Dynamic Memory

Allocate memory as the program is running.

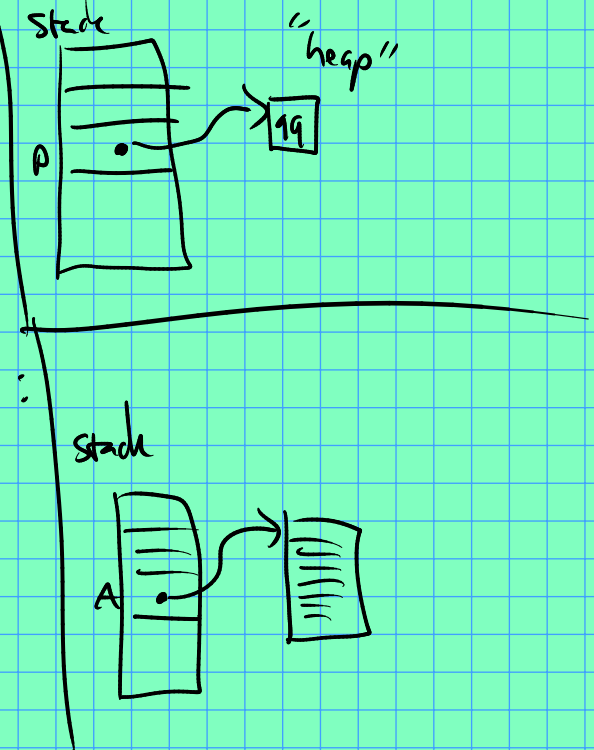
```
int x;  // allocated on stack
        // x is referenced by an
        // offset from the stack pointer.
```

Note: this offset is computed at compile time.

Any variable whose size is not known @ compile time could break this scheme.

In C++, dynamic allocations are done with the new operator. E.g.:

```
int * p = new int;
*p = 99;
```



You can also allocate arrays:

```
int* A = new int[10];
```


Remember: You must deallocate whatever you allocated w new!

This is done via delete:

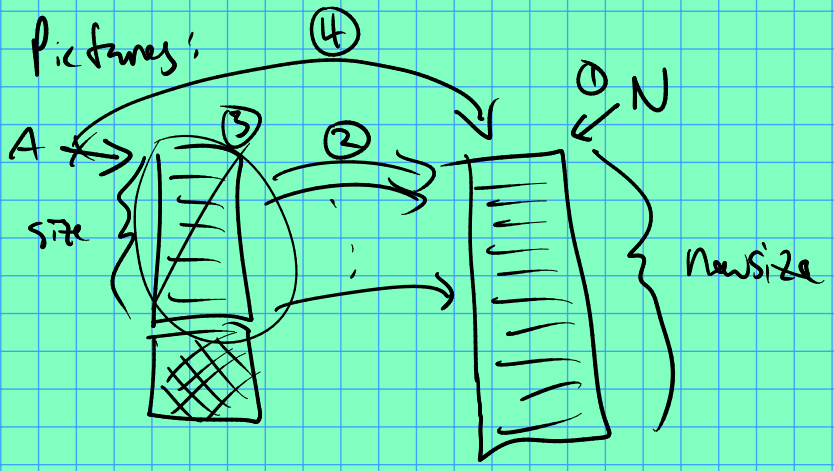
```
int * p = new int;
        :
delete p;

int * A = new int[10];
        :
delete[] A;
```

Exercise: resize an array.
(Think of how vectors expand when necessary...)

```
void resize(int*& A; int size; int newsize);
```

Pictures:




```
void resize(int*& A; int size; int newsize) {
    int * N = new int[newsize]; // ①
    for (int i=0; i< size; i++)
        N[i] = A[i];            // ②
    delete[] A;                 // ③
    A = N;                      // ④
}
```

We could use the above to help with implementing a vector: this is how to handle push_back() when the vector's array is already full.

Speaking of push_back: how to choose the new size when you reach the capacity?

size++? ①

size *= 2? ②

Using ①, what is the approx. cost of n consecutive calls to push_back()?

```
vector<int> V;
for(i=0; i<n; i++) V.push_back(i*i);
```

as a function of n, what is the approx. cost?

Total cost $\geq 1+2+3+4+\cdots+n$

$= \frac{n(n+1)}{2} \geq n^2/2.$

What about using strategy ② (size *= 2)?

1 → 2

2 → 4

4 → 8

8 → 16

16 → 32

With ②, only need $\simeq \log_2 n$ calls to resize.

Obvious bound on the # of steps:

$n + n\log_2 n$

But actually it is closer to $2n$...

Exercise →