Lecture 03

List ADT

It is very pervasive

Lecture Overview

- List ADT
  - Specification
- Implementation for List ADT
  -
    - Pros and Cons
  - Linked List Based
    - Pros and Cons

List Applications

- Board games i.e. chess, Ludo etc.
- Store the images
- CPU scheduling in computer
- Contacts list in a phone
- Speech processing
- MS word
- Online ticket booking
- Viewing screen is a 2D list

List in C++ Standard Template Library (STL)

```
// CPP program to show the implementation of
List  #include <iostream>
#include <iterator>
#include <list>
using namespace std;

// Driver Code
int main()
{
        list<int> gqlist1, gqlist2;

        for (int i = 0; i < 10; ++i)

        {

        }      gqlist1.push_back(i * 2);
               gqlist2.push_front(i * 3);
        cout << "\nList 1 (gqlist1) is : ";
        showlist(gqlist1);

        cout << "\nList 2 (gqlist2) is : ";
        showlist(gqlist2);

        cout << "\ngqlist1.front() : " <<
        gqlist1.front();  cout << "\ngqlist1.back() : "
        << gqlist1.back();
```

List in C++ Standard Template Library (STL)

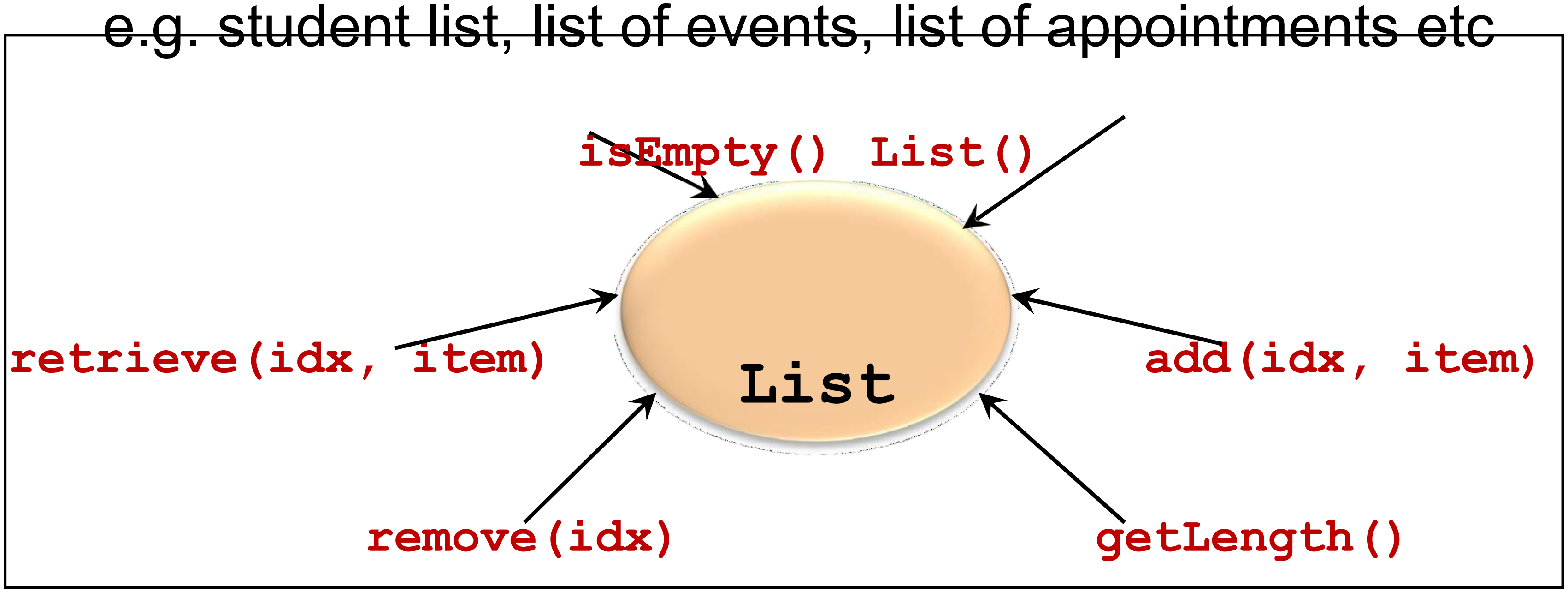
```
cout << "\ngqlist1.pop_front() :
        "; gqlist1.pop_front();
        showlist(gqlist1);

        cout << "\ngqlist2.pop_back() :
        "; gqlist2.pop_back();
        showlist(gqlist2);

        cout << "\ngqlist1.reverse() :
        "; gqlist1.reverse();
        showlist(gqlist1);

        cout << "\ngqlist2.sort():
        "; gqlist2.sort();
        showlist(gqlist2);

        return 0;
}
```

List in C++ Standard Template Library (STL)

```
Output:

List 1 (gqlist1) is :     0     2     4     6     8     10     12     14
16     18

List 2 (gqlist2) is :     27     24     21     18     15     12     9     6
3     0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :     2     4     6     8     10     12     14     16
18

gqlist2.pop_back() :     27     24     21     18     15     12     9     6
3

gqlist1.reverse() :     18     16     14     12     10     8     6     4
2

gqlist2.sort():     3     6     9     12     15     18     21
```

List ADT

- A sequence of items where positional order matter `<a`$_1$`, a`$_2$`,…, a`$_{n-1}$`, a`$_n$`>`
- Lists are very pervasive in computing
  - e.g. student list, list of events, list of appointments etc



The `list` ADT

`idx` : Position, integer
`item` : Data stored in list, can be any data type

List ADT :C++

```
// includes are not shown

class ListBase {
public:
    virtual bool isEmpty() = 0;
    virtual int getLength() = 0;

    virtual bool insert(int index, const int& newItem) = 0;
    virtual bool remove(int index) = 0;
    virtual bool retrieve(int index, int& dataItem) = 0;

    virtual string toString() = 0;
};
```

Operations to check on the state of list.

The three major operations

Operation to ease printing & debugging.

ListBase.h

Design Decisions

- This is a simplified design:
  - to reduce the "syntax burden"
  - to concentrate on the internal logic

- You are encouraged to enhance the class:
  - After you have understood the internal logic

- Possible enhancements:
- Use **Template Class**:
- So that list can contain item of any data type
- Use **Inheritance + Polymorphism**:
- Similar to the Complex Number ADT

Two Major Implementations

1. Array implementation
2. Linked list implementation (discussed soon)

□

General steps:

1. □Choose an **internal data structure**

   e.g. Array or linked list

2. Figure out the algorithm needed for each of □the major operations in List ADT:

   insert, **remove,** and **retrieve**

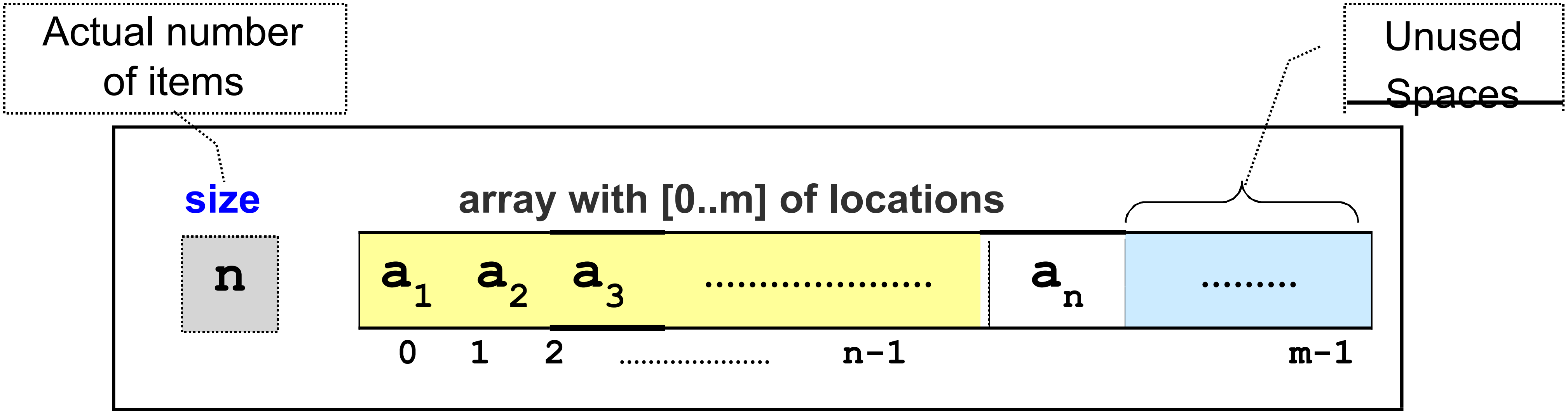
3. Implement the algorithm from step (2)

List ADT –Version A

Array Implementation

Implement List ADT: Using Array

- Array is a prime candidate for implementing the ADT
  - Simple construct to handle a collection of items
- **Advantage:**
  - Verv fast retrieval



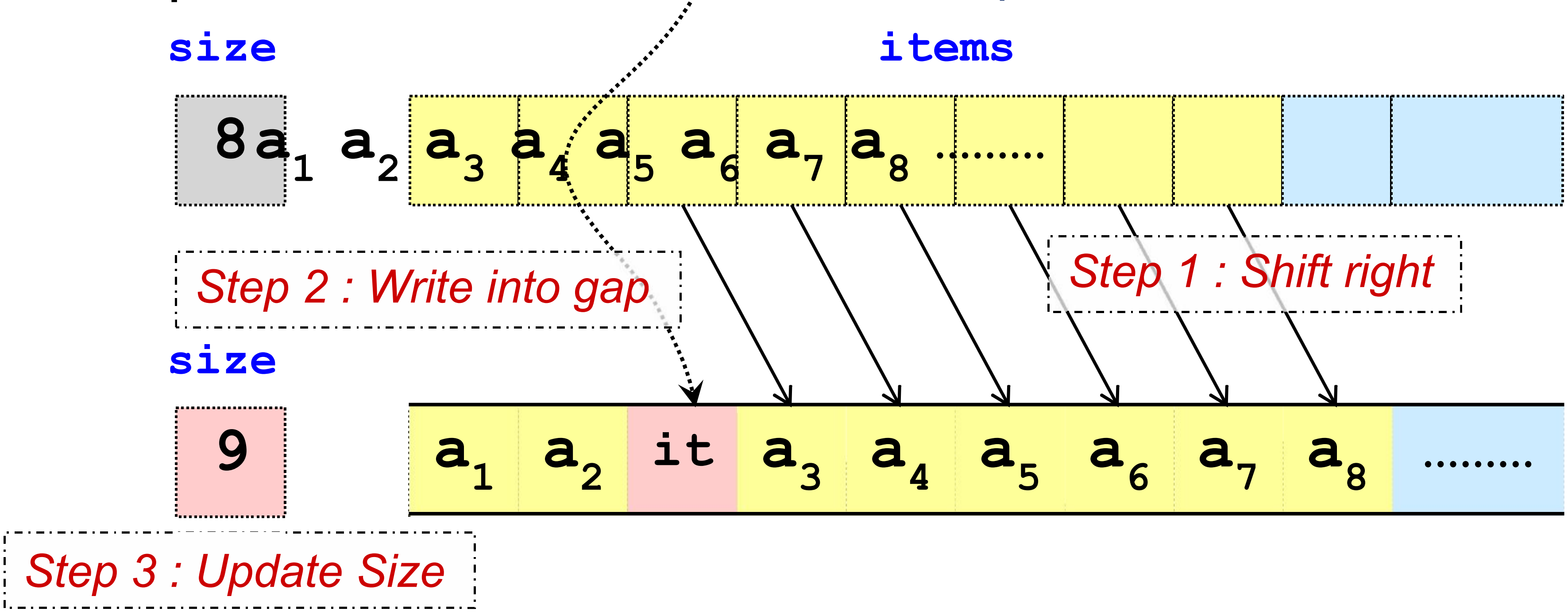
Internal of the `list` ADT, Array Version

Insertion : Using Array

- **Simplest Case:** Insert to the end of array
- Other Insertions:
  - Some items in the list needs to be shifted
  - **Worst case:** Inserting at the head of array

Example

Insert item “***it***” into the 3rd position



size
items
8
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 ....
size
9
a_1 a_2 it a_3 a_4 a_5 a_6 a_7 a_8 ....

Step 1 : Shift right

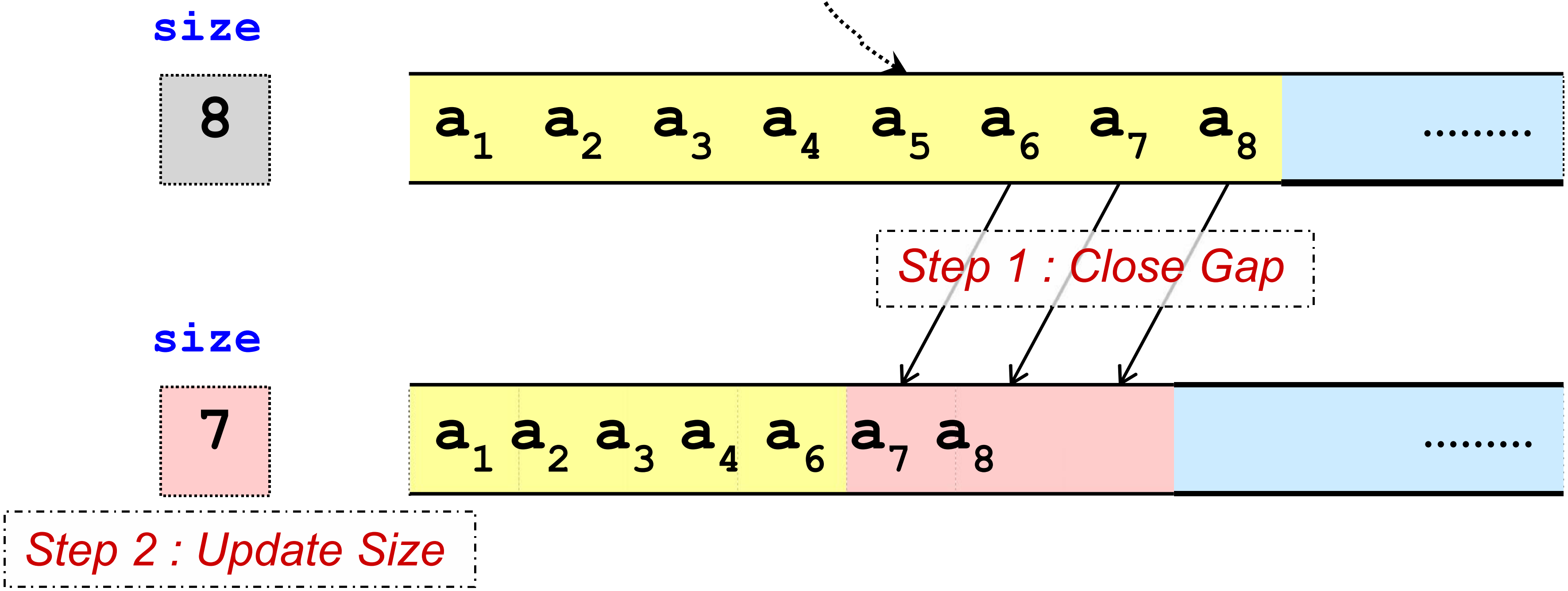
Step 2 : Write into gap

Step 3 : Update Size

Deletion: Using Array

- □ **Simplest Case:** Delete item from the end of array
- □ Other deletions:
  - □ Items needs to be shifted
  - □ **Worst Case:** Deleting at the head of array

Example: remove the item at 5th position


size
8
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 ....
Step 1 : Close Gap
size
7
a_1 a_2 a_3 a_4 a_6 a_7 a_8 ....
Step 2 : Update Size

List Array:

```
#include "ListBase.h"

const int MAX_LIST = 50;

class ListArray : public ListBase {

private:
    int _size;
    int _items[MAX_LIST];

public:
    ListArray();

    virtual bool isEmpty();
    virtual int getLength();

    virtual bool insert(int index, const int& newItem);
    virtual bool remove(int index);
    virtual bool retrieve(int index, int& dataItem);

    virtual string toString();
};
```

Items stored in an static array

ListArray.h

`List Array`: Implementation

```cpp
#include <sstream>
#include "ListArray.h"

ListArray::ListArray() {
   _size = 0;
}

bool ListArray::isEmpty() {
   return _size == 0;
}

int ListArray::getLength() {
    return _size;
}
```

`ListArray.cpp` (Part 1)

- ***`isEmpty()`*** and ***`getLength()`*** methods are easy to code:
  - will be omitted in later implementations

List Array: Implementation

```
bool ListArray::insert(int userIdx, const int& newItem) {
    int index = userIdx-1;

    if (_size >= MAX_LIST)
        return false;

    if ((index < 0) || (index >= _size+1))
        return false;

    for (int pos = _size-1; pos >= index; pos--)
        _items[pos+1] = _items[pos];

    _items[index] = newItem;

    _size++;

    return true;
}
```

List index starts from 1, but array index starts from 0

Maximum capacity reached

List index out of range

Step 1. Shift items

Step 2. Write into gap

Step 3. Update Size

`ListArray.cpp` (Part 2)

List Array: Implementation (3/4)

```
bool ListArray::remove(int userIdx) {
    int index = userIdx-1;

    if ((index < 0) || (index >=
         _size)) return false;

    for (int pos = index; pos < _size-1; pos++)
        _items[pos] = _items[pos+1];

    _size--;

    return true;
}
```

List index out of range

Step 1. Close gap

Step 2. Update size

`ListArray.cpp` (Part 3)

List Array: Implementation (/)

```
bool ListArray::retrieve(int userIdx, int& dataItem) {
    int index = userIdx-1;

    if ((index < 0) || (index >= _size))
        return false;

    dataItem = _items[index];
    return true;
}
```

Retrieval is simple, as array item can be accessed directly.

The result is passed back through the reference parameter

```
string ListArray::toString() {
    ostringstream os;

    os << "[ ";
    for (int i = 0; i < _size; i++)
        os << _items[ i ] << " ";
    os << "]";

    return os.str();
}
```

A useful method to print all items into a string with the format

`[ item1 item2 … itemN ]`

`ListArray.cpp` (Part 4)

Using the List ADT: UserProgram

- Instead of an actual List ADT application, we show a program used to test the implementation of various List ADT operations

- Pay attention to **how we test** the operations:
  - For each operations:
    - Test different scenarios, basically to exercise different "decision path" in the implementation
  - For example, to test the **insert** operation:
    - Insert into an empty list
    - Insert at the first, middle and last position of the list
    - Insert with incorrect index

List ADT :Sample User

```
#include <iostream>
#include "ListArray.h"
using namespace std;

int main() {
    ListArray intList;
    int rItem;

    if (intList.insert(1, 333))
        cout << "Insertion successful!\n";
    else
        cout << "Insertion failed!\n";

    intList.insert(1, 111);
    intList.insert(3, 777);
    intList.insert(3, 555);
```

Using the array implementation of list

This is one way to use the operations: Check the return result for the status of the operation.

If the insertion is implemented properly, the list should contain `[ 111 333 555 777 ]` at this point

`ListTest.cpp` (Part 1)

List ADT :Sample User

```
    cout << intList.toString() << endl;

    intList.retrieve(1, rItem);
    cout << "First item is " << rItem << endl;
    intList.retrieve(intList.getLength(), rItem);
    cout << "Last item is " << rItem << endl;

    cout << "Remove test" << endl;
    intList.remove(1);
    intList.remove(2);
    intList.remove(intList.getLength());

    intList.retrieve(1, rItem);
    cout << "First item is " << rItem << endl;
    intList.retrieve(intList.getLength(), rItem);
    cout << "Last item is " << rItem << endl;

    return 0;
}
```

Test **toString**() and also confirm the content of List

Test **retrieve**() and **getLength**()

Test **removal**():

-remove 1st item in the middle
-remove last item

ListTest.cpp (Part 2)

Array Implementation: **Efficiency** (time)

- **Retrieval:**
  - **Fast:** one access
- **Insertion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all ***N*** elements.
- **Deletion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all ***N*** elements

Array Implementation :Efficiency (space)

- Size of array is restricted to **`MAX_LIST`**
- **Problem:**
  - Maximum size is **not known in advance**
    - **`MAX_LIST`** is too big == unused space is wasted
    - **`MAX_LIST`** is too small == run out of space easily
- **Solution:**
  - Make **`MAX_LIST`** ***a variable***
  - When array is full:
    1. Create a larger array
    2. Move the elements from the old array to the new array
  - No more limits on size, but *space wastage and copying overhead is still a problem*

Array Implementation : Observations

- For **fixed-size collections**
  - Arrays are **great**
- For **variable-size collections**, where dynamic operations such as insert/delete are common
  - Array is a **poor choice** of data structure
  - For such applications, ***there is a better way……***

List ADT –Version B

Linked List Implementation

Summary

- List ADT
  - Usage
  - Specification
- Implementation of List ADT
  - Array Based
    - Pros and Cons
  - Linked List Based
    - Pros and Cons