**Data Structures
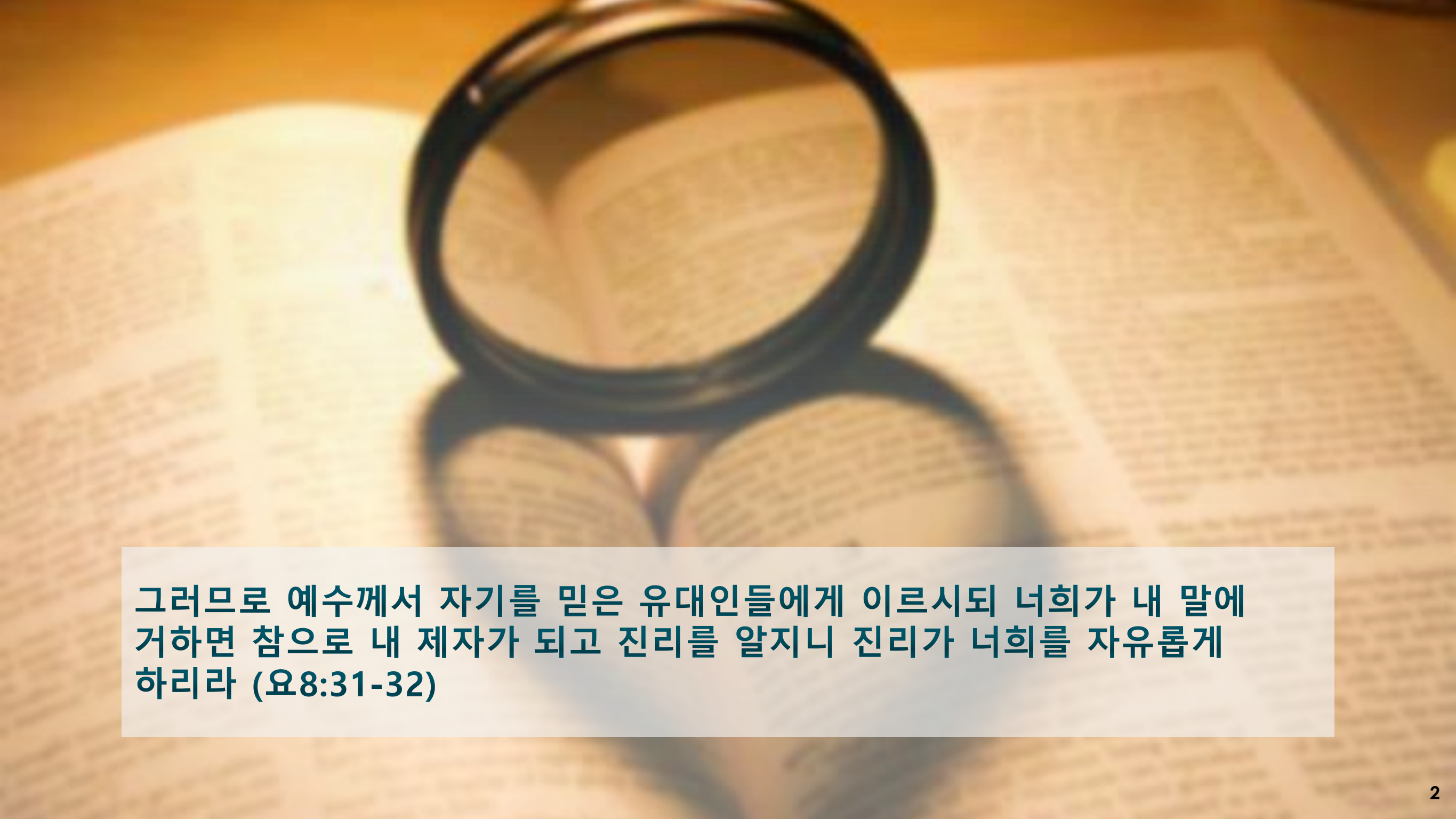Chapter 3**

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - Using Dynamic Array
     - Using Vector
     - Using Template
2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

그러므로 예수께서 자기를 믿은 유대인들에게 이르시되 너희가 내 말에 거하면 참으로 내 제자가 되고 진리를 알지니 진리가 너희를 자유롭게 하리라 (요8:31-32)

Stack

- **Stack** is a linear data structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.
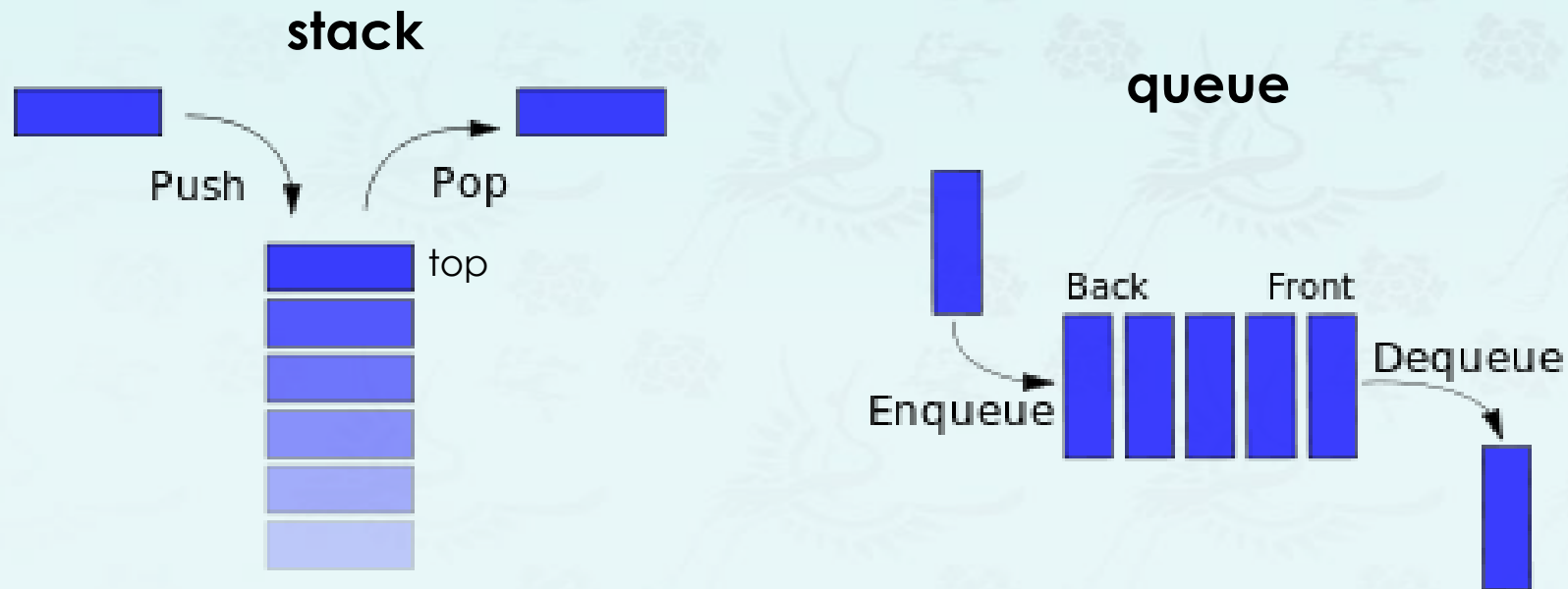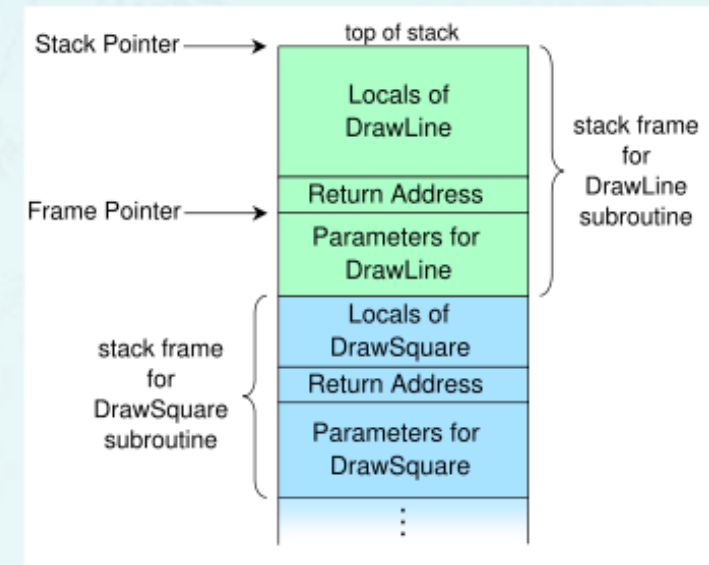
Stack

- **Stack** is a linear data structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.
- This particular order of the operation is called **LIFO(Last In First Out).**

stack

Push   Pop

top

queue

Back   Front

Enqueue

Dequeue

FIFO = "First in First out"
Remove the item
least recently added.

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

4

Stack Applications

- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Backtracking as in a maze
- Implementing function calls in a compiler.
- ...
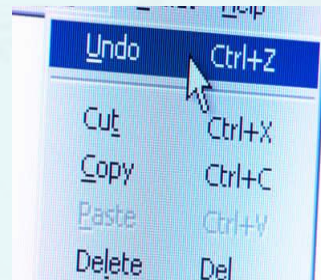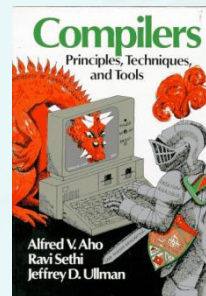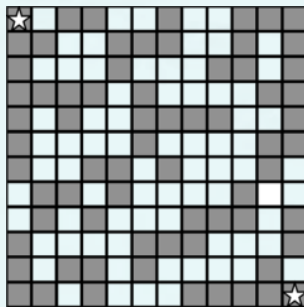
Stack – ADT (Abstract Data Type)

ADT Stack is

- **Objects**: a finite ordered list with zero or more elements

- **Operations (or Functions)**:
    ```
    Stack newStack(maxStackSize)
    bool empty()
    void push(item)
    void pop()
    int top()
    int size;
    ```

Stack - Why ADT?

- **Separate interface and implementation.**
  - Ex: stack, queue, bag, priority queue, symbol table, union-find, ...

- **Benefits.**
  - **Driver (or Client)** can't know details of implementation
    - Client has many implementations from which to choose.
    - Program using operations defined in **interface.**
  - **Interface** is description of data type, basic operations.
  - **Implementation** is actual code implementing operations.
    - Design: creates modular, reusable libraries.
    - Performance: use optimized implementation where it matters.

Stack: Example in C++

STL	**#include <stack>**	Stack class in C++ STL
	stack<value_type>	*creates an empty stack of <value_type>*
void	push(const value_type& item)	*inserts a new item onto stack*
void	pop()	*removes top item from stack (which is most recently added)*
value_type&	top()	*returns a reference to the top item*
bool	empty()	*is the stack empty?*
int	size()	*returns the number of items in the stack*

https://cplusplus.com/reference/stack/stack/

Warm-up client: Reverse sequence of strings using stack.

Stack: Driver/Client using stack class in C++ STL

- Read strings from a collection using a range-for loop.
  - If string equals "-", pop string from stack and print.
  - Otherwise, push string onto stack.

```cpp
int main () { // stack initialization using range-based for
  string list[] = {"to", "be", "or", "not", "to", "-", "be",
                   "-", "-", "that", "-", "-", "-", "is"};
  stack<string> s;
  for (auto item : list) {   // to be not that or be
    if (item != "-")
      s.push(item);
    else {
      cout << s.top() << ' ';
      s.pop();
    }
  }
  cout << "\nsize(): " << s.size();
  cout << "\ntop() : " << s.top();
}
```

```cpp
void printStack(stack<string> s) {
  while (!s.empty())  {
    cout << s.top() << ' ';
    s.pop();
  }
  cout << endl; // now, s is empty
}
```

Stack: Implementation

Let's implement our own stack in several different ways.

- Array implementation
  - fixed size array
  - dynamic array
- Vector implementation
- Using Template
  - Array implementation
  - Vector implementation

Stack: Array implementation

Let's implement our own stack in several different ways.

- Array implementation of a stack:
  - Use array `s[ ]` to store `N` items on stack.
  - `push():` add new item at `s[N].`
  - `top()`: return item from `s[N-1].`
  - `pop()`: remove item from `s[N-1]`, it just decrements `N` by one.

s[ ]	to	be	or	not	to	be	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

N   capacity = 10

Defect. Stack overflows when N exceeds capacity. *[stay tuned]*

Stack: Things to consider

- ## Underflow:
  - Throw exception if pop from an empty stack or return null;

```
string top(stack s) {
  return s->item[s->N - 1];
} ;
```

```
string top(stack s) {
  if (empty())
      throw std::out_of_range("underflow");

  return s->item[s->N - 1];
}
```

- ## Overflow:
  - Use resizing array for array implementation. [stay tuned]
  - Use successive doubling method
- ## Generic programming using Template in C++
  - It makes the stack data(item) type-independent
  - **template<typename T>**

```
struct Stack {
  string *item;
  int N;
  int capacity;
};
using stack = Stack *;

stack newStack(int capacity) {
  stack s = new Stack;
  s->item = new string[capacity];
  s->N = 0;
  s->capacity = capacity;
  return s;
}
void free(stack s) {
  delete[] s->item;
  delete s;
}
```

a shortcoming
(stay tuned)

item[N] is next to be filled if any.

```
int size(stack s)    { return s->N; }

bool empty(stack s) { return s->N == 0; }

void pop(stack s)    { s->N--; }

string top(stack s) {
  return s->item[s->N - 1];
}
void push(stack s, string item) {
  s->item[s->N++] = item;
}
void printStack(stack s) {
  while (!empty(s))  {
    cout << top(s) << ' ';
    pop(s);
  }
  cout << endl;  // stack is empty now
}
```

N is not decremented

use N and incremented
N points an empty slot

13

Data Structures
Chapter 3

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - **Using Dynamic Array**
     - Using Vector
     - Using STL Template
2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

**Data Structures
Chapter 3**

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - **Using Dynamic Array**
     - Using Vector
     - Using STL Template
2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Stack: Using dynamic arrays

- Problem:
  - Requiring client to provide capacity (size of stack) is inappropriate.
  - Question: How to grow and shrink array?

- First try.
  - **push()**: increase size of array **s[]** by 1.
  - **pop()**: decrease size of array **s[]** by 1.

- **Too expensive.**
  - Need to copy all items to a new array.
  - Inserting first N items takes time proportional to $1 + 2 + 3 + \ldots + N \approx N^2/2$.

infeasible for large N

Challenge: Ensure that array resizing happens infrequently.

Stack: Using dynamic arrays

Q. How to grow and shrink array?

"successive doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
stack newStack(int capacity = 1) {
  stack s = new Stack;
  s->item = new string[capacity];
  s->capacity = capacity;
  s->N = 0;
  return s;
}

void resize(stack s, int new_capacity) {
  string *copied = new string[new_capacity];
  for (int i = 0; i < s->N; i++)
    copied[i] = s->item[i];
  delete[] s->item;
  s->item = copied;
  s->capacity = new_capacity;
}
```

```
copy(s->item; s->item + s->N, copied);
```

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(s.length + 10)?

- **A.** $T(N) = 10N + (10 + 20 + 30 + \ldots + N)$

        ↑   ↑

1 array access per push   k array accesses when memory is resized by increment of 10
(ignoring cost to create new array)
(assuming new() costs copying each item one by one)

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(s.length + 10)?
- Let us consider an example of a simple array stack pushes.

push 1~10

size=capa

1..10

push 11~20

size=capa

1..10	11..20

push 21~30

size=capa

1..10	11..20	21..30

push 31~40

size=capa

1..10	11..20	21..30	31..40

push 41~50

1..10	11..20	21..30	31..40	41..50

push N

1..10	11..20	21..30	31..40	41..50	...	(N-10)+1..N

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(s.length + 10)?
- Let us consider an example of a simple array stack pushes.

push 1~10
size=capa

1..10

push 11~20
size=capa

1..10	11..20

push 21~30
size=capa

1..10	11..20	21..30

push 31~40
size=capa

1..10	11..20	21..30	31..40

push 41~50

1..10	11..20	21..30	31..40	41..50

push N

1..10	11..20	21..30	31..40	41..50	...	(N-10)+1..N

push(N)	1~10	11~20	21~30	31~40	41~50		(N-10)+1 ~ N
stack capacity	10	20	30	40	50	...	N
time T(N) push+copy	10+0	10+10	10+20	10+30	10+40		10+(N-10)

```
T(N) = (10) + (10+10) + (10+20) + (10+30) + ... + (10+N-10)
     =         (10+10) + (10+20) + (10+30) + ... + (10+N)
     = (10 + 10 + 10 + ...) + (10 + 20 + 30 + ... + N)
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

20

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(s.length + 10)?
- **A.** $T(N) = 10N + (10 + 20 + 30 + \ldots + N)$

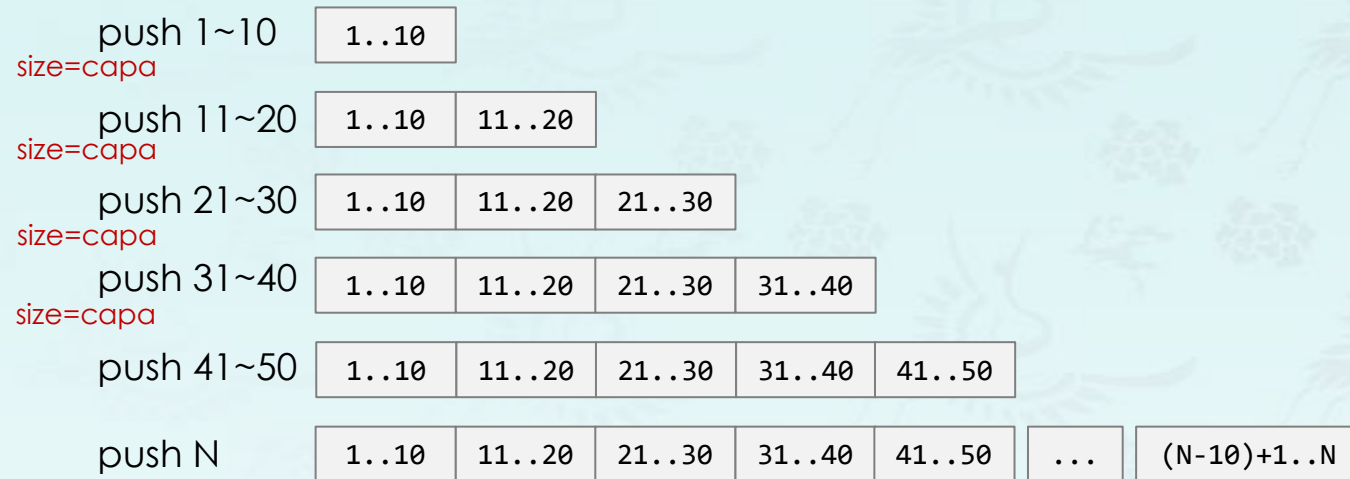   ↑   ↑

1 array access per push   k array accesses when memory is resized by increment of 10
(ignoring cost to create new array)
(assuming new() costs copying each item one by one)
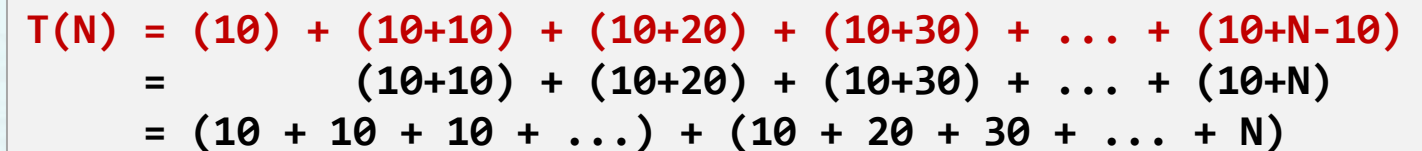
Stack: Using dynamic arrays

- **Q.** Cost of inserting first N items by resize(s.length + 10)?
- **A.** $T(N) = 10N + (10 + 20 + 30 + \ldots + N)$

How many terms? k terms, then N = 10k

$$T(N) = 10N + (10 + 20 + 30 + \cdots + N)$$

$$Let\ N = 10k, then\ it\ becomes$$

$$T(N) = 10N + (10 + 20 + 30 + \cdots + 10k)$$
$$= 10N + 10(1 + 2 + 3 + \cdots + k)$$

$$= 10N + 10\ \frac{k(k + 1)}{2}$$

$$= 10N + 10\ \frac{\frac{N}{10}(\frac{N}{10} + 1)}{2}$$

$$Therefore, T(N) = 10N + \frac{N}{2}\left(\frac{N}{10} + 1\right)$$

The time complexity of the algorithm is $O(n^2)$.

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(capacity * 2)?
- **A.** $T(N) = N + (1 + 2 + 4 + 8 + \ldots + N)$

Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(capacity * 2)?
- Let us consider an example of a simple array stack pushes.

push 1 | 1 |

size=capa

push 2 | 1 | 2 |

size=capa

push 3 | 1 | 2 | 3 | |

push 4 | 1 | 2 | 3 | 4 |

size=capa

push 5 | 1 | 2 | 3 | 4 | 5 | | | |

push 6 | 1 | 2 | 3 | 4 | 5 | 6 | | |

push 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

push 8 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

size=capa

push 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

Stack: Using dynamic arrays
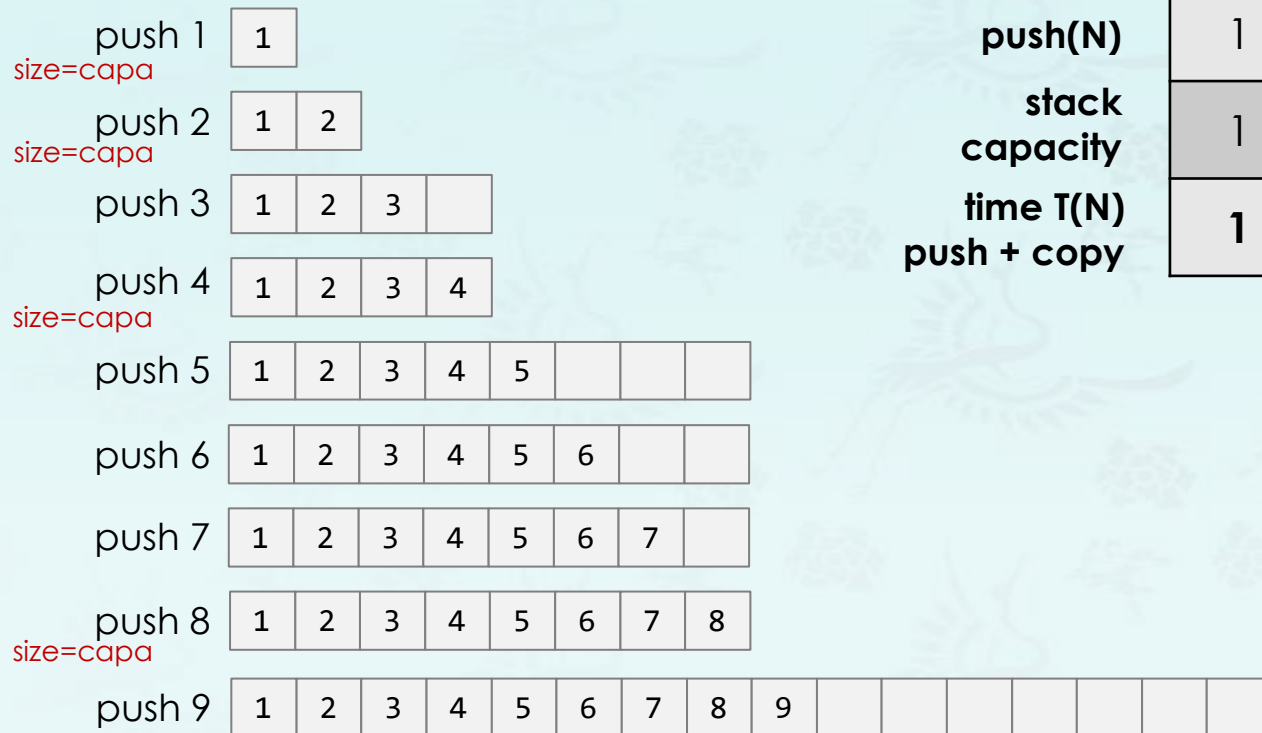
- **Q.** Cost of inserting first *N* items by resize(capacity * 2)?
- Let us consider an example of a simple array stack pushes.

push 1 | 1 |

size=capa

push 2 | 1 | 2 |

size=capa

push 3 | 1 | 2 | 3 | |

push 4 | 1 | 2 | 3 | 4 |

size=capa

push 5 | 1 | 2 | 3 | 4 | 5 | | | |

push 6 | 1 | 2 | 3 | 4 | 5 | 6 | | |

push 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

push 8 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

size=capa

push 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |

push(N)	1	2	3	4	5	6	7	8	9	10
stack capacity	1	2	4	4	8	8	8	8	16	16
time T(N) push + copy	1	1+1	1+2	1	1+4	1	1	1	1+8	1

$(1+1) + (1+2) + (1+4) + (1+8) + ... + (1 + 2^k)$
$= (1 + 1 + 1 + ...) + (1 + 2 + 4 + 8 + ... + 2^k)$
$= (k + 1) + (1 + 2 + 4 + 8 + ... + 2^k)$

```
For example, N = 9 (or push 9, k = 3):
T'(9) = (1+1) + (1+2) + (1+4) + (1+8)
      = (1+1+1+1) + (1+2+4+8)
      = (3+1) + (1+2+4+8)
      = 19        // resizing steps only
```

```
How many 1's in the time for N pushes?
Let it be x. Then T(N) can be expressed
T(N) = x + (k + 1) + (1 + 2 + 4 + 8 + ... + 2^k)
     = N + (1 + 2 + 4 + 8 + ... + 2^k)
     = N + (1 + 2 + 4 + 8 + ... + N)
```
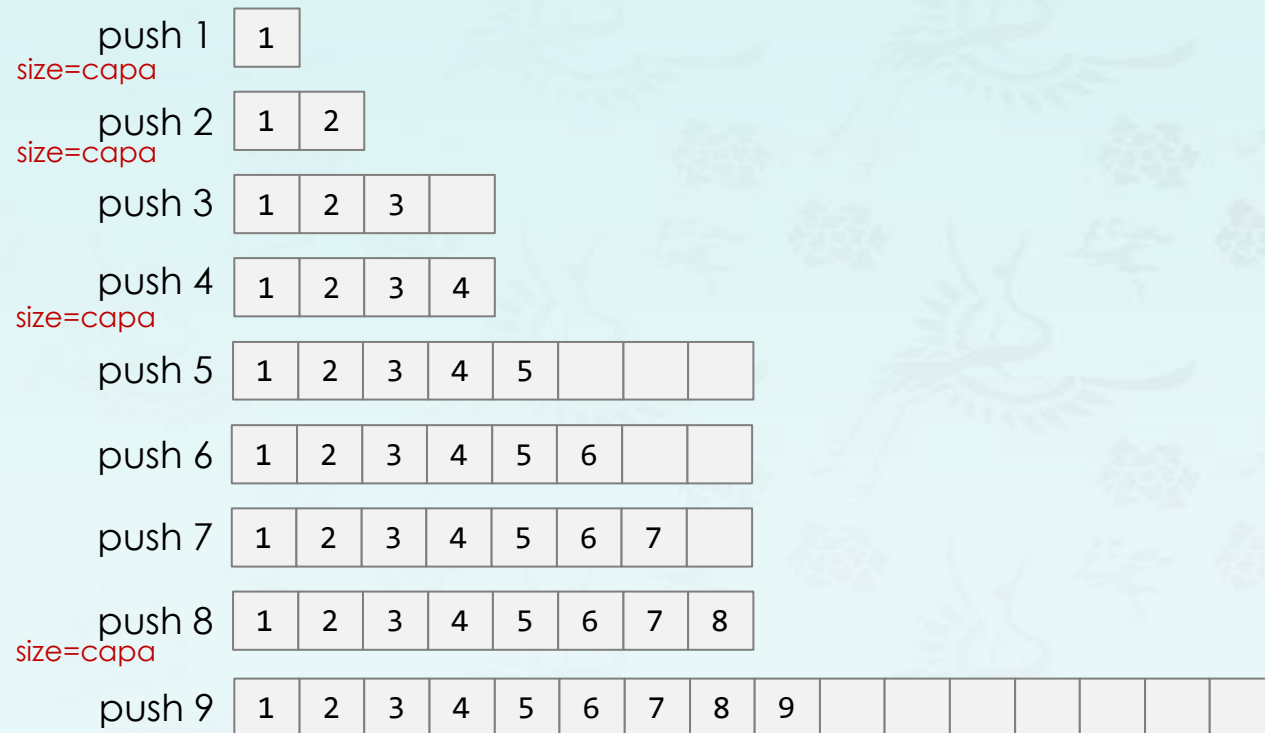
Stack: Using dynamic arrays

- **Q.** Cost of inserting first *N* items by resize(capacity * 2)?
- Let us consider an example of a simple array stack pushes.

push(N)	1	2	3	4	5	6	7	8	9	10
stack capacity	1	2	4	4	8	8	8	8	16	16
time T(N) push + copy	1	1+1	1+2	1	1+4	1	1	1	1+8	1

```
When (N – 1) is not a power of 2,
we can use k = floor(log(N – 1))

For example:

For N = 8, k = 2
T(N) = N + (1 + 2 + 4)
     = 8 + (1 + 2 + 4)
     = 15


For N = 10, k = 3
T(N) = 10 + (1 + 2 + 4 + 8)
     = 25
```

$$(1+1) + (1+2) + (1+4) + (1+8) + \ldots + (1 + 2^k)$$
$$= (1 + 1 + 1 + \ldots) + (1 + 2 + 4 + 8 + \ldots + 2^k)$$
$$= (k + 1) + (1 + 2 + 4 + 8 + \ldots + 2^k)$$

```
For example, N = 9 (or push 9, k = 3):
T'(9) = (1+1) + (1+2) + (1+4) + (1+8)
      = (1+1+1+1) + (1+2+4+8)
      = (3+1) + (1+2+4+8)
      = 19      // resizing steps only
```

```
How many 1's in the time for N pushes?
Let it be x. Then T(N) can be expressed
T(N) = x + (k + 1) + (1 + 2 + 4 + 8 + ... + 2^k)
     = N + (1 + 2 + 4 + 8 + ... + 2^k)
     = N + (1 + 2 + 4 + 8 + ... + N)
```
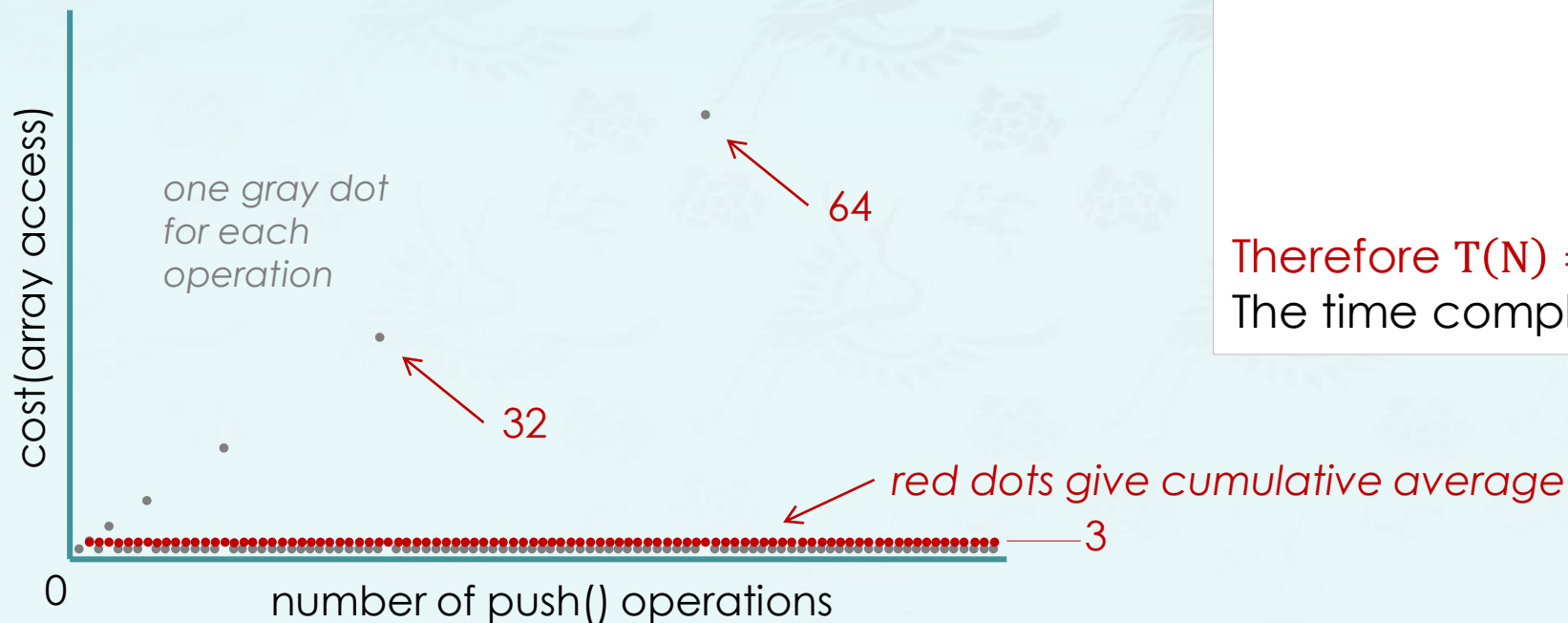
Stack: Using dynamic arrays

- **Q.** Cost of inserting first N items by resize(capacity * 2)?
- **A.** $T(N) = N + (1 + 2 + 4 + 8 + \ldots + N)$

$$1 + a + a^2 + a^3 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$1 + 2 + 4 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Therefore $T(N) = N + (1 + 2 + 4 + \cdots + N) = ?$
The time complexity of the algorithm is O(n).

one gray dot for each operation

64

32

red dots give cumulative average

3

cost(array access)

0

number of push() operations

Stack: Using dynamic arrays

Q. How to grow and shrink array?

"successive doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
stack newStack(int capacity = 1) {
  stack s = new Stack;
  s->item = new string[capacity];
  s->capacity = capacity;
  s->N = 0;
  return s;
}

void resize(stack s, int new_capacity) {
  string *copied = new string[new_capacity];
  for (int i = 0; i < s->N; i++)
    copied[i] = s->item[i];
  delete[] s->item;
  s->item = copied;
  s->capacity = new_capacity;
}
```

```
void push(stack s, string item) {
  // your code here
  s->item[s->N++] = item;
}
```

Stack: Using dynamic arrays

- **Q:** How to shrink array?

- **First try.**
  - **push():** double size of array **s[ ]** when array is full
  - **pop():** halve size of array **s[ ]** when array is one-half full.

- **Too expensive in worst case.**
  - Consider push-pop-push-pop- … sequence when array is full
  - Each operation takes time proportional to N.

N= 5	to	be	or	not	to	be	*null*	*null*

N= 4	to	be	or	not

N= 5	to	be	or	not	to	be	*null*	*null*

N= 4	to	be	or	not

Stack: Using dynamic arrays

- **Q:** How to shrink array?

- **Efficient solution**
  - **push():** double size of array **s[ ]** when array is full
  - **pop():** halve size of array **s[ ]** when array is one-quarter full.

```
void pop(stack s) {
  s->N--;
  // your code here
}
```

❖ **Invariant**. Array is between 25% and 100% full.

Stack: Using dynamic arrays

- **Amortized analysis:**
  - Average running time per operation over a worst-case sequence of operations.
- **Proposition:**
  - Starting from an empty stack, any sequence of N push and pop operations takes time proportional to N.

	best	**worst**	**amortized**
construct	O(1)	O(1)	O(1)
push	O(1)	O(n)	O(1)
pop	O(1)	O(n)	O(1)
size	O(1)	O(1)	O(1)

doubling and halving operations

order of growth of running time
for resizing stack with N items

Stack: Using dynamic arrays

Q. How to grow and shrink array?

"successive doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
stack newStack(int capacity = 1) {
  stack s = new Stack;
  s->item = new string[capacity];
  s->capacity = capacity;
  s->N = 0;
  return s;
}

void resize(stack s, int new_capacity) {
  string *copied = new string[new_capacity];
  for (int i = 0; i < s->N; i++)
    copied[i] = s->item[i];
  delete[] s->item;
  s->item = copied;
  s->capacity = new_capacity;
}
```

```
bool empty(stack s) { return s->N == 0; }

string top(stack s) {
    return s->item[s->N - 1];
}

void push(stack s, string item) {
    // your code here
    s->item[s->N++] = item;
}

void pop(stack s)   {
    s->N--;
    // your code here
}
```

Data Structures
Chapter 3

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - Using Dynamic Array
     - **Using Vector**
     - **Using Template**

2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

**Data Structures
Chapter 3**

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - Using Dynamic Array
     - **Using Vector**
     - **Using Template**
2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Stack: **Vector implementation**

Let's implement our own stack in several different ways.

- Array implementation
  - fixed size array
  - dynamic array
- **Vector implementation**
  - A good tutorial in Korean at https://codechacha.com/ko/cpp-stl-vector/
- Using Template
  - Array implementation
  - Vector implementation

Vector Container

- An alternative to the built-in array.

- A vector is self-grown.

  - No allocation/free is necessary!

- Template implementation advantage!!

- For example:
  vector<int> - vector of integers.
  vector<string> - vector of strings.
  vector<int *> - vector of pointers to integers.
  vector<Shape> - vector of Shape objects. **Shape is a user defined class.**

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

36

Operations on vector

- **iterator begin();**
- **iterator end();**
- bool empty();
- void push_back(const T& x);
- void pop_back();
- void back();
- void clear();
- size_type size();
- size_type capacity();

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

37

Vector Container Example

```cpp
#include<iostream>
#include<vector>
using namespace std;
int main(){
  vector<int> v(5);
  for(int i=0; i < v.size(); i++)
    cin >> v[i];

  for(int i=0; i < v.size(); i++)
    cout << v[i] << ' ';
  cout << endl;
}
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

38

Vector Container Example

```cpp
#include<iostream>
#include<vector>
using namespace std;
int main(){
  vector<int> v(5);
  for(int i=0; i < v.size(); i++)
    cin >> v[i];

  for(int i=0; i < v.size(); i++)
    cout << v[i] << ' ';
  cout << endl;
}
```

```cpp
for(int x: v)
  cout << x << ' ';
cout << endl;


for(auto x: v)
  cout << x << ' ';
cout << endl;


vector<int>::iterator it;
for(it = v.begin(); it!=v.end(); it++)
    cout << *it << ' ';
cout << endl;
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

39

Operations on vector

- iterator begin();
- iterator end();
- **bool empty();**
- **void push_back(const T& x);**
- **void pop_back();**
- **const_reference** back();
- void clear();
- size_type size();
- size_type capacity();

```cpp
int main() {
    int count = 0;
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    while (!vec.empty()) {
        count++;
        vec.pop_back();
    }
    cout << count;
    return 0;
}
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

40

Operations on vector

- `iterator begin();`
- `iterator end();`
- `bool empty();`
- `void push_back(const T& x);`
- `void pop_back();`
- **`const_reference back();`**
- `void clear();`
- **`size_type size();`**
- `size_type capacity();`

```cpp
int main () {
  vector<int> vec;
  vec.push_back(10);
  while (vec.back() != 0) {
    vec.push_back ( vec.back() - 1 );
  }

  cout << "vec contains: ";
  for (unsigned i=0; i<vec.size() ; i++)
    cout << vec[i] << ' ';
  cout << endl;
  return 0;
}
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

41

Operations on vector

- iterator begin();
- iterator end();
- bool empty();
- void push_back(const T& x);
- void pop_back();
- **const_reference back();**
- void clear();
- **size_type size();**
- size_type capacity();

```cpp
int main () {
  vector<int> vec;
  vec.push_back(10);
  while (vec.back() != 0) {
    vec.push_back ( vec.back() - 1 );
  }

  cout << "vec contains: ";
  for (unsigned i=0; i<vec.size() ; i++)
    cout << vec[i] << ' ';
  cout << endl;
  return 0;
}
```

```
vec contains: 10 9 8 7 6 5 4 3 2 1 0
```

Operations on vector

- `iterator begin();`
- `iterator end();`
- `bool empty();`
- `void push_back(const T& x);`
- `void pop_back();`
- `const_reference back();`
- `void clear();`
- **`size_type size();`**
- **`size_type capacity();`**

```cpp
int main () {
  vector<int> vec;
  for (int i=0; i<100; i++) vec.push_back(i);
  cout << "size: " << vec.size() << endl;
  cout << "capa: " << vec.capacity() << endl;
  return 0;
}
```

```
size: 100
capa: 128
```

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

43

`stack4_vec.cpp`

```cpp
struct Stack {
   vector<string> item;
};
using stack = Stack *;

void free(stack s) {
  delete s;
}


int size(stack s) {
   return s->item.size();
}


bool empty(stack s) {
   return s->item.empty();
}
```

```cpp
void pop(stack s) {
   // your code here
}
string top(stack s) {
   // your code here
}
void push(stack s, string item) {
   // your code here
}
void printStack(stack s) {
   while (!empty(s)) {
      cout << top(s) << ' ';
      pop(s);
   }
   cout << endl;  // stack is empty now
}
```

44

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

Stack: Using template

- A **template** is a mechanism that allows a programmer to use types as parameters for a class or a function. The compiler then generates a specific class or function when we **later** provide specific types as arguments.

- A function/class defined using **template** is called a **generic function/class.** This is one of the key features of C++.

- Use **templates** when we need functions/classes that apply the same algorithm to a several types. So we can use the same function/class regardless of the types of the argument or result.

- The syntax is:
  - **template <class T> function_declaration;**

  or
  - **template <typename T> function_declaration;**

Stack: Using template

- A **template** is a mechanism that allows a programmer to use types as parameters for a class or a function. The compiler then generates a specific class or function when we **later** provide specific types as arguments.

- A function/class defined using **template** is called a **generic function/class.** This is one of the key features of C++.

- Use **templates** when we need functions/classes that apply the same algorithm to a several types. So we can use the same function/class regardless of the types of the argument or result.

- The syntax is:
  - **template <class T> function_declaration;**

  or
  - **template <typename T> function_declaration;**

Stack: Using template

Pros and Cons of Templates

- **Pros:**
    - It provides us **type-safe**, **efficient** generic containers and generic algorithms
    - The main reason for using C++ and templates is the trade-offs in performance and maintainability outweigh the bigger size of the resulting code and longer compile times.
    - The drawbacks of not using them are likely to be much greater.

- **Cons:**
    - Templates can lead to **slower compile-times** and possibly larger executable.
    - Compilers often produce incomprehensible poor error diagnostics and **poor error messages**.

Stack: version.4T – using a vector<> in C++ STL

```
struct Stack {
  vector<string> item;
};

using stack = Stack *;

void free(stack s) {
  delete s;
}

string top(stack s) {
  return s->item.back();
}
```

`stack4_vec.cpp`

```
template<typename T>
struct Stack {
  vector<T> item;
};

template<typename T>
using stack = Stack<T> *;

template<typename T>
void free(stack<T> s) {
  delete s;
}

template<typename T>
T top(stack<T> s) {
  return s->item.back();
}
```

Compare these two program segments and see how to use **Templates** in C++ for generic programming.

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

**Data Structures
Chapter 3**

1. **Stack**
   - Stack Concept
     - STL stack class
   - Stack Implementations
     - Using Fixed Array
     - Using Dynamic Array
     - Using Vector
     - Using Template
2. Queue

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University