CSC 222: Computer Programming II

Spring 2004

Pointers and dynamic memory

- pointer type
- dereference operator (*), address-of operator (&)
- sorting lists of pointers
- dynamic memory allocation, new operator
- dynamic arrays and vectors
- destructors, copy constructors, assignment operators

Sorting revisited…

recall from lectures & hw4

- sorting involves some # of inspection and some # of swaps
- the amount of time required for a swap is dependent on the size of the values

```
vector<int> nums(10000);     SelectSort(nums);

vector<string> names(10000);     SelectSort(names);

vector<IRSTaxRecord> audits(10000);     SelectSort(audits);
```

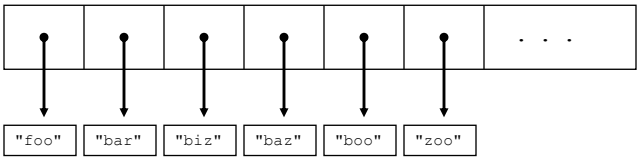
to selection sort a list of 10,000 items:

- must perform 9,999 swaps (= 29,997 assignments)
- if the items being sorted are large, then swap time can add up!

POSSIBLE WORKAROUNDS?

Lists of pointers

idea: instead of storing the (potentially large) items in the vector, store them elsewhere and keep pointers (addresses)



can sort as before

- when comparing values, have to follow the pointers to the actual data
- when swapping, copy the pointers, not the data values
  pointers are addresses (integers), so copying is fast

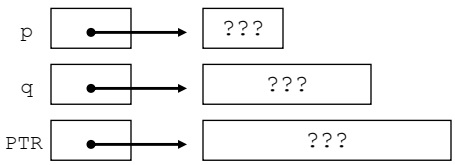
  note: since all pointers are same size, swap time is independent of data size

Pointers

a pointer is nothing more than an address (i.e., an integer)

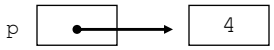
- can declare a pointer to a particular type of value using *

```
int * p;
string * q;
TYPE * PTR;
```



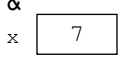
operations on pointers:

- *dereference operator:* given a pointer to some memory location, can access the value stored there using *




```
cout << *p << endl;
```

- *address-of operator:* given a memory cell, can get its address (i.e., a pointer to it) using &




```
p = &x;
```

Pointer examples

```
int * p;

int x = 6;

p = &x;

cout << *p << endl;

x++;

cout << *p << endl;

*p += 3;

cout << x;
```

using pointers and operations, can create aliases for memory locations

- in the above examples, `x` and `*p` refer to the same location

reminiscent of reference parameters?

Reference parameters and pointers

reference parameters are implemented using pointers

- when you pass an argument by-reference, you are really passing a pointer to it

implicitly,

- the address of the argument is obtained and passed in to the parameter
- each access of the parameter first dereferences the address

what you write…

```
void Foo(int & x)
{
    cout << x << endl;
}

-----------------------------------

int a = 3;
Foo(a);
```

what C++ sees…

```
void Foo(int * x)
{
    cout << *x << endl;
}

-----------------------------------

int a = 3;
Foo(&a);
```

Tracing code…

```
template <class Type> void Swap(Type & x, Type & y)
{
    Type temp = x;
    x = y;
    y = temp;
}


int a = 5, b = 9;
Swap(a, b);
cout << a << " " << b << endl;


int * p1 = &a;
int * p2 = &b;
cout << a << " " << b << endl;


Swap(*p1, *p2);
cout << a << " " << b << endl;
cout << *p1 << " " << *p2 << endl;


Swap(p1, p2);
cout << a << " " << b << endl;
cout << *p1 << " " << *p2 << endl;
```


Reimplementing selection sort

```
template <class Comparable>
void SelectionSortPtr(vector<Comparable *> & nums, int low, int high)
{
    for (int i = low; i <= high-1; i++) {
        int indexOfMin = i;
        for (int j = i+1; j <= high; j++) {
            if (*(nums[j]) < *(nums[indexOfMin])) {
                indexOfMin = j;
            }
        }

        Comparable * temp = nums[i];
        nums[i] = nums[indexOfMin];
        nums[indexOfMin] = temp;
    }
}
```

the vector contains pointers to comparable values

when comparing values, dereference (note: must parenthesize since `*` has higher precedence than `[]`)

when swapping, swap pointers, not values

```
template <class Comparable> void SelectionSortPtr(vector<Comparable *> &
    nums)
{
    SelectionSortPtr(nums, 0, nums.size()-1);
}
```

Reimplementing merge sort

```
template <class Comparable> void MergePtr(vector<Comparable *> & nums, int low, int high)
{
    vector<Comparable *> copy;
    int size = high-low+1, middle = 1 + (low+high)/2;  // middle rounds up if even
    int front1 = low, front2 = middle;
    for (int i = 0; i < size; i++) {
        if (front2 > high || (front1 < middle && *(nums[front1]) < *(nums[front2]))) {
            copy.push_back(nums[front1]);
            front1++;
        }
        else {
            copy.push_back(nums[front2]);
            front2++;
        }
    }
    for (int k = 0; k < size; k++) {
        nums[low+k] = copy[k];
    }
}

template <class Comparable> void MergeSortPtr(vector<Comparable *> & nums, int low, int high)
{
    if (low < high) {
        int middle = (low + high)/2;
        MergeSortPtr(nums, low, middle);
        MergeSortPtr(nums, middle+1, high);
        MergePtr(nums, low, high);
    }
}

template <class Comparable> void MergeSortPtr(vector<Comparable *> & nums)
{
    MergeSortPtr(nums, 0, nums.size()-1);
}
```

again, vector stores pointers

dereference before comparing

swap pointers, not values

Dynamic memory

so far, we have only considered pointers to existing (static) memory

- when you declare a variable, the compiler automatically allocates memory for it and reclaims that memory location when the lifetime ends

we can also use pointers to access dynamically-allocated memory

- i.e., memory locations that are explicitly created/destroyed on user demand

the `new` operator returns a pointer to a new, dynamically-allocated memory location

```
int * iptr = new int;
*iptr = 4;
cout << *iptr << endl;


string * sptr = new string;
*sptr = "foo";
cout << *p << endl;
```

Testing the sorts

```
#include <iostream>
#include <vector>
#include <string>
#include "Sorts.h"
using namespace std;

int main()
{
    vector<string *> words1, words2;
    string str;
    while (cin >> str) {
        string * sptr = new string;
        *sptr = str;

        words1.push_back(sptr);
        words2.push_back(sptr);
    }

    MergeSortPtr(words1);
    for (int i = 0; i < words1.size(); i++) {
         cout << *(words1[i]) << endl;
    }

    SelectionSortPtr(words2);
    for (int i = 0; i < words2.size(); i++) {
         cout << *(words2[i]) << endl;
    }

    return 0;
}
```

here, the test program:

- creates two vectors of string pointers
- reads and stores words in the vectors

  note: only creates one copy of the word, has two different pointers to it

- sorts the vectors and displays the contents

note: new requires a class constructor

here, the default constructor for string (with no arguments is used)

could have used 1-argument constructor

```
string * sptr = new string(str);
```


Vector class implementation

recall: vector is not a built-in type in C++

- it is a class, defined in the `<vector>` library

C++ has a built-in array type, which is the underlying data structure for vector

- like vector, an array is a contiguous sequence of items, accessible via indexing

```
int nums[100];

for (int i = 0; i < 100; i++) {
    nums[i] = 0;
}
```

unlike vectors, arrays:

- do not have size info associated with them (you must keep track yourself)
- the size of the array must be known at compile time (& stays fixed)
- no bounds-checking is done for arrays
- when you pass an array by-value, the array elements can still be changed!

WHY?

Arrays as pointers

inutitively, we think of an array as a contiguous sequence of locations

```
int nums[100];
```

technically, a C++ array is implemented as a pointer to the sequence

to access `nums[i]`:

- follow the `nums` pointer to `nums[0]`, then
- move ahead `i` locations in memory

Arrays as parameters

recall how a value parameter works:

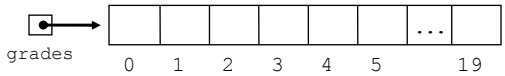
- a copy of the argument is stored in the parameter
- changes to the parameter only affect the copy

but what if the parameter is a pointer (e.g., an array)?

```
void Init(int nums[], int size)
{
    for (int i = 0; i < size; i++) {
        nums[i] = 0;
    }
}

-------------------------------------

int grades[20];
Init(grades, 20);
```



note: this is equivalent to:

```
void Init(int * nums, int size)
```

since an array is really a pointer, passing an array by-value still allows for access to the original array elements

Dynamic arrays

the pointer nature of arrays gives us a way around the compile-time size restriction

- ordinary declaration: `int nums[20];`
  creates the `nums` pointer , allocates space for 20 ints, and makes `nums` point to it

- can separate the pointer declaration and memory allocation steps

```
int * nums;
nums = new int[20];
```

creates the nums pointer (to an int)
allocates space for 20 ints and points `nums` to it

the new operator "dynamically" allocates storage, size can be determined during execution

```
cout << "How many numbers? ";
cin >> size;

int * nums;
nums = new int[size];
```


Vectors and dynamic arrays

the underlying data structure of a vector is a dynamically-allocated array

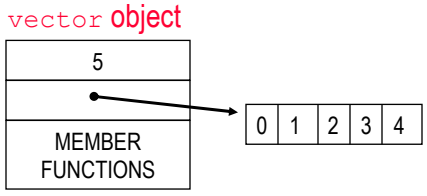
```
template <class Type>
class vector
{
  public:
    vector(int size = 0)
    {
        vecLength = size;
        vecList = new Type[size];
    }

    int size()
    {
        return vecLength;
    }

    Type & operator[](int index)
    {
        return vecList[index];
    }

    // OTHER MEMBER FUNCTIONS

  private:
    Type * vecList;
    int vecLength;
};
```



Resizing a vector

you can't resize an array, but you can create a new (bigger) array and copy

```
void resize(int newSize)
{
    Type * newList = new Type[newSize];      // ALLOCATE A NEW ARRAY OF DESIRED SIZE

    int numToCopy = min(newSize, vecLength); // DETERMINE HOW MANY ITEMS TO COPY
    for (int i = 0; i < numToCopy; i++) {    // AND COPY FROM vecLIST TO THE newList
        newList[i] = vecList[i];
    }

    delete[] vecList;                        // DEALLOCATE THE MEMORY IN vecList

    vecList = newList;                       // RESET vecList TO POINT TO THE NEW ARRAY
    vecLength = newSize;                     // AND RESET vecLength
}
```

note that dynamically-allocated memory is only accessible through a pointer

- if you lose the pointer, then the memory is lost (will NOT be automatically reclaimed)
  →MEMORY LEAKAGE can eventually use up all free memory

- any memory the programmer explicitly allocates using `new` must be explicitly deallocated using `delete` (or `delete[]` for an array)

List & SortedList revisited

recall the List and SortedList classes

- List : stores a vector of items
  Add operation inserts item in order
  IsStored operation uses binary search to see if item is in the vector
- SortedList : stores a vector of items
  Add operation inserts item in order
  AddFast operation adds at end
  IsStored checks to see if currently sorted
  if sorted, then perform binary search
  if not sorted, then merge sort the vector before binary search

we could improve the performance of the List and SortedList classes by storing pointers in the vector

- note: member functions are unchanged, so change is transparent to client
- underlying data structure changes, and member functions that access it

ListPtr.h

```
template <class ItemType>
class List
{
  public:
    List<ItemType>() { /* does nothing */ }

    virtual void Add(const ItemType & item)
    {
      ItemType * newPtr = new ItemType(item);
      items.push_back(newPtr);
    }

    virtual bool IsStored(const ItemType & item)
    {
      for (int i = 0; i < items.size(); i++) {
        if (item == *(items[i])) {
            return true;
        }
      }
      return false;
    }

    int NumItems() const
    {
        return items.size();
    }

    void DisplayAll() const
    {
        for (int i = 0; i < items.size(); i++) {
            cout << *(items[i]) << endl;
        }
    }
  protected:
    vector<ItemType *> items;       // storage for strings
};
```

`Add` needs to allocate space for a copy, then add a pointer to that copy to the vector

`IsStored` needs to dereference the pointers when comparing

`DisplayAll` needs to dereference the pointers when displaying

`items` stores pointers

SortedListPtr.h

```
template <class ItemType>
class SortedList : public List<ItemType>
{
  public:
    SortedList<ItemType>()
    {
      isSorted = true;
    }

    void Add(const ItemType & item)
    {
      ItemType * newPtr = new ItemType(item);
      items.push_back(newPtr);

      int i;
      for (i = items.size()-1;
           i > 0 && *(items[i-1]) > item;
           i--) {
          items[i] = items[i-1];
      }
      items[i] = newPtr;
    }

    void AddFast(const ItemType & item)
    {
      ItemType * newPtr = new ItemType(item);
      items.push_back(newPtr);
      isSorted = false;
    }
```

```
    bool IsStored(const ItemType & item)
    {
      if (!isSorted) {
        MergeSortPtr(items);
        isSorted = true;
      }

      int left = 0, right = items.size()-1;
      while (left <= right) {
        int mid = (left+right)/2;
        if (item == *(items[mid])) {
            return true;
        }
        else if (item < *(items[mid])) {
            right = mid-1;
        }
        else {
            left = mid + 1;
        }
    }
    return false;
  }

private:
    bool isSorted;
};
```

Timing the new SortedList

```
. . .

SortedList<string> slist1, slist2;

start = clock();
for (int i = 0; i < listSize; i++) {
    slist1.Add(randomData[i]);
}
found = slist1.IsStored("aaa");
stop = clock();

cout << "Simple sort/search required " << stop-start
     << " milliseconds" << endl;


start = clock();
for (int i = 0; i < listSize; i++) {
    slist2.AddFast(randomData[i]);
}
found = slist2.IsStored("aaa");
stop = clock();

cout << "Modified sort/search required " << stop-start
     << " milliseconds" << endl;

. . .
```

using random 20-letter words:

List size: 1000
Simple w/ words: 1623
Modified w/ words: 390

Simple w/ pointers: 1132
Modified w/ pointers: 100

List size: 2000
Simple w/ words: 6650
Modified w/ words: 841

Simple w/ pointers: 4556
Modified w/ pointers: 231

With great power…

when you dynamically allocate memory using new, you are circumventing the standard memory management of C++

- you must explicitly `delete` any memory you `new`
  e.g., if we added a `Remove` method to `List` & `SortedList`, must delete space

```
void Foo()
{
    int * ptr = new int;
    *ptr = 876;

    ...

    delete ptr;
}
```

pointers are statically allocated values – when lifetime ends, memory for pointer is reclaimed

delete reclaims the dynamic memory, not the pointer

```
void Foo()
{
    SortedList words;
    words.Add("foo");
    words.Add("bar");
    words.Add("biz");

    ...
}
```

if a class object contains dynamically allocated memory, how/when does it get reclaimed?

Destructor

classes with dynamic memory require an additional member function called a *destructor* (the antithesis of a *constructor*)

- a destructor has the same name as the class, only preceded with ~
- it is automatically called when the lifetime of a class object ends
- should contain code for reclaiming dynamically allocated memory to avoid leakage

```
~List<ItemType>()
// destructor: deletes any remaining items
//   (automatically called when the lifetime
//    of the List item ends)
{
    for (int i = 0; i < items.size(); i++) {
        delete items[i];
    }
    items.clear();   //not nec., but OK
}
```

with a derived class, the destructor for that class (if any) is called first, then the destructor for the parent class (if any)

→ `SortedList` doesn't need a destructor since the `List` destructor does it all!

But wait, there's more…

there are times when you want to make a copy of an object

- value parameters
- assignments

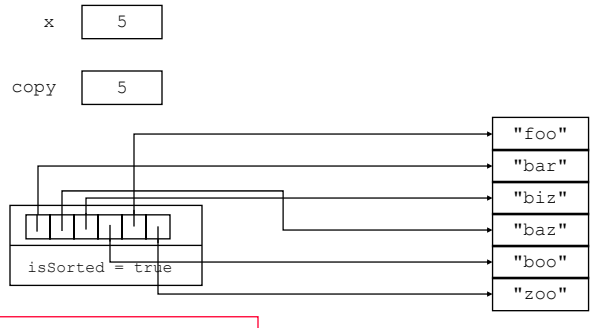
by default, an object is copied bit-by-bit

```
int x = 5;

int copy;
copy = x;

SortedList words;

SortedList copy;
copy = words;
```



what happens? problems?

Copy constructor

by default, if you copy a "dynamic" object, you only copy the pointers

- both structures point to the same memory
- if one alters the contents, it affects the other!
- if one reaches end of lifetime, its destructor will destroy the other one's data!

a *copy constructor* is a constructor that takes another object of the same type as argument, and creates a new copy

```
List<ItemType>(const List<ItemType> & other)
// copy constructor, creates a copy of other
{
  items.clear();
  for (int i = 0; i < other.items.size(); i++) {
    Add(*(other.items[i]));
  }
}
```

note: `SortedList` copy constructor first calls `List` copy constructor, then copies `isSorted` value

```
SortedList<ItemType>(const SortedList<ItemType> & other) : List<ItemType>(other)
{
    isSorted = other.isSorted;
}
```


Assignment operator

copy constructor works for copies related to value parameters, construction

to handle assignments, must overload the = operator to work for Lists

- similar to copy constructor, but must return reference to new value

```
List<ItemType> & operator=(const List<ItemType> & other)
// assignment constructor, creates a copy of other
{
  if (this != &other) {
    items.clear();
    for (int i = 0; i < other.items.size(); i++) {
      Add(*(other.items[i]));
    }
  }
  return *this;
}
```

also, must be careful in the case where `x = x;`

```
SortedList<ItemType> & operator=(const SortedList<ItemType> & other)
{
  if (this != &other) {
    List<ItemType>::operator=(other);
    isSorted = other.isSorted;
  }
  return *this;
}
```

Dynamic memory summary

dynamic memory adds great flexibility to code

- can allocate new memory on demand, resize vectors, avoid swapping data, …

when you utilize dynamic memory, you are responsible for its deallocation and making sure copies/assignments are handled correctly

when implementing a class that utilizes dynamic memory, generally need:

- destructor: to automatically reclaim dynamic memory when the object's lifetime ends
- copy constructor: to construct copies (e.g., for value parameters)
- assignment operator: to handle assignments correctly