4.2 sorting, templates

Thursday, September 22, 2022

Reminder: test 1 in lab today

TODAY

- quicksort analysis
- best, worst, expected big O runtime
- creating generic functions + classes in C++ using templates
- abstract data types (ADTs)
- lists

Quicksort

another divide-and-conquer sorting algorithm

To sort the whole array: quicksort(array, 0, size-1)

```
quickSort(array, i, j): // sorts the array[i,...,j]
        if j ≤ i:
                return // region to sort is ≤1 element, so it's definitely sorted!
        k = partition(array, i, j)  ← divide
        quicksort(array, i, k-1)  } conquer
        quicksort(array, k+1, j)  }

partition(array, left, right):
// rearranges array[left,...,right] using a pivot element
// should return the index where pivot element ends up
        pivot = right
        right --
        while (left ≤ right):
                if array[left] ≤ array[pivot]:
                        left++
                else if array[right] ≥ array[pivot]:
                        right--
                else:
                        swap(array,left,right)
        swap(array,left,pivot) // put the pivot element into its place
        return left
```

note: updated "partition" pseudocode from Tuesday's version, to be clearer.

Idea: Keep incrementing 'left' and decrementing 'right' until we find values that should be swapped.

We stop when left crosses right, so left is the index of the place where the "bigger half" of the array begins.

After running partition, the array looks like:

$$\left[\underbrace{\text{all values} \le \text{pivot value}}_{\text{not yet sorted}}\right.\quad \boxed{\text{pivot value}}\quad \left.\underbrace{\text{all values} \ge \text{pivot value}}_{\text{not yet sorted}}\right]$$

Some observations:

- memory is in-place (never makes extra copies of array)
- Each call to partition takes linear time (with respect to the piece of the array it's called on).

- Overall runtime depends on choice of pivot.

What would be the best pivot choice? and best runtime?

Best pivot value would be the median. $O(n \log n)$

best case

Q: What would be the worst pivot choice? and worst runtime?

Worst pivot value would be the smallest/largest.

After pivoting, $[\underbrace{\sim\sim\sim\sim}_{\text{unsorted}} \quad \square_{\text{pivot}}]$

worst case:

$O(n^2)$

In the worst case, always having the worst pivot results in linearly many levels of recursion.

Q: How can we adjust the "partition" function to try to avoid the worst case?

```
partition(array, left, right):
// rearranges array[left,...,right] using a pivot element
// should return the index where pivot element ends up
    pivot = randomly choose an index in the set {left, left +1 , ... , right}
    swap(array,right,pivot)
    right--
    while (left ≤ right):
        if array[left] ≤ array[pivot]:
            left++
        else if array[right] ≥ array[pivot]:
            right--
        else:
            swap(array,left,right)
    swap(array,left,pivot) // put the pivot element into its place
    return left
```

changed choice of pivot index

expected runtime $O(n \log n)$

("Expected" over the random choices the algorithm makes — some runs might be faster or slower compared to each other, but overall we have expected $O(n \log n)$.)

In practice, usually quicksort is faster than mergesort.

(But in worst case analysis, mergesort is faster than quicksort.)

mergesort vs quicksort

similarities

- use recursion
- best case $O(n \log n)$
- divide & conquer technique

differences

- quicksort in-place, doesn't need additional memory
- worst case quicksort $O(n^2)$
  mergesort $O(n \log n)$
- quicksort uses pivot element to divide array, won't always be exactly in half
- quicksort does the interesting part during divide
  mergesort does the interesting part during conquer

Generic functions & classes

In python we could write one function:

```
def min(a, b):
    if a < b:
        return a
    else:
        return b
```

. . . and use it for many different types:

min(1,3) ⟶ 1

min(4.1, 3.4) ⟶ 3.4

min("hello", "bye") ⟶ "bye"

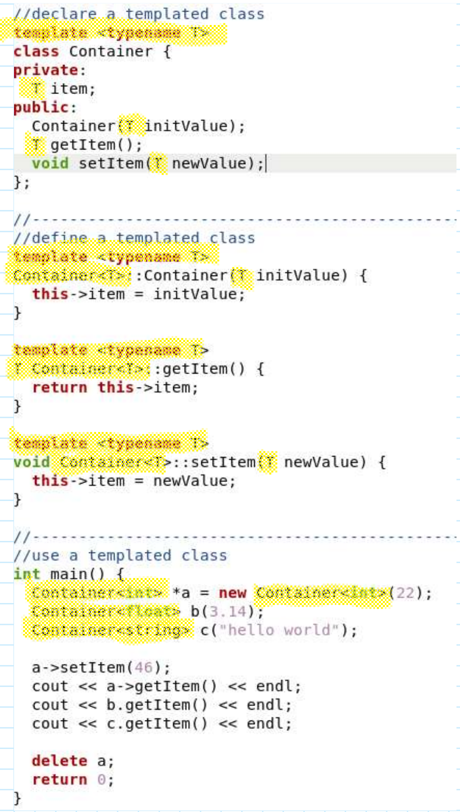
But to do this in C++ we need to specify the return type & parameter types, so this would require 3 different functions:

```
int intMin(int a, int b) { ... }
float floatMin(float a, float b) { ... }
string stringMin(string a, string b) { ... }
```

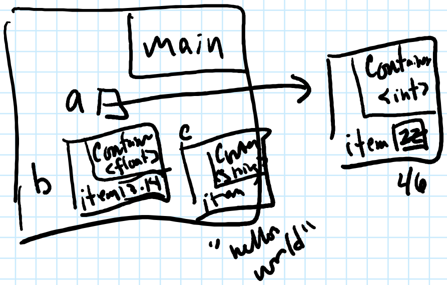
C++ allows us to use TEMPLATES to define generic functions:

```
template <typename T>
T genericMin(T a, T b) {
        if (a <=b) {
                return a;
        }
        else {
                return b;
        }
}

int main() {
        string x = "hello";
        string y = "bye";
        cout << "min of 3 and 4 is " << genericMin<int>(3,4) << endl;
        cout << "min of 3.4 and 1.2 is " << genericMin<float>(3.4,1.2) << endl;
        cout << "min of hello and bye is " << genericMin<string>(x,y) << endl;

        // or the compiler looks at the argument types to the generic function and determines which type
        to replace T with --- works for basic types but for classes we'll need to specify
        cout << "testing the automatic type detection" << endl;
        cout << "min of 3 and 4 is " << genericMin(3,4) << endl;
        cout << "min of 3.4 and 1.2 is " << genericMin(3.4,1.2) << endl;
        cout << "min of hello and bye is " << genericMin(x,y) << endl;

}
```

C++ allows us to use TEMPLATES to define generic classes, too:



```
//declare a templated class
template <typename T>
class Container {
private:
  T item;
public:
  Container(T initValue);
  T getItem();
  void setItem(T newValue);
};

//--------------------------------------------------
//define a templated class
template <typename T>
Container<T>::Container(T initValue) {
  this->item = initValue;
}

template <typename T>
T Container<T>::getItem() {
  return this->item;
}

template <typename T>
void Container<T>::setItem(T newValue) {
  this->item = newValue;
}

//--------------------------------------------------
//use a templated class
int main() {
  Container<int> *a = new Container<int>(22);
  Container<float> b(3.14);
  Container<string> c("hello world");

  a->setItem(46);
  cout << a->getItem() << endl;
  cout << b.getItem() << endl;
  cout << c.getItem() << endl;

  delete a;
  return 0;
}
```



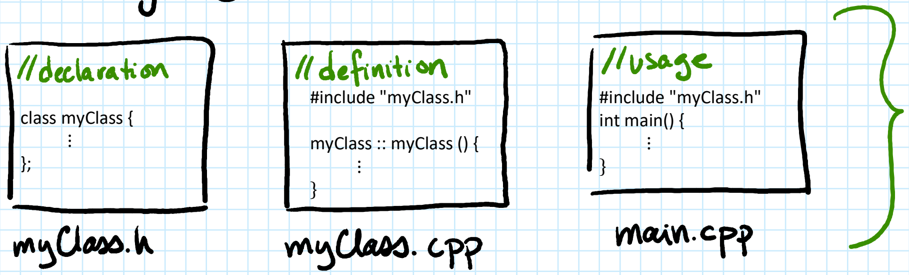
printed:

46

3.14

hello world

File organization for a STANDARD class:



```
//declaration
class myClass {
    ⋮
};
```

myClass.h

```
//definition
#include "myClass.h"

myClass :: myClass () {
    ⋮
}
```

myClass.cpp

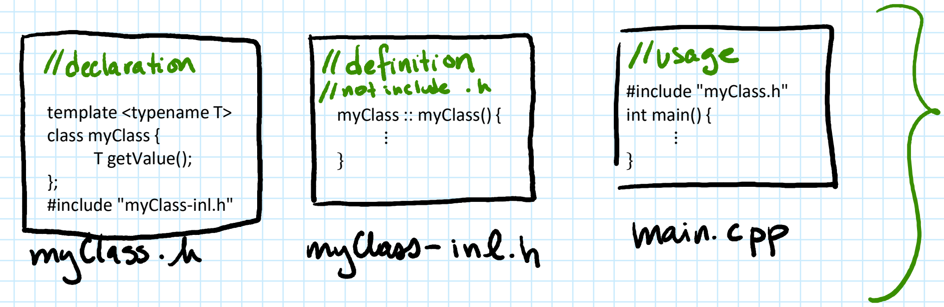
```
//usage
#include "myClass.h"
int main() {
    ⋮
}
```

main.cpp

for example, see

/public/fontes/cs35/week4/templateClass

File organization for a TEMPLATED class:



```
//declaration
template <typename T>
class myClass {
        T getValue();
};
#include "myClass-inl.h"
```

myClass.h

```
//definition
//not include .h
myClass :: myClass() {
    ⋮
}
```

myClass-inl.h

```
//usage
#include "myClass.h"
int main() {
    ⋮
}
```

main.cpp

for example, see

/public/fontes/cs35/week4/templateClassSeparated

ABSTRACT DATA TYPES (ADTs)

We want our data structures to work with any type of data.

An ADT provides a high-level overview of what a data structure can do.

- uses templates
- implementation details of the data structure are ignored at this level
- purely abstract superclass (cannot be instantiated, has no constructor)

If you want to implement an ADT, you must create a subclass.

LIST : an ordered sequence of elements all of the same type

Why bother with lists when we have arrays?

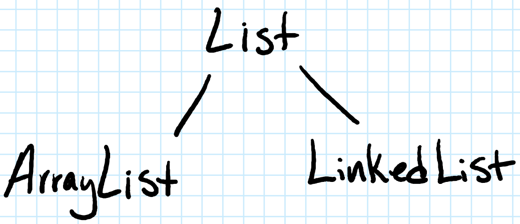
lists can know their own length (arrays don't)
lists can use templates, so we can have many different types of list
avoid out-of-bounds indexing
lists have no fixed size

} more user-friendly

declaration of List ADT:

```
template <typename T>
class List {
    public:
            virtual ~List(); // destructor
            virtual int getSize() = 0;
            virtual void insertFirst(T item) = 0;
            virtual void insertLast(T item) = 0;
            virtual T removeFirst() = 0;
            virtual T removeLast() = 0;
            virtual T get(int index) = 0; // return the element at index "index"
            virtual T getFirst() = 0;
            virtual T getLast() = 0;
            virtual bool isEmpty() = 0; // return true if the list is empty


};
```

We will implement List in two ways:



Assuming we will have an implementation, let's trace some code:

```
LinkedList<int> numbers;
numbers.insertFirst(2);
numbers.insertFirst(6);
numbers.insertFirst(3);
```