Practicum vrijdag: je eigen vector

Collapse context

We maken een implementatie van het ADT Lijst voor integers genaamd ArrayList. Een array heeft een vaste lengte. In deze oefening maken we die dynamisch, gelijkaardig aan de C++ `vector` die intern ook een array van vaste lengte gebruikt, maar als die vol zit, zichzelf gaat vergroten.

Alle functies, behalve createList, krijgen een pointer naar een list mee als eerste parameter.

Your answer passed the tests! Your score is 100.0%. [Submission #67446b5bde404ce80b3b76ee (**2024-11-25 12:19:39**)]

Question 1: De data

✓ Perfect

Definieer een `ArrayList` als een `struct` met drie velden: `size`, `capacity` en `items`. `capacity` is de grootte van de onderliggende array. `size` is de het aantal opgeslagen objecten in de lijst. `items` is een pointer naar een int (waar de array gaat komen). Wanneer de array vol is, wordt er een nieuwe array aangemaakt met het dubbele van de grootte.

```
1  struct ArrayList{
2      int size;
3      int capacity;
4      int* items;
5  };
```

Submit

Question 2: createList(initialCapacity)

✓ Perfect

Schrijf een functie `createList` die een initiële capaciteit meekrijgt. De functie geeft een pointer terug naar een lege `ArrayList` waarbij het geheugen dynamisch werd aangemaakt. De struct uit vraag 1 wordt automatisch geïncludeerd.

```
1  ArrayList* createList(int c){
2      ArrayList *a = new ArrayList;
3      int* i = new int[c];
4      a->items = i;
5      a->capacity = c;
6      a->size = 0;
7
8      return a;
9  }
```

Submit

Question 3: clearList(list)

✓ Perfect

Schrijf een functie `clearList` die een lijst (pointer) meekrijgt zonder return waarde. De functie maakt er een lege lijst van zonder het geheugen vrij te geven. Vanaf nu wordt de code van vraag 1 en 2 automatisch geïncludeerd. Denk eraan dat de lijst leeg is van zodra de size gelijk is aan ... wat dus niets te maken heeft met hoeveel geheugen er in gebruik is!

```
1  void clearList(ArrayList *p)
2  {
3      for (int i = 0; p->size != 0; i++)
4      {
5          p->items[i] = NULL;
6          --p->size;
7      }
8  }
```

Submit

Question 4: destroyList(list)

✓ Perfect

Schrijf een functie `destroyList` die al het geheugen vrijgeeft, d.w.z. de lijst en de items uit de lijst.

```
1  void destroyList(ArrayList *p)
2  {
3      delete[] p->items;
4      delete p;
5  }
```

Submit

Question 5: isEmpty(list)

✓ Perfect

Geeft een boolean terug die true is asa de lijst leeg is.

```
1  bool isEmpty(ArrayList *p)
2  {
3      return p->size == 0;
4  }
```

Submit

Question 6: getSize(list)

✓ Perfect

Geeft aan hoeveel items er in de lijst zitten.

```
bool getSize(ArrayList* p)
{
    return p->size;
}
```

Submit

Question 7: print(list)

✓ Perfect

Drukt de lijst af als een Python list (dus bv [1,2,3]).

```
void print(ArrayList *list)
{
    cout << '[';
    for (int i = 0; i < list->size; i++)
    {
        if (i < list->size - 1)
            cout << list->items[i] << ", ";
        else
            cout << list->items[i];
    }
    cout << ']' << endl;
}
```

Submit

Question 8: set(list,index,newItem)

✓ Perfect

Op de gegeven index wordt newItem gezet. De index gaat van 0 t.e.m. size-1. De functie geeft een boolean terug. Als een ongeldige index wordt gegeven, dan geeft die false terug, anders true.

```
bool set(ArrayList* l, int i, int item)
{
    if (i >= 0 && i < l->capacity)
    {
        l->items[i] = item;
        return true;
    }
    return false;
}
```

Submit

Question 9: get(list,index,item)

✓ Perfect

De waarde die op de gegeven index zit wordt in item gezet. De functie geeft een boolean terug. Als een ongeldige index wordt gegeven, dan geeft die false terug, anders true.

```
1  bool get(ArrayList* l, int i, int& item)
2  {
3      if (i >= 0 && i < l->capacity)
4      {
5          item = l->items[i];
6          return true;
7      }
8      return false;
9  }
```

Submit

Question 10: del(list,index)

✓ Perfect

Het item op de gegeven index wordt verwijderd en de array wordt opnieuw geïndexeerd, d.w.z. alle elementen na de gegeven index worden 1 positie naar links verplaatst. De functie geeft opnieuw een boolean terug of het gelukt is.

```
1  bool del(ArrayList *l, int i)
2  {
3      if (i >= 0 && i < l->size)
4      {
5          l->items[i] = NULL;
6
7          for (int j = i + 1; j < l->size; j++)
8          {
9              (l->items[j - 1]) = l->items[j];
10         }
11         --l->size;
12
13         return true;
14     }
15     return false;
16 }
```

Submit

Question 11: doubleCapacity(list)

✓ Perfect

Verdubbel de capaciteit. Er wordt een nieuwe array dynamisch aangemaakt met een dubbele grootte. De items in de oude array worden overgezet naar de nieuwe met behoud van index. De oude array wordt terug vrijgegeven.

```
1  void doubleCapacity(ArrayList* list)
2  {
```

```
    list->capacity *= 2;
    int* newItems = new int[list->capacity];

    for (int i = 0; i < list->size; ++i)
    {
        newItems[i] = list->items[i];
    }

    delete [] list->items;
    list->items = newItems;
}
```

Submit

Question 12: add(list,newItem)

✓ Perfect

Voegt een element achteraan de lijst toe. Als de capaciteit bereikt werd, wordt er een nieuwe array dynamisch aangemaakt met een dubbele grootte. De items in de oude array worden overgezet naar de nieuwe met behoud van index. De oude array wordt terug vrijgegeven. De functie `doubleCapacity` wordt automatisch mee geïncludeerd.

```
void add(ArrayList *list, int newItem)
{
    int capa = list->capacity;
    int size = list->size;
    if (capa != size)
    {
        list->items[size] = newItem;
        list->size++;
    }
    else
    {
        doubleCapacity(list);
        list->items[size] = newItem;
        list->size++;
    }
}
```

Submit

Question 13: insert(list, index, newItem)

✓ Perfect

Zelfde functionaliteit als add alleen wordt newItem geplaatst op de gegeven index. Eerst wordt er plaats gemaakt door alle items vanaf de gegeven index op te schuiven naar rechts en dan wordt newItem op de gegeven index gezet. Denk eraan dat ook hier de array mogelijks groter moet worden. De functie `doubleCapacity` wordt automatisch mee geïncludeerd.

```
void insert(ArrayList *list, int index, int newItem)
{
    int capa = list->capacity;
    int size = list->size;
    if (capa == 1 && size == 0)
    {
        list->items[0] = newItem;
    }
```

```
    else if (capa != size)
    {
        for (int i = size; i > index; i--)
        {
            list->items[i] = list->items[i - 1];
        }
        list->items[index] = newItem;
    }
    else
    {
        doubleCapacity(list);
        capa = list->capacity;
        for (int i = size; i > index; i--)
        {
            list->items[i] = list->items[i - 1];
        }
        list->items[index] = newItem;
    }
    ++list->size;
}
```

Submit

Question 14: loadList(filename)

✓ Perfect

Lees alle items in vanuit een file. Alle items staan op een nieuwe regel. De functie geeft een pointer terug naar een nieuwe lijst. Alle vorige code wordt mee geïncludeerd.

```
ArrayList* loadList(string filename)
{
    ifstream file{filename};

    ArrayList* newList = createList(2);

    string line = "";

    while (file >> line){
        add(newList, stoi(line));
    }

    file.close();
    return newList;
}
```

Submit

Question 15: saveList(list,filename)

✓ Perfect

Schrijf alle items weg in een file. Alle items staan op een nieuwe regel. De functie geeft niets terug.

```
void saveList(ArrayList* list, string filename)
{
    ofstream file{filename};
```

```
4
5       for (int i = 0; i < list->size; i++)
6       {
7           file << list->items[i] << endl;
8       }
9       file.close();
10  }
```

Submit

Question 16: Valgrind: testen op memory leaks

✓ Perfect

Hier moet je geen code schrijven. Al de code van de vorige oefeningen wordt geïncludeerd en de output van valgrind wordt getoond na het uitvoeren van bijgevoegde main.

```
1   int main(){
2       ArrayList* l = createList(3);
3       add(l,203);
4       print(l); //Prints [203]
5       add(l,123);
6       print(l); //prints [203,123]
7       set(l,1,204);
8       print(l); //prints [203,204]
9       int value;
10      get(l,0,value);
11      set(l,0, value - 1);
12      print(l); //prints [202,204]
13      del(l,0);
14      print(l); //prints[204]
15      clearList(l);
16      destroyList(l);
17      return 0;
18  }
```

Submit

Question 17: Extra: een stackimplementatie

✓ Perfect

Implementeer het ADT Stack op basis van de ArrayList. Intern gebruik je een ArrayList, maar je maakt functies volgens het ADT Stack. We gebruiken een typedef. Dit is een andere naam voor een bestaand datatype. Zorg dat volgende code werkt. Al de code van de vorige oefeningen wordt automatisch geïncludeerd. Je moet dus enkel de functies createStack, destroyStack, push en pop implementeren. De print hoef je niet te implementeren omdat de print van de ArrayList gebruikt wordt (het zijn immers synoniemen).

```
1   typedef ArrayList Stack;
2   // hier komt jouw code
3   Stack *createStack()
4   {
5       return createList(1);
6   }
7
8   void destroyStack(Stack *s)
9   {
10      destroyList(s);
```

```
11  }
12
13  void push(Stack *s, int item)
14  {
15      int capa = s->capacity;
16      int size = s->size;
17
18      if (capa != size)
19      {
20          s->items[size] = item;
21      }
22      else
23      {
24          doubleCapacity(s);
25          s->items[size] = item;
26      }
27      ++s->size;
28  }
29
30  int pop(Stack *s)
31  {
32      int capa = s->capacity;
33      int size = s->size;
34      int poppedItem = NULL;
35
36      if (size > 0)
37      {
38          poppedItem = s->items[size - 1];
39          s->items[size-1] = NULL;
40          --s->size;
41      }
42      return poppedItem;
43  }
44
45  int main(){
46      Stack* s = createStack();
47      push(s,203);
48      print(s); //prints [203]
49      push(s,99);
50      print(s); //prints [203,99]
51      int last = pop(s);
52      print(s); //prints [203]
53      destroyStack(s);
54  }
```

Submit