# C++ & Python Program Design -- Basics

# Array, Lists & Vectors

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

# Creating Lists 链表

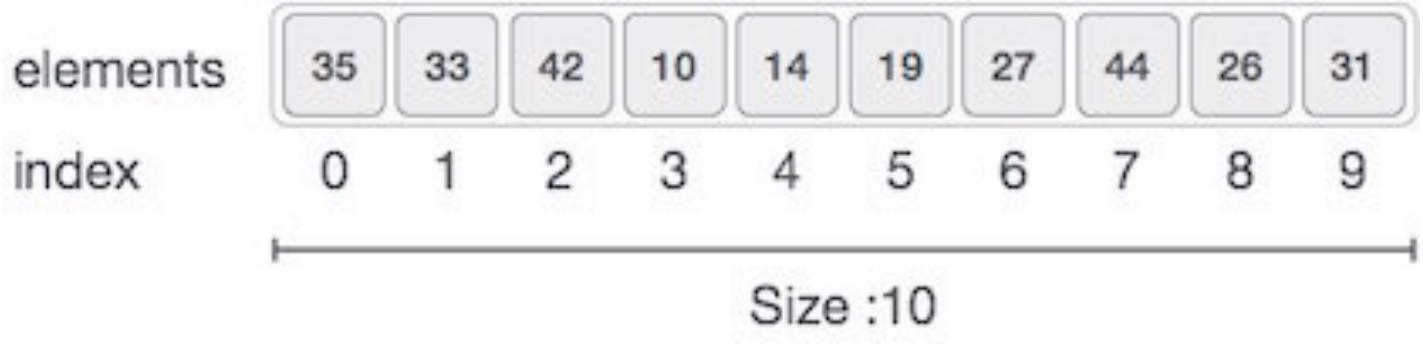
```
colors = ["red", "yellow", "green", "blue"]
type(colors)#<class 'list'>

list("Python")#['P', 'y', 't', 'h', 'o', 'n']

groceries = "eggs, milk, cheese"
grocery_list = groceries.split(", ")
print(grocery_list) #['eggs', 'milk', 'cheese']

"The quick brown fox".split(" ")
"abbaabba".split("ba")
"abbaabba".split("c")
```

# Array in Python

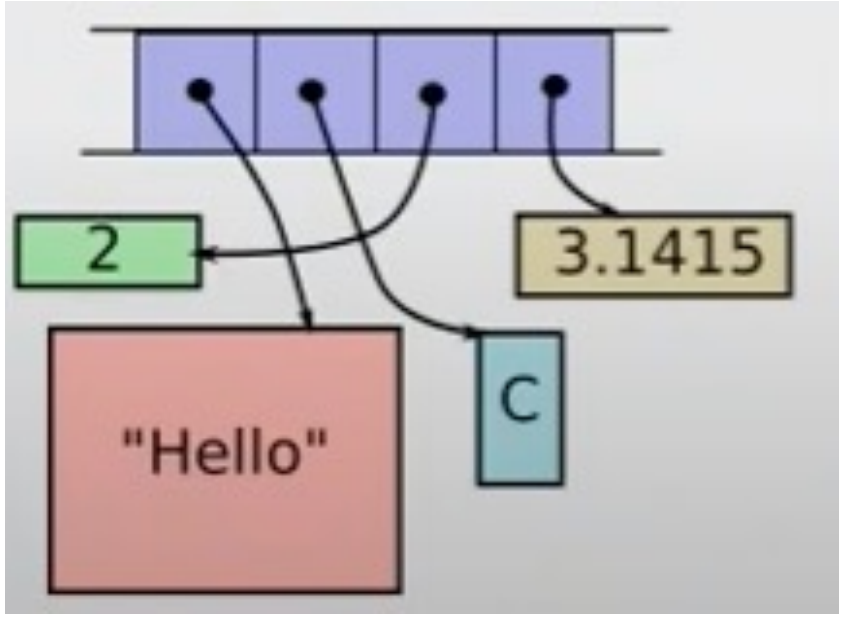


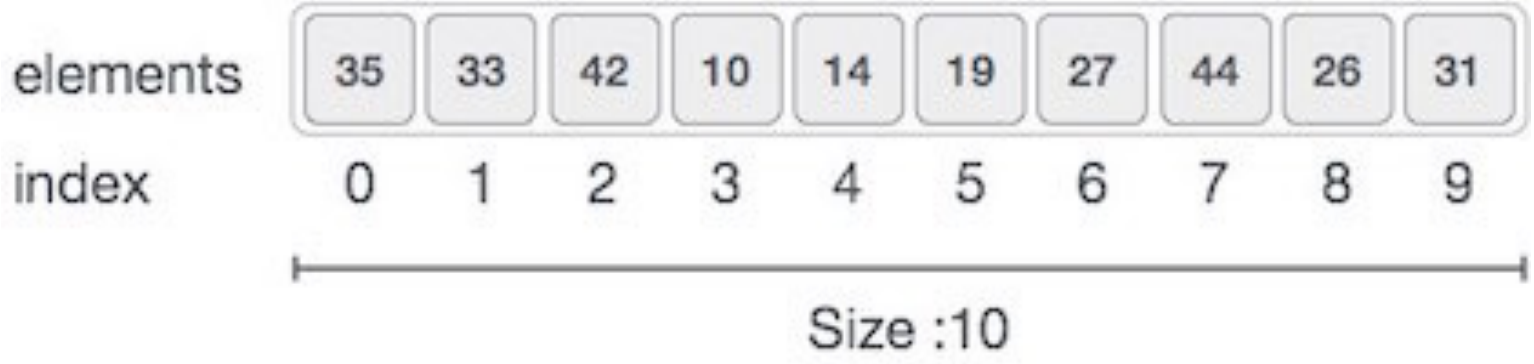

- a sequence 序列 of multiple items/elements that are of the same type.
  - Item/element: 项，元素
- contain items that are indexed by integers, starting with 0.
  - Index索引
- 但是不常用，费劲，一般用list或者numpy中的array

# Definition of list in Python

- Lists are mutable sequences.
- Lists contain items that are indexed by integers, starting with 0.

- arr = [1, "hello", 'c', 3.14]
- print(arr[0], arr[1], arr[2], arr[3])



# Basic List Operations

```
numbers = [1, 2, 3, 4]
numbers[1]
numbers[1:3]#[2, 3], 左闭右开区间[)

"Bob" in numbers #False

for number in numbers:
    if number % 2 == 0:
        print(number)
```

# Basic List Operations

| Python Expression | Results | Description |
|---|---|---|
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] * 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

# Indexing, Slicing, and Matrixes

- L = ['spam', 'Spam', 'SPAM!’]

- print(L[2])
- 'SPAM!
- print(L[1:])
- ['Spam', 'SPAM!’]
- print(L[-2])
- 'Spam

# Changing Elements in a List

```
colors = ["red", "yellow", "green", "blue"]
colors[0] = "burgundy"
colors[1:3] = ["orange", "magenta"]
print(colors)
#['burgundy', 'orange', 'magenta', 'blue']
```

# Adding and Removing Elements

```
colors = ["red", "yellow", "green", "blue"]
colors.insert(1, "orange")
#['red', 'orange', 'yellow', 'green', 'blue']
colors.insert(-1, "indigo")
#['red', 'orange', 'yellow', 'green', "indigo",
'blue']

color = colors.pop(3) #'green'
print(colors)#['red', 'orange', 'yellow', "indigo", 'blue']

colors.append("green")#['red', 'orange', 'yellow',
"indigo", 'blue', 'green’]
colors.extend(("violet", "ultraviolet"))
```

# Lists of Numbers

```
numbers = [1, 2, 3, 4, 5]
sum(numbers)
min(numbers)

#list Comprehension: a short-hand for a for loop
squares = [num**2 for num in numbers]
print(squares)

str_numbers = ["1.5", "2.3", "5.25"]
float_numbers = [float(value) for value in
str_numbers]
```

# Python would never let you iterate beyond the end of a list.

```
mylist = [1,2,3]
print(mylist[10])
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(mylist[10])
IndexError: list index out of range
```

# range() 2 list

range([start,] stop [, step]) -> range object
3 examples:

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]

>>> list(range(1, 20, 3))
[1, 4, 7, 10, 13, 16, 19]

>>> list(range(20, 10, -5))
[20, 15]
```

# Built-in List Functions & Methods

| No. | Function with Description |
|---|---|
| 1 | cmp(list1, list2)Compares elements of both lists. |
| 2 | len(list)Gives the total length of the list. |
| 3 | max(list)Returns item from the list with max value. |
| 4 | min(list)Returns item from the list with min value. |
| 5 | list(seq)Converts a tuple into list. |

# Built-in List Functions & Methods

| No. | Methods with Description |
|---|---|
| 1 | list.append(obj)Appends object obj to list |
| 2 | list.count(obj)Returns count of how many times obj occurs in list |
| 3 | list.extend(seq)Appends the contents of seq to list |
| 4 | list.index(obj)Returns the lowest index in list that obj appears |
| 5 | list.insert(index, obj)Inserts object obj into list at offset index |
| 6 | list.pop(obj=list[-1])Removes and returns last object or obj from list |
| 7 | list.remove(obj)Removes object obj from list |
| 8 | list.reverse()Reverses objects of list in place |
| 9 | list.sort([func])Sorts objects of list, use compare func if given |

# Review Exercises

1. Create a list named food with two elements "rice" and "beans".
2. Append the string "broccoli" to food using .append().
3. Add the string "bread" and "pizza" to "food" using .extend().
4. Print the first two items in the food list using print() and slicing notation.
5. Print the last item in food using print() and index notation.

6. Create a list called breakfast from the string "eggs, fruit, orange juice" using the string .split() method.
7. Verify that breakfast has three items using len().
8. Create a new list called lengths using a list comprehension that contains the lengths of each string in the breakfast list.

# Array in C++

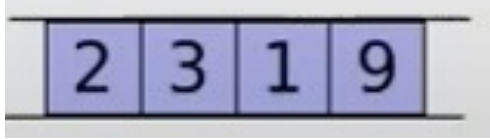
# What is array?

- always stored in contiguous memory

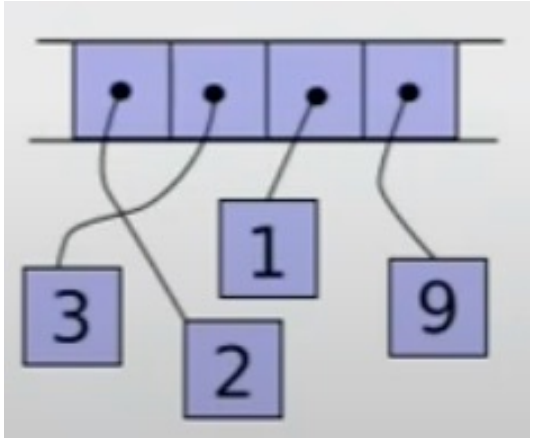
| Memory Address | 0x0400 | 0x0401 | 0x0402 | 0x0403 | 0x0404 |
|---|---|---|---|---|---|
| Content | 21 | 47 | 87 | 35 | 92 |
| Index | 0 | 1 | 2 | 3 | 4 |

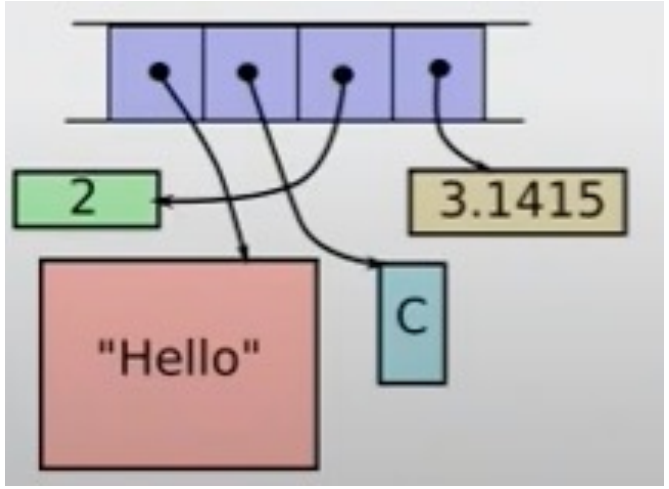
# C Array vs Python List

- Same type of objects
- int arr[4];




- Many type of objects
- arr = [1, "hello", 'c', 3.14]
- arr[0]=2;arr[1]=3;arr[2]=1;arr[3]=9




From 8.3 C Arrays vs Python Lists

# C Array vs Python List Initialization 初始化

```
char arr1[10];
int arr2[2] = {1, 2};
double arr3[] = {1.2, 3.3};

cout << arr2[0] << endl;
```

- Must specify size of the array in advance

```
arr = []
arr2 = [1, 2]

Print(arr2[0])
```

- No need to specify size of the array in advance

# Visiting the item specified by index索引

- C array

- `int arr[2] = {1, 2};`
- arr[0];

- 1st item is index 0
- Last item is SIZE - 1
- Can not use negative indices

- Python list

- arr = [1, 2]
- arr[0]
- print(arr[-1]) #2

- 1st item is index 0
- Last item is SIZE - 1
- Can use negative indices

# Fixed size vs. Variable size

```
int arr2[2];
```

```
arr2 = [1, 2]
arr2.append(3)
arr2.pop(1)
print(arr2) # [1, 3]
```

Once created, can not grow or shrink

Can grow
Can shrink

# Without vs. with builtin methods

- No builtin methods

- Many builtin methods

- append()
- pop()
- reverse()
- count()
- …

# No slicing vs has slicing

```
array[2:5] ✗
array[:3] ✗
array[2:] ✗
array[::-1] ✗
```

- Write own code for slicing

```
array[2:3] ✓
array[:3] ✓
array[2:] ✓
array[::-1] ✓
```

# Array data types

- Arrays can be made from any data type.
  - Arrays can also be made from structs结构体.

```
struct Rectangle
{
    int length;
    int width;
};
Rectangle rects[5]; // declare an array of 5 Rectangle
```

# Arrays and off-by-one errors

What will be the output?

```
int scores[] = { 84, 92, 76, 81, 56 };
const int numStudents = sizeof(scores) / sizeof(scores[0]);

int maxScore = 0; // keep track of our largest score
for (int student = 0; student <= numStudents; ++student)
    if (scores[student] > maxScore)
        maxScore = scores[student];

std::cout << "The best score was " << maxScore << '\n';
```

# 越界导致改写了otherdata

A illustrative example, not work as it means actually.

```
int myarray[] = {2, 4};
int otherdata[]={777, 777};
for (int i=0; i<4; i++){
     myarray[i]=0;
     cout <<"myarray["<< i << "]=" <<
myarray[i]<< endl;
     cout << "add:" << &myarray[i] <<
endl;
}
for (int i=0; i<2; i++){
     cout <<"otherdata["<< i << "]=" <<
otherdata[i]<< endl;
     cout << "add:" << &otherdata[i] <<
endl;
}
```

```
myarray[0]=0
add:0x7ffd854b3538
myarray[1]=0
add:0x7ffd854b353c
myarray[2]=0
add:0x7ffd854b3540
myarray[3]=0
add:0x7ffd854b3544
otherdata[0]=0
add:0x7ffd854b3540
otherdata[1]=0
add:0x7ffd854b3544
```

# Arrays and off-by-one errors

```
int scores[] = { 84, 92, 76, 81, 56 };
if (scores[5] > maxScore)
        maxScore = scores[5];
```


- But scores[5] is undefined!
- This can cause all sorts of issues, with the most likely being that scores[5] results in a garbage value. In this case, the probable result is that maxScore will be wrong.
- However, imagine what would happen if we inadvertently assigned a value to array[5]!
- We might overwrite another variable (or part of it), or perhaps corrupt something -- these types of bugs can be very hard to track down!

# Be Cautious with Arrays

- The speed and low-level control is powerful… and dangerous.
- As a Python programmer, using a C++ array will help you better understand the trade-offs of the protections Python offers.
- Because C++ will generally try to do everything you ask for.

# Passing arrays to functions

```
// value is a copy of the argument
// so changing it here won't change the value of the argument
void passValue(int value) { value = 99; }
// prime is the actual array (pointer of 1st element)
// so changing it here will change the original argument!
void passArray(int prime[5]) {  prime[0] = 11;  prime[4] = 22;  }

int main(){
   int value = 1;      passValue(value);
   cout << "after passValue: " << value << "\n";
   int prime[5] = { 2, 3, 5, 7, 11 };      passArray(prime);
   cout << "after passArray[0]: " << prime[0]  <<  ”[4]: " <<  prime[4] << "\n";
   return 0;}
```

# Prevent updating

```
// even though prime is the actual array, within this function it should be trea
ted as a constant
void passArray(const int prime[5])
{
    // so each of these lines will cause a compile error!
    prime[0] = 11;
    prime[1] = 7;
    prime[2] = 5;
    prime[3] = 3;
    prime[4] = 2;
}
```

# sizeof and arrays

```
void printSize(int array[]){
    // prints the size of a pointer, not the size of the array!
   std::cout << sizeof(array) << '\n';
}

int main(){
   int array[] = { 4, 1, 2, 3, 5, 8, 13, 21 };
    // will print the size of the array in bytes
   std::cout << sizeof(array) << '\n';
   printSize(array);
   return 0;
}
```

```
array: 0x00007ffeefbff670
  *array: 4
&array: 0x00007ffeefbff648
> *$4: 0x00007ffeefbff670
&array[0]: 0x00007ffeefbff670
  *$5: 4
array[0]: 4
```

```
WATCH
> array: [8]
> &array: 0x00007ffeefbff670
> &array[0]: 0x00007ffeefbff670
```

**What is printed?**

**32**

**4**

# Array as function parameters

```
void printSize(int array[], int len){
    cout << sizeof(array) << '\n’;
    for (int i = 0; i <len; ++i){cout << array[i] << endl ;}
}
int main(){
    int array[5] = { 9, 7, 5, 3, 1 };
    std::cout << "The array has address: " << array << '\n’;
    std::cout << "Element 0 has address: " << &array[0] << '\n';
    std::cout << sizeof(array) << '\n';
    arrInfo(array, 5);
    return 0;}
```

# Typeid & sizeof

自己测试，不讲

```
void printSize(int array[]){
int* parr; int& arr=array[0];
cout << typeid(array).name() << ": " << sizeof(array) << endl;
cout << typeid(parr).name() << ": " << sizeof(parr) << endl;
cout << typeid(arr).name() << ": " << sizeof(arr) << endl;
}

int array[] = { 4, 1, 2, 3, 5, 8 };
cout << typeid(array).name() << ": " << sizeof(array) << endl;
printSize(array);
```

A6_i: 24

Pi: 8

Pi: 8

i: 4

# Review Exercises

- Print the following array to the screen using a loop:

```
const int arrayLength(9);
int array[arrayLength] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
```

# std::array

# An introduction to std::array in C++11

- #include <array>
- std::array<int, 5> myarray; // declare an integer array with length 5
- myarray = { 0, 1, 2, 3, 4 }; // okay
- myarray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!
- myarray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer l ist!

- std::array<int, 5> myarray2 { 9, 7, 5, 3, 1 }; // uniform initialization

# at() has bounds checking, but () hasn't

- `std::array<int, 5> myarray { 9, 7, 5, 3, 1 };`

- `myarray.at(1) = 6;` // array element 1 valid, sets array element 1 to value 6
- `myarray.at(9) = 10;` // array element 9 is invalid, will throw error

- `myarray[9] = 6;` // bad things will probably happen, but who knows, no exception thrown

# Size and sorting

- Because std::array doesn't decay to a pointer when passed to a function, the size() function will work even if you call it from within a function:

```
void printSize(const std::array<double, 5> &myarray){
    std::cout << "size: " << myarray.size();
}

int main(){
    std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

    printSize(myarray);

    return 0;}
```

```
#include <array>
#include <algorithm> // for std::sort

int main(){
    std::array<int, 5> myarray { 7, 3, 1, 9, 5 };
    std::sort(myarray.begin(), myarray.end()); // sort the array forwards
//    std::sort(myarray.rbegin(), myarray.rend()); // sort the array backwards

    for (const auto &element : myarray)
        std::cout << element << ' ';

    return 0;}
```

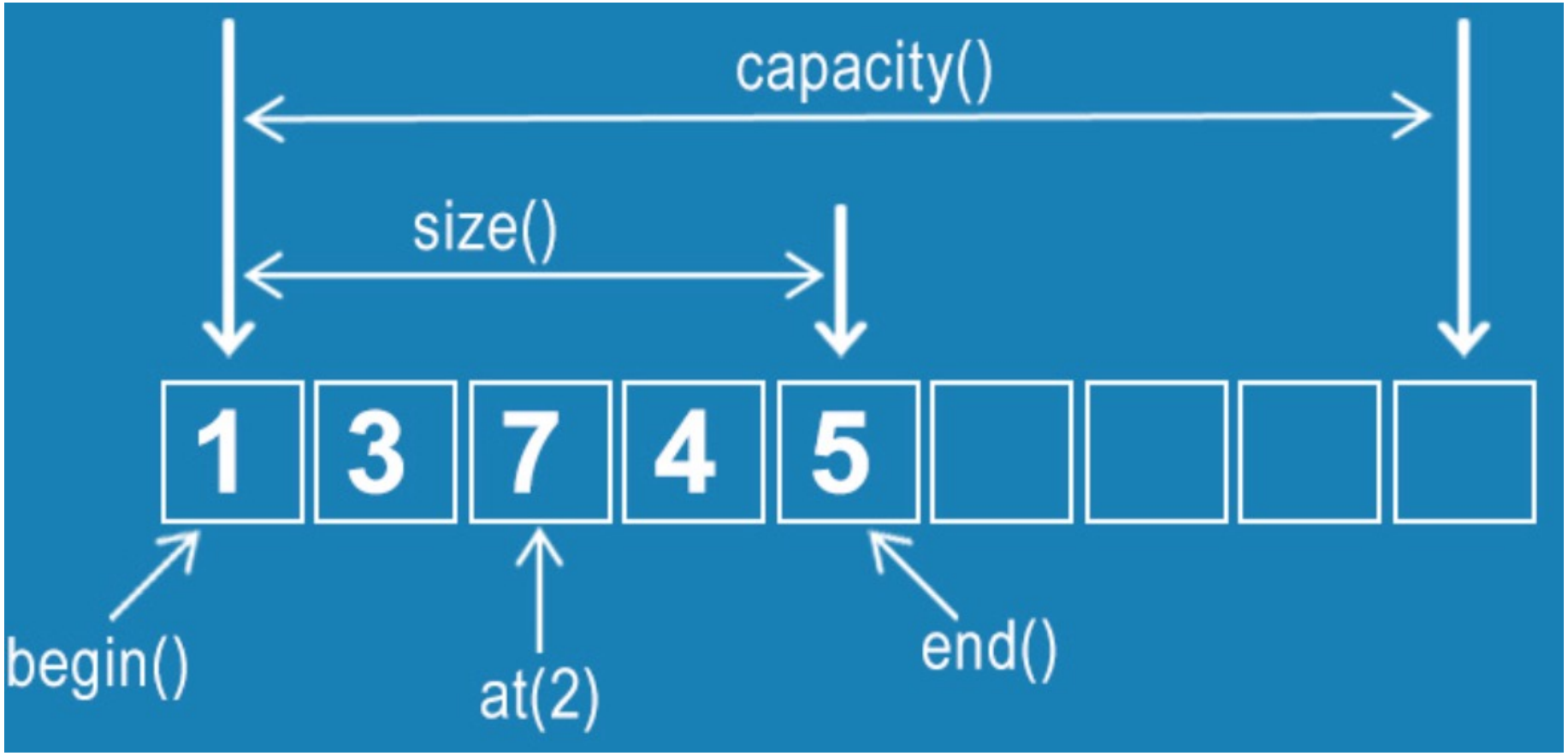
# Summary

- std::array is a great replacement for build-in fixed arrays.
- It’s efficient, in that it doesn’t use any more memory than built-in fixed arrays.
- using std::array over built-in fixed arrays for any non-trivial use.

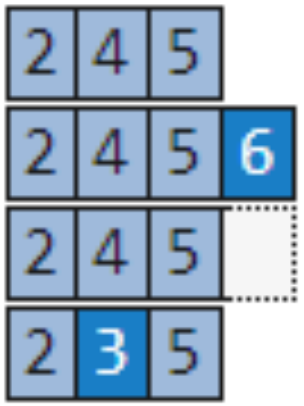
# std::vector

# std::vector

- is a sequence container (dynamic array) which resizes itself automatically.



```
#include <vector>
std::vector<int> v {2,4,5};
v.push_back(6);
v.pop_back();
v[1] = 3;   v.at(1) = 3;// safer, slower
```




```
cout << v[2];
for(int x : v) cout << x << ' '
```

prints 5

prints 2 3 5

```
v.reserve(8);
v.resize(5, 0);
cout << v.capacity();
cout << v.size();
```


prints 8

prints 5

# #include <vector>

| Common Operation | Use | Explanation |
|---|---|---|
| [ ] | myvector[i] | access value of element at index i |
| = | myvector[i]=value | assign value to element at index i |
| push_back | myvect.push_back(item) | Appends item to the far end of the vector |
| pop_back | myvect.pop_back() | Deletes last item (from far end) of the vector |
| insert | myvect.insert(i, item) | Inserts an item at index i |
| erase | myvect.erase(i) | Erases an element from index i |
| size | myvect.size() | Returns the actual size used by elements |
| capacity | myvect.capacity() | Returns the size of allocated storage capacity |
| reserve | myvect.reserve(amount) | Request a change in capacity to amount |

# Conclusion

- std::vector handle their own memory management (which helps prevent memory leaks),
- remember their size,
- and can be easily resized,

- **using std::vector in almost all cases where dynamic arrays are needed.**

# Practices: python & c++

// function that uses a vector to square every number from 0 to 49

// uses the reserve operation to save space in memory

```
vector<int> intvector;
intvector.reserve(50);
for (int i=0; i<50; i++){
    intvector.push_back(i*i);
    cout << intvector[i] << endl;
}
```

```
intlist=[]
for i in range(50):
    intlist.append(i*i)
    print(intlist[i])
```

# Why reserve()?

```
vector<int> intvector;
//intvector.reserve(50);
for (int i=0; i<50; i++){
    intvector.push_back(i*i);
    cout << intvector[i] << endl;
    cout << "capacity: " <<
intvector.capacity() << endl;
}
```

```
0
capacity: 1
1
capacity: 2
4
capacity: 4
9
capacity: 4
16
capacity: 8
```

不使用reserve，会产生额外操作

# comprehensive quiz

- Arrays, C-style strings
- **Be careful not to index an array out of the array's range.**
- Represent 2D matrix as 1D array
- **std::array, std::vector**

# Quiz time: Basic

- What's wrong with each of these snippets, and how would you fix it?

```
int main()
{
    int array[5] { 0, 1, 2, 3 };
    for (int count = 0; count <= 5; ++count)
        std::cout << array[count] << " ";

    return 0;
}
```

# Quiz time: Advanced

- What's wrong with each of these snippets, and how would you fix it?

```
void printArray(int array[]){
        for (const int &element : array)
        std::cout << element << ' ';
}

int main(){
        int array[] { 9, 7, 5, 3, 1 };
        printArray(array);

return 0;}
```