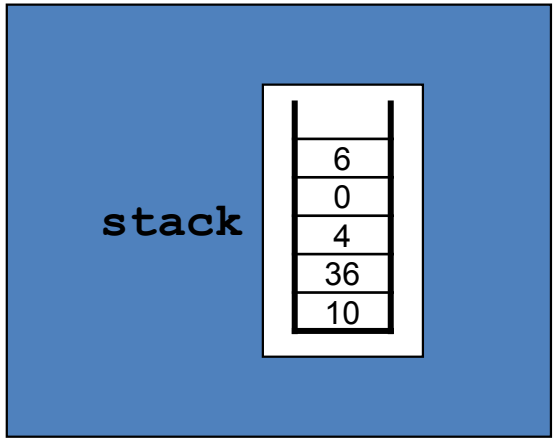
```
struct stackCDT {
   stackElementT elements[100];
   int count;
};
```

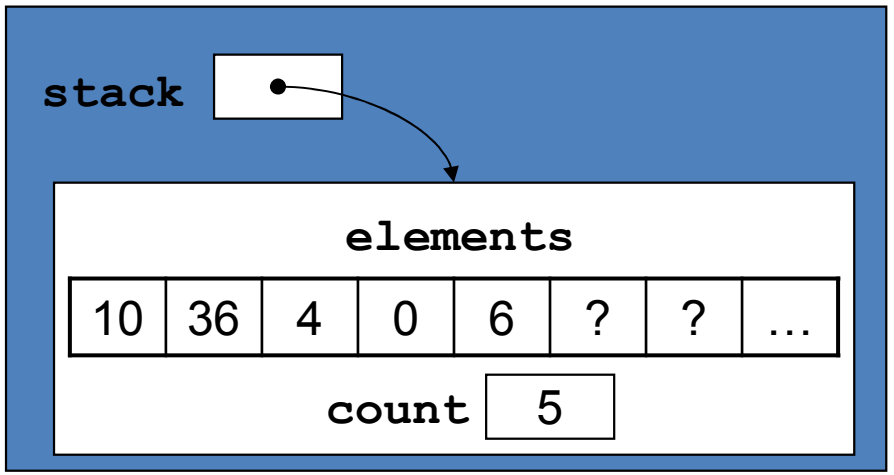


- This representation *limits* the maximum depth of a stack to be 100.

Stack *Implementation* (Ver 1.0)

- The elements in the stack are stored in the array **`elements`** from *bottom to top*.
- The member **`count`** stores the number of elements in the stack (i.e., the depth).



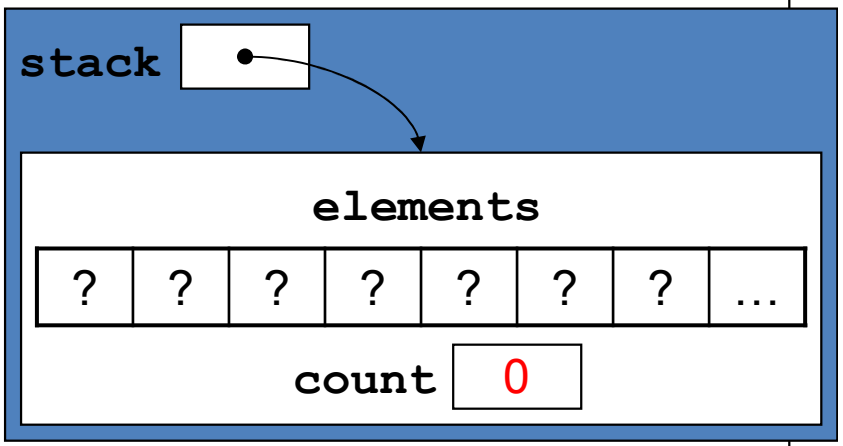


stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

stack.c

```
#include "stack.h"
#include <stdlib.h>

struct stackCDT {
   stackElementT elements[100];
   int count;
};

stackADT EmptyStack() {
   stackADT stack;
   stack = (stackADT)malloc(sizeof(struct stackCDT));
   stack->count = 0;
   return stack;
}
```

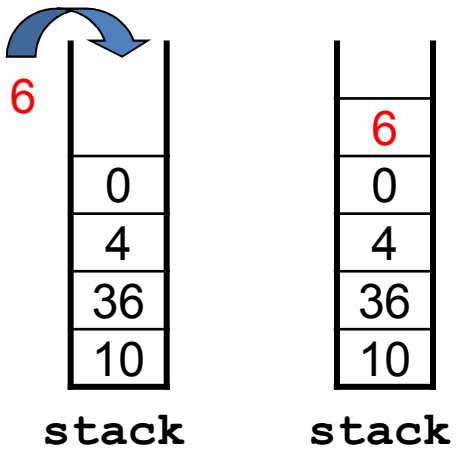


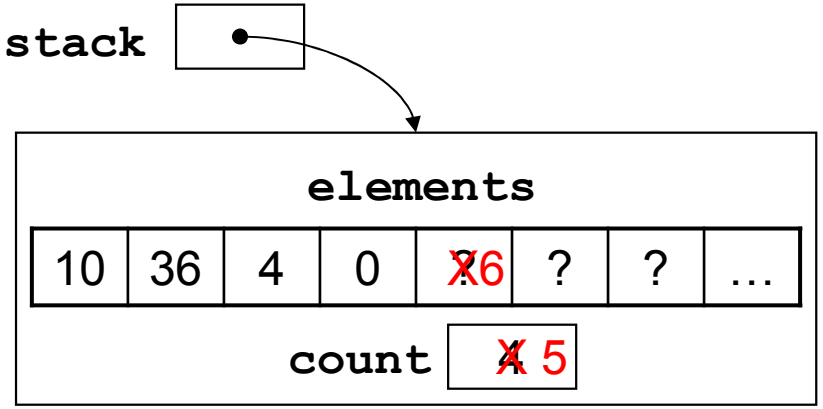
stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

stack.c (continue)

```
void Push(stackADT stack, stackElementT element) {
   stack->elements[stack->count] = element;
   (stack->count)++;
}
```



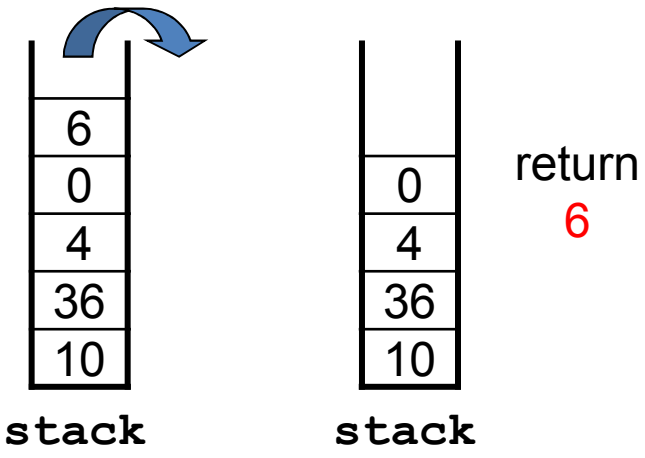


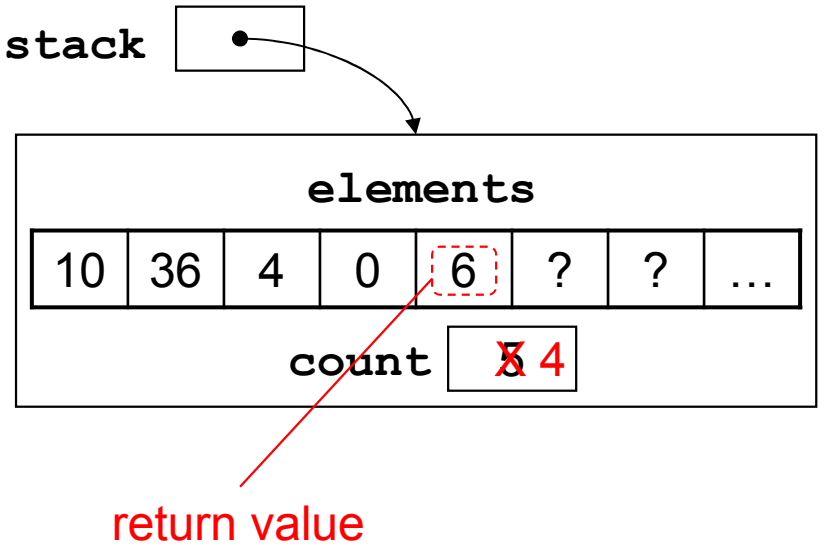
stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

stack.c (continue)

```
stackElementT Pop(stackADT stack) {
   (stack->count)--;
   return stack->elements[stack->count];
}
```





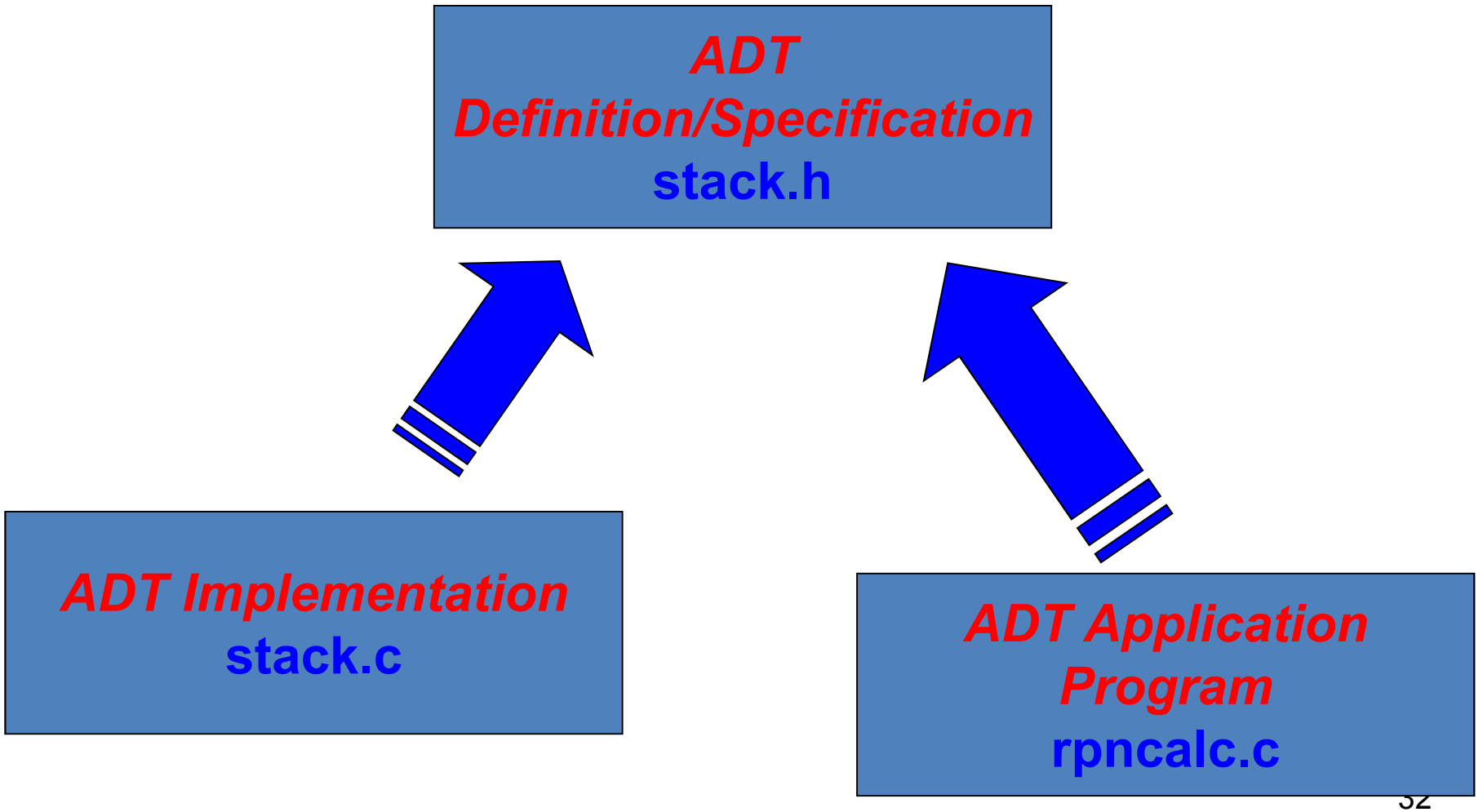
stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

stack.c (continue)

```
int StackDepth(stackADT stack) {
   return stack->count;
}

int StackIsEmpty(stackADT stack) {
   return (stack->count == 0);
}
```

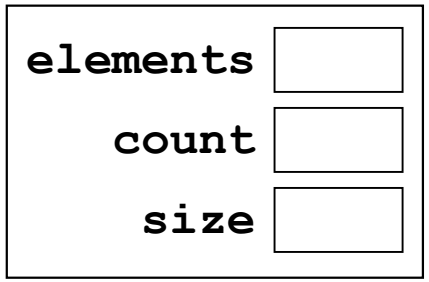
The Complete Program



Stack Implementation (Ver ***2.0***)

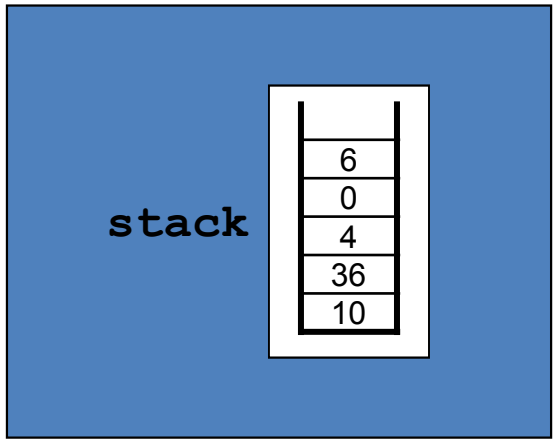
- In Version 1.0, when the stack depth is larger than 100, there will be array-index-out-of-bounds errors (***stack overflow***).
- We improve the implementation to give Version ***2.0***, using *dynamic arrays*.

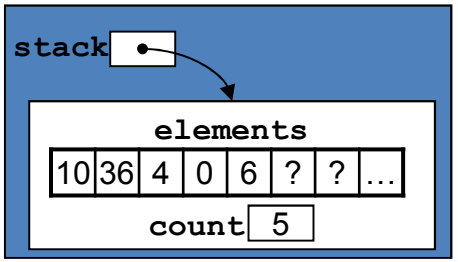
```
struct stackCDT {
   stackElementT *elements;
   int count;
   int size;
};
```



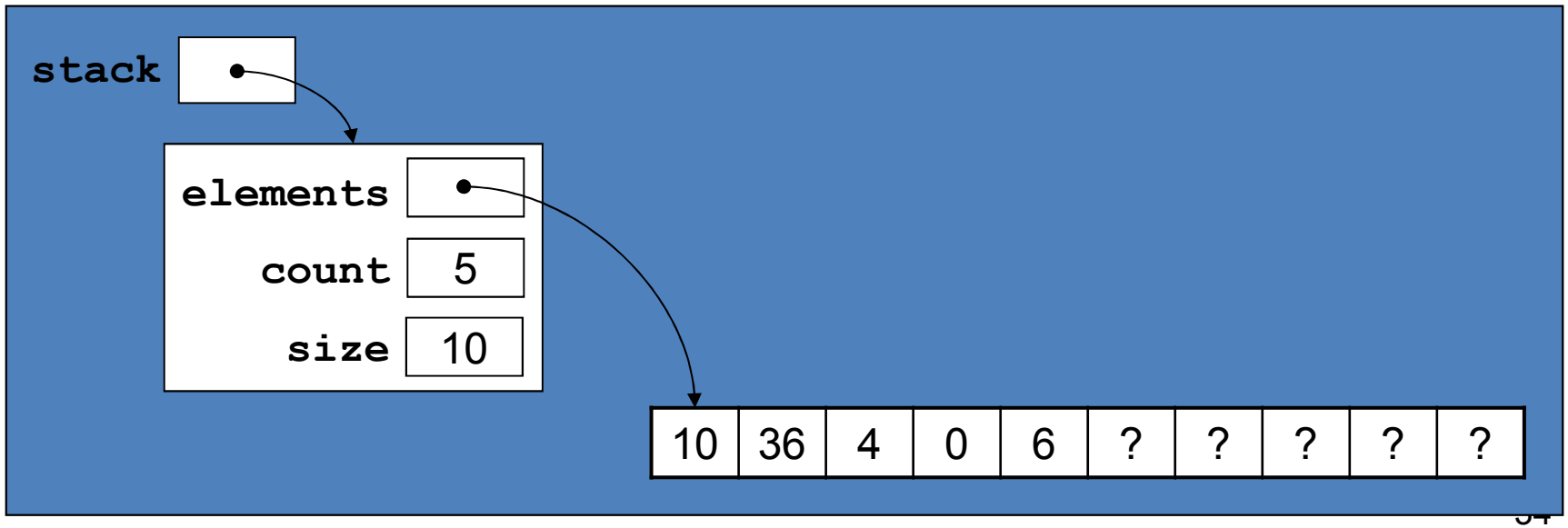
struct stackCDT

Stack Implementation (Ver ***2.0***)





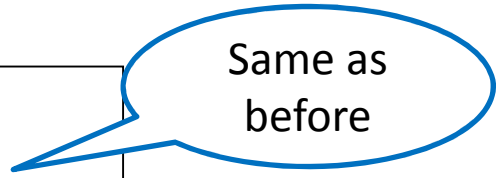
(Version 1.0)



Stack Implementation (Ver *2.0*)

stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```




```
stackADT EmptyStack() {
   stackADT stack;
   stack = (stackADT)malloc(sizeof(struct stackCDT));
   stack->elements = (stackElementT *)
                     malloc(10 * sizeof(stackElementT));
   stack->count = 0;
   stack->size = 10;
   return stack;
}
```

stack.c (Ver 1.0)

```
struct stackCDT {
   stackElementT elements[100];
   int count;
};
```

stack.c (Ver 2.0)

```
struct stackCDT {
   stackElementT *elements;
   int count;
   int size;
};
```

stack.c (Ver 1.0)

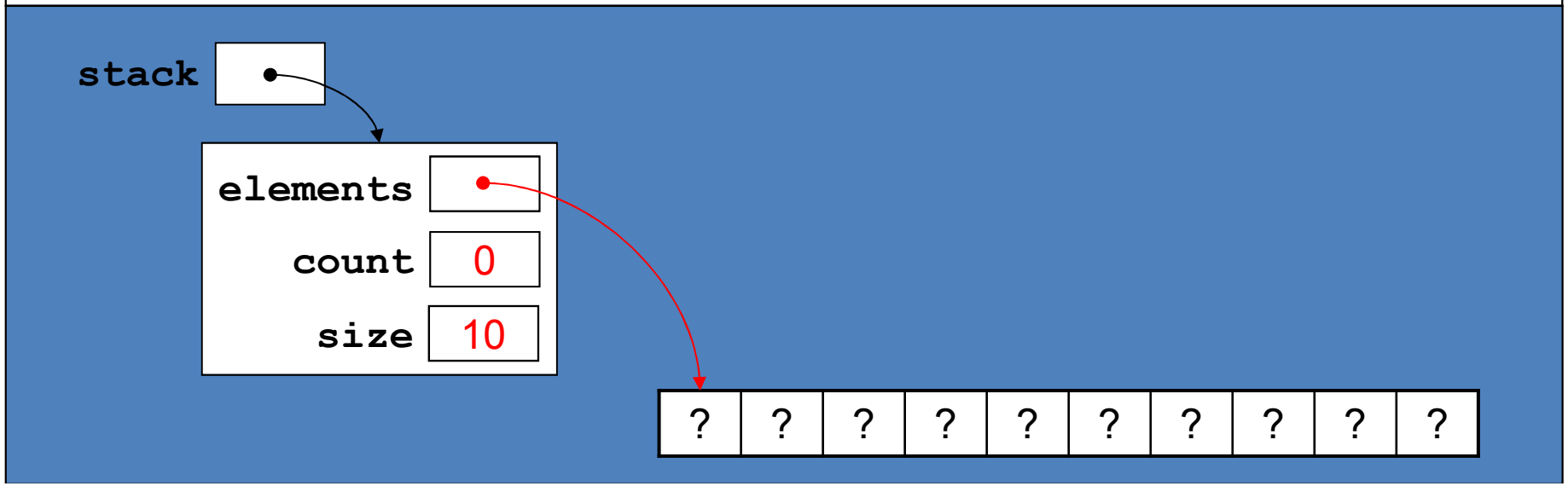
```
stackADT EmptyStack() {
   stackADT stack;
   stack = (stackADT)malloc(sizeof(struct stackCDT));
   stack->count = 0;
   return stack;
}
```

stack.c (Ver 2.0)

```
stackADT EmptyStack() {
   stackADT stack;
   stack = (stackADT)malloc(sizeof(struct stackCDT));
   stack->elements = (stackElementT *)
                     malloc(10 * sizeof(stackElementT));
   stack->count = 0;
   stack->size = 10;
   return stack;
}
```

Stack Implementation (Ver *2.0*)

```
stackADT EmptyStack() {
   stackADT stack;
   stack = (stackADT)malloc(sizeof(struct stackCDT));
   stack->elements = (stackElementT *)
                     malloc(10 * sizeof(stackElementT));
   stack->count = 0;
   stack->size = 10;
   return stack;
}
```



Stack Implementation (Ver *2.0*)

stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

```
void Push(stackADT stack, stackElementT element) {
   if (stack->count == stack->size) {
      stack->size += 10;
      stack->elements = (stackElementT *)realloc(
                        stack->elements,
                        stack->size * sizeof(stackElementT));
   }
   stack->elements[stack->count] = element;
   (stack->count)++;
}
```

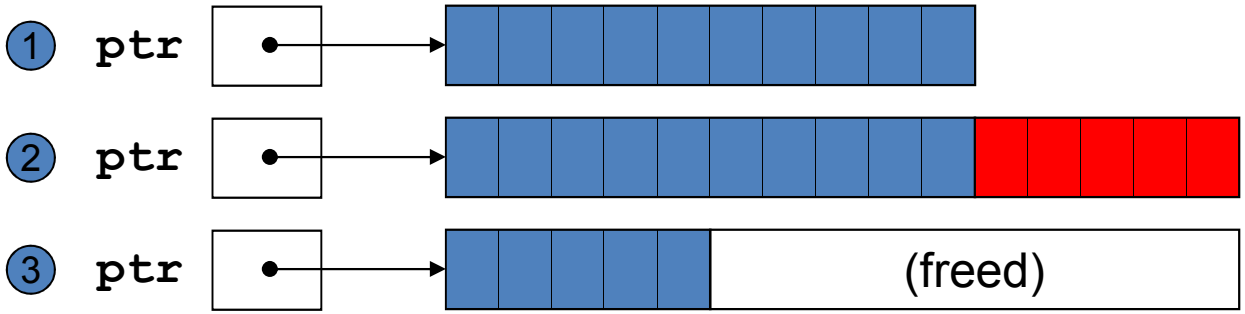
Memory Reallocation: `realloc()`

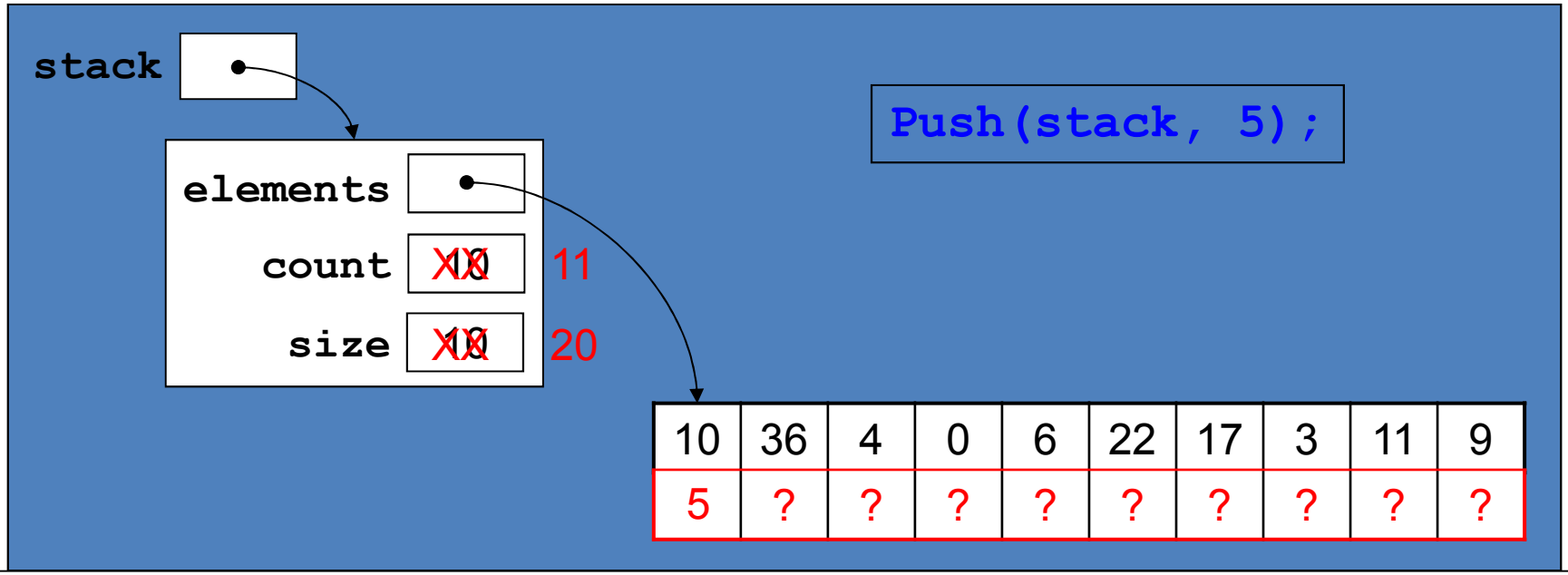
```
void *realloc(void *ptr,
              size_t size);
```

- Changes the size of the block pointed to by `ptr` to `size` bytes.
- Returns a pointer to the (possibly moved) block.
- The contents are unchanged up to the smaller of the new and old sizes.

realloc(): Example

```
#include <stdlib.h>

int main() {
    int *ptr;

① ptr = (int *)malloc(10 * sizeof(int));
    ……
② ptr = (int *)realloc(ptr, 15 * sizeof(int));
    ……
③ ptr = (int *)realloc(ptr, 5 * sizeof(int));
    ……
}
```






```
void Push(stackADT stack, stackElementT element) {
   if (stack->count == stack->size) {
      stack->size += 10;
      stack->elements = (stackElementT *)realloc(
                        stack->elements,
                        stack->size * sizeof(stackElementT));
   }
   stack->elements[stack->count] = element;
   (stack->count)++;
}
```

Stack Implementation (Ver *2.0*)

stack.h

```
typedef struct stackCDT *stackADT;

typedef int stackElementT;

stackADT EmptyStack();
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
int StackDepth(stackADT stack);
int StackIsEmpty(stackADT stack);
```

```
stackElementT Pop(stackADT stack) {
   (stack->count)--;
   return stack->elements[stack->count];
}

int StackDepth(stackADT stack) {
   return stack->count;
}

int StackIsEmpty(stackADT stack) {
   return (stack->count == 0);
}
```

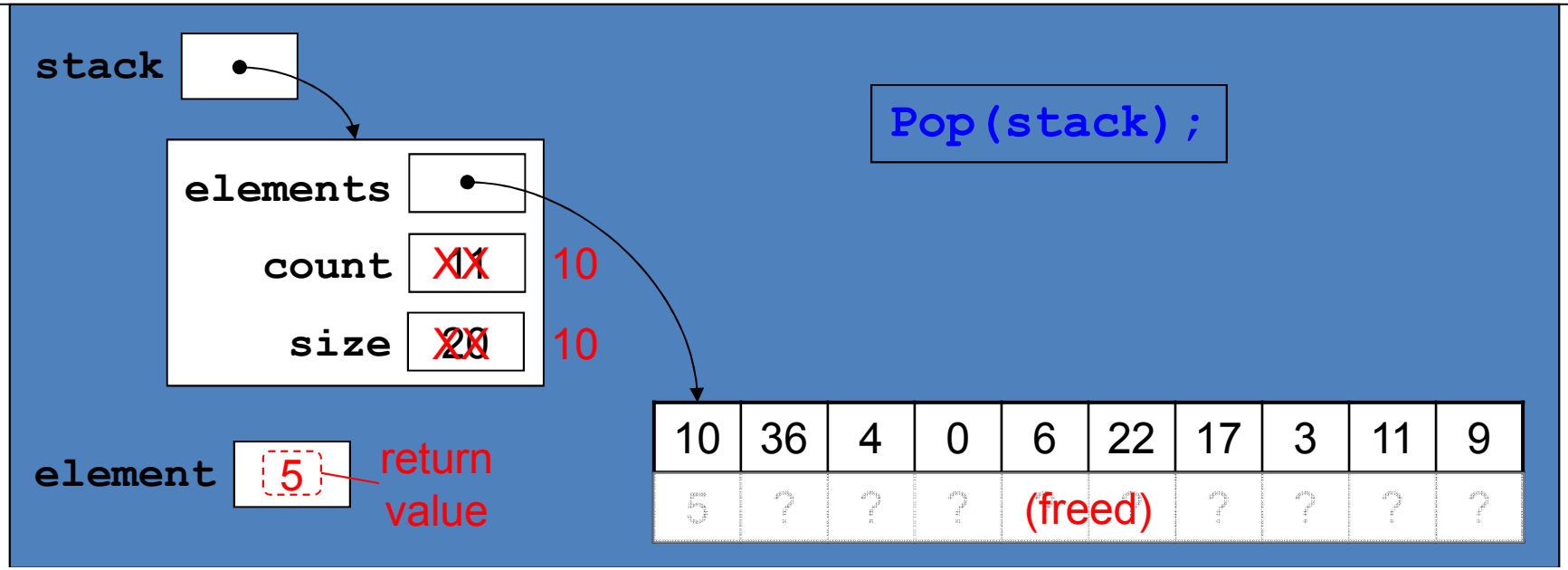
Same as Version 1.0.

Stack Implementation (Ver 2.*1*)

- One drawback of Version 2.0 is that the memory block size never decreases.
- We can improve the implementation by refining `Pop()` for more efficient memory usage.

```
stackElementT Pop(stackADT stack) {
   stackElementT element;
   (stack->count)--;
   element = stack->elements[stack->count];
   if (stack->count == stack->size - 10) {
      stack->size -= 10;
      stack->elements = (stackElementT *)realloc(
                        stack->elements,
                        stack->size * sizeof(stackElementT));
   }
   return element;
}
```

```
stackElementT Pop(stackADT stack) {
   stackElementT element;
   (stack->count)--;
   element = stack->elements[stack->count];
   if (stack->count == stack->size - 10) {
      stack->size -= 10;
      stack->elements = (stackElementT *)realloc(
                        stack->elements,
                        stack->size * sizeof(stackElementT));
   }
   return element;
}
```



Stack Implementation (Ver 2.***2***)

- Popping from an empty stack causes errors (***stack underflow***) too. Version 2.***2*** handles this.

```
stackElementT Pop(stackADT stack) {
   stackElementT element;
   if (StackIsEmpty(stack))
      exit(0);
   (stack->count)--;
   element = stack->elements[stack->count];
   if (stack->count == stack->size - 10) {
      stack->size -= 10;
      stack->elements = (stackElementT *)realloc(
                        stack->elements,
                        stack->size * sizeof(stackElementT));
   }
   return element;
}
```

The Complete Program