Както вече знаем, динамичните масиви могат да бъдат мощно оръжие, когато се използват по правилен начин. За жалост, в C++ динамичната памет е трудна за управление, тъй като трябва да я освобождаваме ръчно. За щастие, обаче, имаме и друго мощно оръжие - ООП, с което можем да опаковаме динамичната памет и да я направим по-лесна за употреба.

Трябва да се напише клас Vector, който да представлява обвивка, околко обикновен динамичен масив от тип int. Но това няма да е всичко. Vector трябва да позволява добавяне на нови елементи към тази динамична памет, както и премахване на вече съществуващи такива.

Как ще става разширяването/свиването? Не можем просто да свиваме и разширяваме динамично заделена памет, но можем да заделяме нова и да пренасяме данните в нея. Всеки път, когато Vector променя размера си, ще се заделя нов динамичен масив, с нужният нов размер. В него ще се копират данните от текущия. Накрая, текущият се освобождава.

Кога трябва да е случва заделянето на нова памет? Ако всеки път, когато искаме да добавим или премахнем елемент, копираме цялата информация, то тези операции ще са ужасно бавни, когато елементите станат много. Този проблем ще разрешим, като правим това копиране възможно по-малко пъти. За да постигнем това, заделената динамична памет, ще бъде с по-голям размер, от използвана в момента:

```
<----------- заделена памет --------------------------------------------------->
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
<----------- използвана памет ----------------->
```

Какво печелим? Можем да използваме остатъка от паметта, за да “вмъкваме” и “премахваме” елементи. Това, което наистина ще се случва, е, че размерът на използваната памет ще се променя, а стойностите на новите елементи ще се записват в оставащите клетки.

Какво става, като паметта свърши? Когато достигнем края - размерите на заделена и използвана памет станат равни, ще трябва да осигурим още памет. Тогава ще заделим нов динамичен масив и в началото му ще копираме елементите на текущия. Но колко по-голям да е новият? Новият масив ще бъде точно 2 пъти по-голям. Това ще отложи следващото копиране двойно повече. Колкото повече вмъкваме, толкова по-рядко ще стават копиранията, но и толкова по-бавни (нещата се балансират).

Стъпки на разширяване:

limit: 5
size: 5
data: **[1][2][6][5][3]**

1. Заделяме нов масив, който е два пъти по-голям:

   limit: 5
   size: 5
   data: **[1][2][6][5][3]**

   newLimit: 10

newData: [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]

2. Копираме елементите:
   limit: 5
   size: 5
   data: **[1][2][6][5][3]**

   newLimit: 10
   newData: **[1][2][6][5][3]**[ ][ ][ ][ ][ ]

3. Изтриваме старата памет и взимаме новата:
   size: 5
   limit: newLimit: 10
   data: newData: **[1][2][6][5][3]**[ ][ ][ ][ ][ ]

А кога да свиваме паметта? Бихме могли да свиваме паметта, когато използваните елементи, станат два пъти по-малко от заделените. Тогава "режем" половината памет. Но при често добавяне и премахване на елементи, Vector постоянно ще се свива и разширява.
Друг подход е отново да свием размера двойно, когато използваните елементи станат четири пъти по-малко. По този начин, след свиването ще останат достатъчно празни клетки (половината), в които да изпълняваме добавяне и премахване бързо.

Стъпки на свивне:
   limit: 16
   size: 4
   data: **[1][2][6][5]**[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]

1. Заделяме нов масив, който е два пъти по-малък:
   limit: 16
   size: 4
   data: **[1][2][6][5]**[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]

   newLimit: 8
   newData: [ ][ ][ ][ ][ ][ ][ ][ ]

2. Копираме елементите:
   limit: 16
   size: 4
   data: **[1][2][6][5]**[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]

   newLimit: 8
   newData: **[1][2][6][5]**[ ][ ][ ][ ]

3. Изтриваме старата памет:
   size: 4
   limit: newLimit: 8

data: newData: [1][2][6][5][ ][ ][ ][ ]

След като се запознахме с по-сложната част, нека напишем самия клас:

1. Vector(int = 10) - конструктор с начален limit.
2. Копиращ конструктор
3. Деструктор
4. Оператор за присвояване

5. int& operator [] (int)
6. int operator[] (int) const
7. Vector& insert(int elem) - добавя нов елемент в края
8. Vector& remove() - премахва последния елемент
9. Vector& insertAt(int elem, int index) - добавя елемент на желаната от нас позиция, като отмества другите елементи с позиция надясно
10. Vector& removeAt(int index) - премахва елементът на желаната позиция, като отмества останалите с позиция наляво

По-нагоре обсъдихме разширяването и свиването на Vector. Нека разгледаме добавянето и премахване на елементи.

Добавяне:

elem: 8
index: 1
limit: 6
size: 4
data: [1][2][6][5][ ][ ]

1. Ако е няма повече място, разширяваме.

2. Присвояваме в посока надясно, за да освободим мястото за новият елемент:
   elem: 8
   index: 1
   limit: 6
   size: 5
   data: [1][2][2][6][5][ ]

3. Записваме стойността в клетката:
   elem: 8
   index: 1
   limit: 6
   size: 5
   data: [1][8][2][6][5][ ]

Премахване:

index: 2
limit: 6
size: 5
data: [1][8][2][6][5][ ]

1. Присвояваме в посока наляво:
   index: 2
   limit: 6
   size: 5
   data: [1][8][6][5][5][ ]

2. Намаляваме size:
   index: 2
   limit: 6
   size: 4
   data: [1][8][6][5][5][ ]

3. Ако е небходимо, свиваме размера на динамичната памет.

Добавянето и премахването в края, става по идентичен начин.

Така написан, класът Vector може да бъде използван, като най-обикновен динамичен масив, като при създаването му подаваме желаният брой елементи, а при използването му, никога не вмъкваме повече от този брой.