Sequences (Outline)

- Storing Objects in Computer Memory
- Pointers and One-Dimensional Arrays
- STL Class Vector
- Two-Dimensional Arrays
- STL Class Matrix
- Linked Lists
- STL Class List
- Sequences

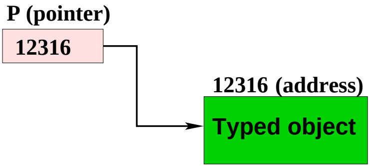
Storing Objects in Computer Memory

- A computer's memory is a sequence of numbered bytes from 0 to the last one.
- A byte number is an address of an object in the computer memory.

```
int main()
{
   int val = 0;
   char ch = 'A';
   string name = "All";
}
```

		val	ch	name		
		0	'A'	"All"		
0	1	......	......	......	$2^{31}-2$	$2^{31}-1$

A *pointer* is a memory address of an object of a specified type, or it is a variable which keeps such an address.



Pointer properties:

- A pointer value is the address of the first byte of the pointed object in the memory.
- A pointer does *not* know about how many bytes it points to.

Pointers (cont)

Pointers are used to:

- create more complex data types such as lists, queues, stacks, trees, or graphs
- process arrays
- keep track of allocated memory returned by the operator **new**
- initialize to nothing (no address) denoted by NULL, 0, or **nullptr** in C++11.
- deallocate a block of memory by the operator **delete** or **delete []**

Arrays

- An array of the fix size represents a contiguous sequence of objects of the same type allocated on the stack computer memory. It is a very simple data structure often used in sorting and searching operations.
- The position of an element in the array is called the *index*. In C++ arrays always begin with the index 0:

0	1	2	3	4	5	(indices)
12	-3	24	65	92	11	(array values)

- To declare a static one-dimensional array with fixed size at compile time (in the stack part of computer memory):

```
constexpr max_size = 6;
int array[max_size] = {12,-3,24,65,92,11};
```

  where `max_size` is a constant expression, a compile-time constant known during compilation.

Arrays (cont.)

- The total number of items inserted into an array is called the *logical size* of the array. A special variable must be used to keep track of the current number of items.
- The logical size should be not greater than the physical size of an array (= maximum size of an array).
- Operations on arrays
  - An array provides a random access to its elements: `array[index]` where `0 <= index < max_size`.
  - C++ does not check the "index-out-of-bounds" condition and it is a programmer's responsibility to ensure that the index range is valid. It is easy to go out of range, e.g., there is no array item such as `array[max_size]`.
  - There is no possibility of resizing this type of arrays.

Arrays (cont.)

- To initialize all array elements to zero: `array[max_size] ={};`
- Reading from the standard input (usually the keyboard) to an array of size 100:

```
for (int i = 0; i < 100; i++) cin >> array[i];
```

- Writing to the standard output (usually the screen) from an array of size 100:

```
for (int i = 0; i < 100; i++) cout << array[i] << " ";
```

Arrays (cont.)

- Linear search for a target `x`:

```
for (int i = 0; i < 100; i++)
   if (x != array[i]) i++;
   else return i; // returns an index if x is found
return -1; // returns -1 if x is not found
```

- Copying:

```
for (int i = 0; i < 100; i++)
   other_array[i] = array[i];
// assigning other_array = array; is illegal
```

Arrays (cont.)

- Note that the base address of an array in computer memory is the address of the first element of the array.
- A size of a constant array can be computed by a compiler.

```
const int days_in_month[] =
   {31,28,31,30,31,30,31,31,30,31,30,31};
int num_of_months =
   sizeof(days_in_month)/sizeof(*days_in_month);
   // size can be computed
```

  The value of `num_of_months` will be `12`.

Enumeration Type

- An example of enumeration type as an index to an array.

```
enum Color {black, white, red, blue};
Color balls[4];
Color col; // a variable used as an array index
for (col = black; col <= blue; col = (Color) (col+1))
   balls[col] = col;
```

Typedef and an Array

- `Typedef` is used to define a new type from an existing one:

```
constexpr int max_size = 100;
typedef double Table[max_size]; // Table is type name
```

- Declarations of other variables of array type with typedef-defined type.

```
Table my_table; // the same as double my_table[max_size];
Table A[12]; // the same as double A[12][max_size];
```

Passing an Array to a Function

- An array as an actual argument of a function is passed by as a pointer to its first element. The formal argument of form `T[]` is converted to `T*` (`T` is any C++ type).

Example

```
// we want to copy elements of the array A to B
  int A[10], B[10];
  ... // insert values to A
  copy_elems(A, 10, B, 10);  // copy A to B
```

Here is the body of the function (possibly we should check if `size_a <= size_b`):

```
void copy_elems(int A[], int size_a, int B[], int size_b)
{
   for (int i = 0; i < size_a; i++)
      B[i] = A[i];
}
```

Dynamic Arrays

```
int* a = new int[3];// with elements 10, 20, 30
int* aptr = a; // aptr is an alias to a
```

It allocates 3 contiguous memory locations on the heap (free store) part of memory, each of type `int` and stores the address of the first memory location in `a`. Note that the pointer variable `a` is kept on the stack part of memory.

	stack						heap				
...	a=1000						10	20	30	...	
	200						1000	1001	1002		(addresses)

```
++a; //move a forward by 1
cout<< *a; // prints 20
--a; // move a backward by 1
cout << *a; // prints 10
--a; // out of range
```

It is easy to go out-of-range in the case of static and dynamic arrays when operations on pointers are used.

Memory Deallocation

- Deallocation of memory allocated by the operator `new`:

```
delete [] a; // now a is a dangling pointer
a = nullptr;
```

- A deep copy of an array. Here `aptr` is a separate array initialized by the values of `a`.

```
aptr = new int[10];
for (int j = 0; j < 10; j++)
   aptr[j] = a[j];
```

- A dynamic array of pointers.

```
int **ptd = new (int *)[10];
for (int i = 0; i < 10; i++)
   ptd[i] = new int(1);
```

Expandable Dynamic Arrays

Steps to create an expandable dynamic array in C++.

1. Create an array with a reasonable default size at the beginning of the program.
2. When an array cannot hold more data, increase its size by creating a new larger array (usually by doubling its size).
3. Copy data from the old array to the new larger one.
4. Set the array variable to point to the new array. Deallocate the old array.
5. (Optional) When an array is too large, decrease its size in a similar way.

An expandable dynamic array is used in the definition of the class `vector`.

Resizing Array

- Resizing takes several steps

```
// resize an array
int* resize_array(int *array, int capacity)
{
   int new_capacity = 2*capacity;
   int new_array[] = new int[new_capacity];
   for (int i = 0; i <= capacity; i++)
      new_array[i] = array[i];
   delete [] array; // remove old contents
   return new_array;
}
... // in main()
int *A = new int[10];
... // populate A
A = resize_array(A, 10); // A is overwritten
```

- Easy to get memory leaks if you forget to deallocate memory by `delete [] array;`

Concerns about Arrays

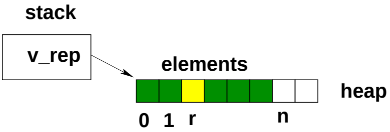
- Advantages of arrays
  - Direct mapping to hardware
  - Efficient for low-level operations
  - Direct language support
- Problems with arrays
  - No range checking
  - An array does not know its own size
  - There are no operations on arrays such as copy or assignment

STL Class Vector

- The Vector ADT extends the notion of the array by storing a sequence of arbitrary objects. A vector class wraps around an array and adds array operations.
- A vector element can be accessed, inserted or removed by specifying its rank (= number of elements preceding it; also called index).
- An exception is thrown if an incorrect rank is specified (e.g., a negative or out-of-bound rank).

Vector Operations: Access & Replace

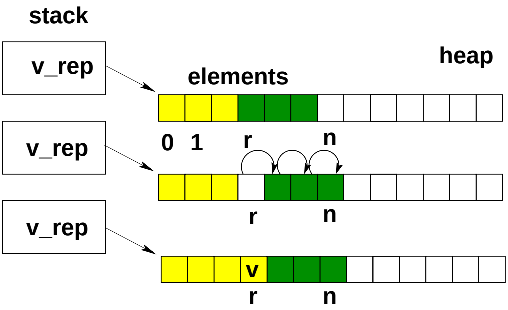
- `elem_at_rank(int r)`: returns the element at rank `r` without removing it



- `replace_at_rank(int r, int v)`: replaces the element value at rank `r` with `v`

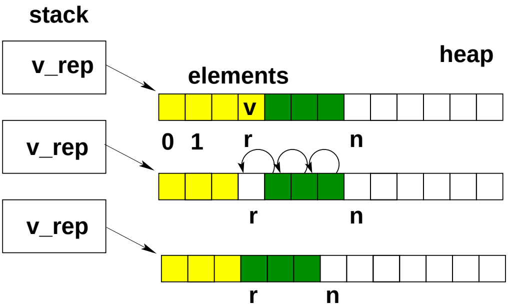
Vector Operations: Insert

- `insert_at_rank(int r, int v)`: insert a new element `v` at rank `r`



Vector Operations: Remove

- `remove_at_rank(int r)`: removes the element at rank `r`



- Additional operations: `size()` and `is_empty()`

Applications of Vectors

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Building blocks for other data structures

STL Class Vector

- Selected functions in the STL vector class
  - `size()` – logical number of elements in a vector
  - `capacity()` – physical number of elements in a vector (allocated memory)
  - `empty()` – returns true if a vector is empty
  - overloaded operator `[]` – `v[r]` returns the element at rank `r` (no index checking)
  - `at(r)` – returns the element at rank `r` (with index checking)
  - `push_back(e)` – insert `e` at the end of a vector
  - `vector(n)` – creates a vector of size `n`
  - `v[r]=e` is equivalent to `v.replace_at_rank(r,e)`
  - functions insert and remove from the arbitrary position are implemented with *iterators*

STL Class Vector (cont)

You should understand the concept of arrays but for software development use the STL vector because it supports vector operations:

- copy constructor
- copy assignment
- checking vector range and throwing an exception
- destructor – prevents memory leaks
- provides vector size
- requires less steps to resize it:
  vector<int> v(3); ...; v.resize(11); ...;

Iterators

- An iterator abstracts the process of scanning through a collection of elements.
- C++ iterator class
  - provides two functions: `begin()` and `end()`, which return an iterator to the first and one-past-the-last element, respectively
  - given an iterator `p`, `*p` is the element to which it refers
  - `++p` advances `p` to refer to the next element
  - `p->m` is equivalent to `(*p).m` for a class member `m`
  - `p==q` returns true if the iterators `p` and `q` refer to the same element
  - `p!=q` returns true f the iterators `p` and `q` refer to different elements

Iterators (cont)

- Iterator ADT
  - `bool hasNext() { return p!=end(); }`
  - `iterator next() { return ++p; }`
  - `void reset() { p=begin(); }`
- The Iterator ADT extends the concept of position by adding a traversal capability.
- Traversing a container c via an iterator:

```
for (auto p=c.begin(); p!=c.end(); ++p) {
   cout << *p << endl;
}
```

- It is even simpler in C++11:

```
for (auto e :  c) { cout << e << endl; }
```

Iterators (cont)

- Functions provided by the STL containers vector and list:
  - `begin()`, `end()` – returns an iterator to the first or one-past-the-last element of a container
  - `insert(p,e)` – inserts `e` before the position pointed by the iterator `p`
  - `erase(p)` – removes the element at the position pointed by the iterator `p`

Two-Dimensional Arrays

- A multidimensional array is called a matrix.
- One-dimensional array can be considered as a `1` by `n` matrix.
- Two-dimensional array $m \times n$ consists of m rows and n columns. Notice that every row is itself an array of n elements.

Example

A 2×3 matrix

a_{00}	a_{01}	a_{02}
a_{10}	a_{11}	a_{12}

A position of an element can be accessed by two indexes: row index and column index.

Two-Dimensional Arrays (cont)

Example

Two-dimensional arrays:

```
int array[max_rows][max_cols]; // declared array

int table[2][2] = {{5, 6}, {6, 5}}; // initializer list

enum carType {FORD, GM, TOYOTA, HONDA, NISSAN, VOLVO};

enum color {RED, BROWN, WHITE, GOLD, SILVER};

int inStock[6][5];
inStock[VOLVO][SILVER] = 5; //indexed by enum constants
```

Two-Dimensional Arrays (cont)

- Different ways of creating two-dimensional dynamic arrays.
  - Declare a one-dimensional array of pointers and initialize each pointer with the base address of another one-dimensional array.

```
int *table[10]; // fixed-size array of pointers
for (int row = 0; row < 10; ++row)
   table[row] = new int[15]; // 10x15 array
```

  - Declare an array to be a pointer to an array of pointers to `int`. The number of rows and columns can be specified during the program execution.

```
int **table;
table = new (int *)[10]; // array of pointers
for (int row = 0; row < 10; row++)
   table[row] = new int[15];  // 10x15 array
...
table[2][3] = -2;  // random access
```

Two-Dimensional Arrays (cont)

- Initializing an array

```
for (int row = 0; row < max_rows; row++) {
   for (int col = 0; col < max_cols; col++) {
      array[row][col] = row+col;
   }
}
```

A Class Matrix

- A Matrix is not a built-in type, but a user-defined class in C++.
- It is possible to define a matrix class as `vector<vector<T>>` where `T` is any type in C++ but typically `T=int` or `T=double`.
- There is a `Matrix` class written in the textbook "*Programming Principles and Pratice Using C++*", chapter 24.

```
#include "Matrix.h"
using namespace Numeric_lib;
int main()
{  // initialization is available only in C++11
   Matrix<double,2> m2 { // 2-dim matrix (3x4)
      {0,1,2,3}, //row 0
      {4,5,6,7}, //row 1
      {8,9,10,11} //row 2
   };
   cout << ‘‘matrix elem(1,2)=’’ << m2(1,2) << endl;
}
```

Position ADT

- The Position ADT models the notion of a place within a data structure where a single element is stored.
- A special `null` position refers to no element.
- The Position ADT provides a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Member functions of the Position class:
  - `Type& element()`: returns a reference to the element stored at this position
  - `bool isNull()`: returns true if this is a `null` position

List ADT

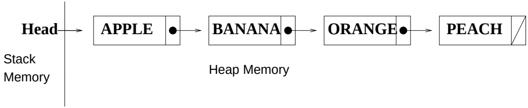
- The List ADT models a sequence of positions storing arbitrary elements.
- It establishes a *before/after* relation between positions.
- Generic methods: `size()`, `is_empty()`
- Query methods (`p` = position): `is_first(p)`, `is_last(p)`
- Accessor methods:

  ```
  first(), last(),
  before(p), after(p)
  ```

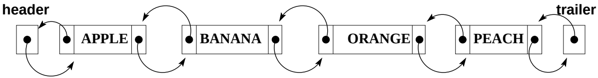
- Update methods (`p`, `q` = positions, `e` = element):
  - `replace_element(p, e)`, `swap_elements(p, q)`
  - `insert_before(p, e)`, `insert_after(p, e)`,
  - `insert_first(e)`, `insert_last(e)`, `remove(p)`

List Implementations

- Singly linked list (called also forward list)



- Doubly linked list



STL Class List

- Operations in the class list
  - `size()`
  - `front()`, `back()`
  - `push_front(e)`, `push_back(e)`
  - `pop_front()`, `pop_back()`
  - Constructor: `list()`
  - Insert and delete functions for inserting and removing from a given position of a list use iterators.

Sequence ADT

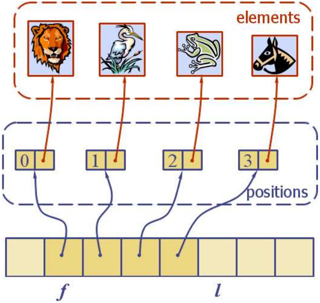
- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - rank, or
  - position
- Generic methods: `size()`, `is_empty()`
- Vector-based methods:
  - `elem_at_rank(r)`, `replace_at_rank(r, e)`, `insert_at_rank(r, e)`, `remove_at_rank(r)`
- List-based methods:
  - `first()`, `last()`, `before(p)`, `after(p)`, `replace_element(p, e)`, `swap_elements(p, q)`, `insert_before(p, e)`, `insert_after(p, e)`, `insert_first(e)`, `insert_last(e)`, `remove(p)`
- Bridge methods: `at_rank(r)`, `rank_of(p)`

Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - Small database (e.g., an address book)
- Indirect applications:
  - Building block of more complex data structures

Array-Based Implementation

- We use a circular array for storing positions
- A position object stores:
  - Element
  - Rank
- Indices f and l keep track of the first and last positions



Sequence Implementations

Operation	Array	List
size, is_empty	1	1
at_rank, rank_of, elem_at_rank	1	n
first, last, before, after	1	1
replace_element, swap_elements	1	1
replace_at_rank	1	n
insert_at_rank, remove_at_rank	n	n
insert_first, insert_last	1	1
insert_after, insert_before	n	1
remove	n	1