072-lec-09

Syntax corner:

--- Constructors (default, copy, shallow copy, deep copy)
--- Operators
--- Global functions, objects
--- static

-------- Default constructor

```
 class F {
     public:
         F();
 }
```

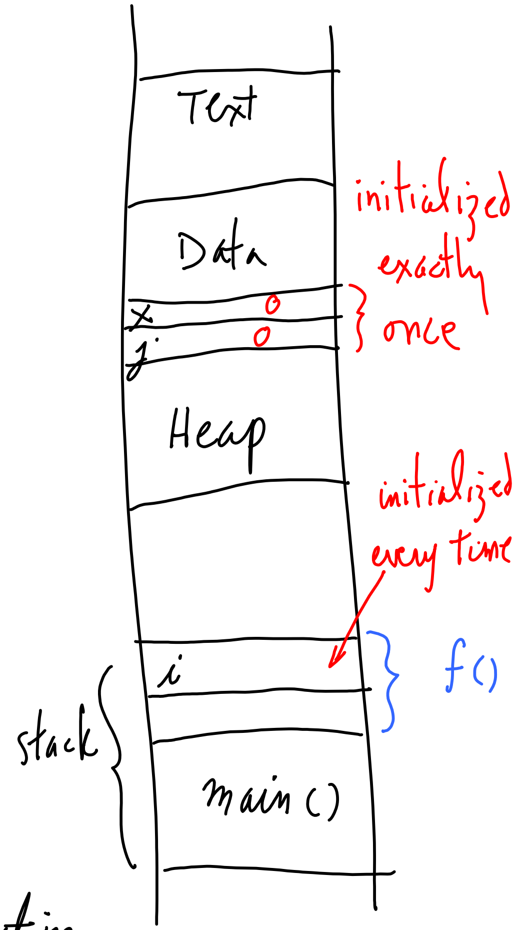
```
int *p = new int[10];
delete [ ] p;                  //--- deletes all of the array
```

static variable, local scope

```
f()
{
    static int x = 0;
    for ( int i = 0; i < x; i++)
    {
        static int j = 0;
        j++;
    }
    x++;
}

main()
{
    while(1) { f() }
}
```

x tells us how many times f() has been called.

j tells us the total loop iterations overall.



x incremented once for each call to f().

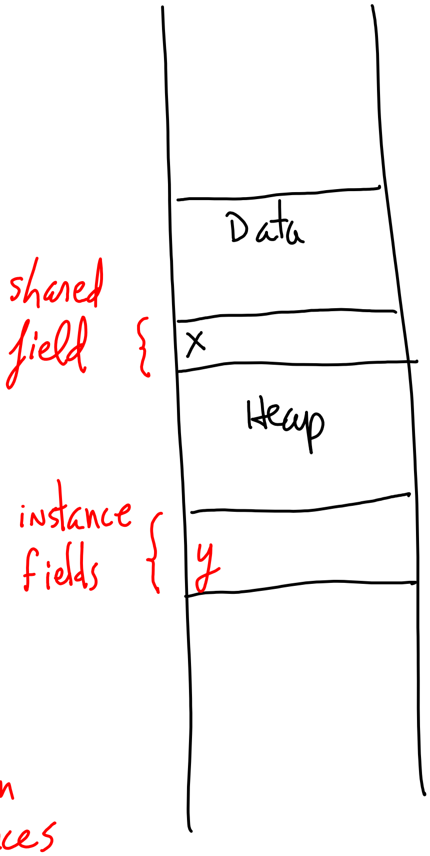
j incremented every loop iteration.

static field in class

```
class F{
    static int x;
    int y;
    void foo();
};

void F::foo(){
    x++;
}

int F::x = 10;

int main(){
    cout << F::x ;
}
```

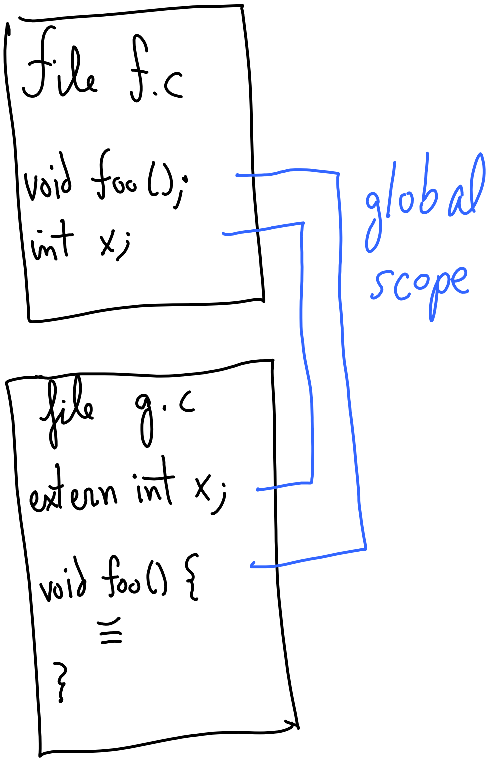
```
Bunny : Thing {
    static numBunnies = 0;
}
Bunny::Bunny() {
    numBunnies++;
Bunny::~Bunny() {
    numBunnies--;
}
```

static methods, same syntax. Cannot use non-static fields.



X exists even when there are no instances of FOO

static variable, global scope



```
file f.c
void foo();
int x;
```

```
file g.c
extern int x;
void foo() {
    ≡
}
```

global scope

```
//--------------
//--- file f.c
static int x;
int y;
//------ end file f.c
```

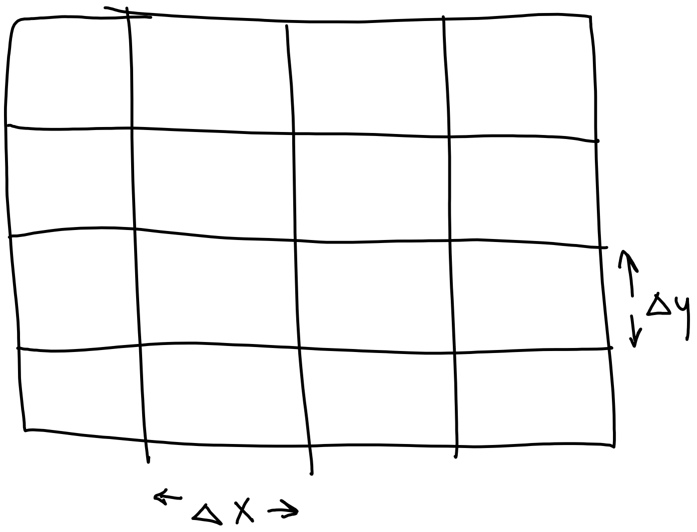
x is global only for file f.c

y is global

```
//----------
//--- file g.c
extern int y;
y =10;
//----- end file g.c
```

static global functions have file scope.

Environment



fix Δx, Δy

what's in a cell?

Some things:

Grass
Bunny
Fox
Carrot
⋮

Sounds like a vector?

How many per cell?

```
#include <vector>

vector<thing*>
```

<— type for a cell

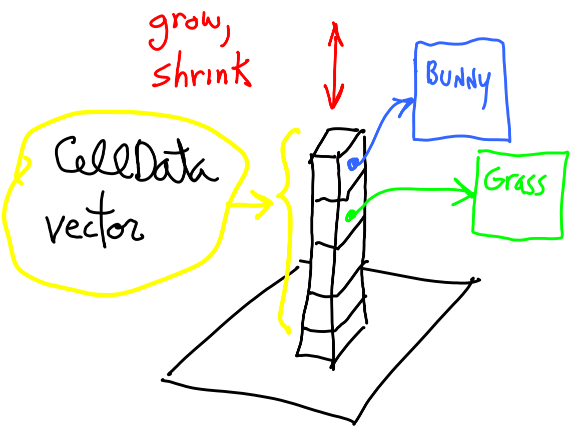
copy constructor

```
vector<int> V;      //-- empty
vector<int> V(4);   //-- 4 elements
                    //-- V(4) is same as
                    V = 4;
```

not assignment!

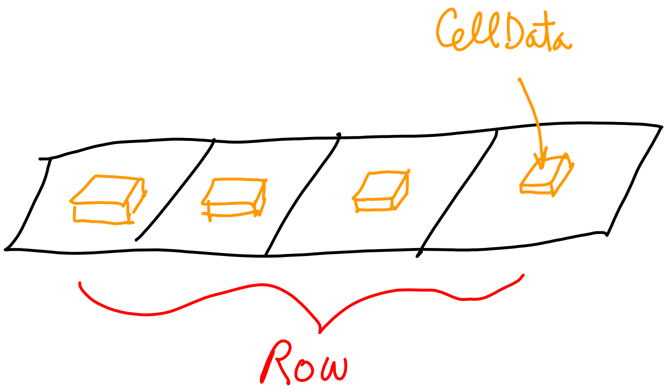
```
vector<int> V(4, 0)
```

initial value, copied 4 times ⇒ copy constructor



```
typedef vector<Thing *> CellData;
```

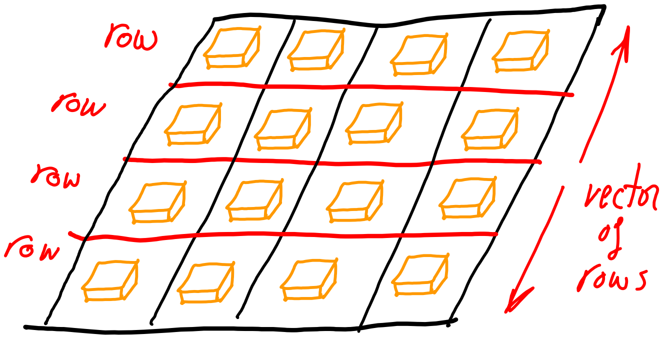
//-- If we get one, it should be empty, right?



```
CellData emptyCD;

vector<CellData> row(4, emptyCD);
```

Type of rows

//-- Let's try same trick again.



```
vector< vector<CellData> > field(4, row);
```

accessing?

```
Thing *p;

p = (Thing*) new Bunny;

((field.at(1)).at(1)).push_back(p);
```

//-- Look cleaner?

```
typedef vector<Thing *> CellData;
typedef vector< CellData> Row;
typedef vector< Row > Field;
```

unique IDs

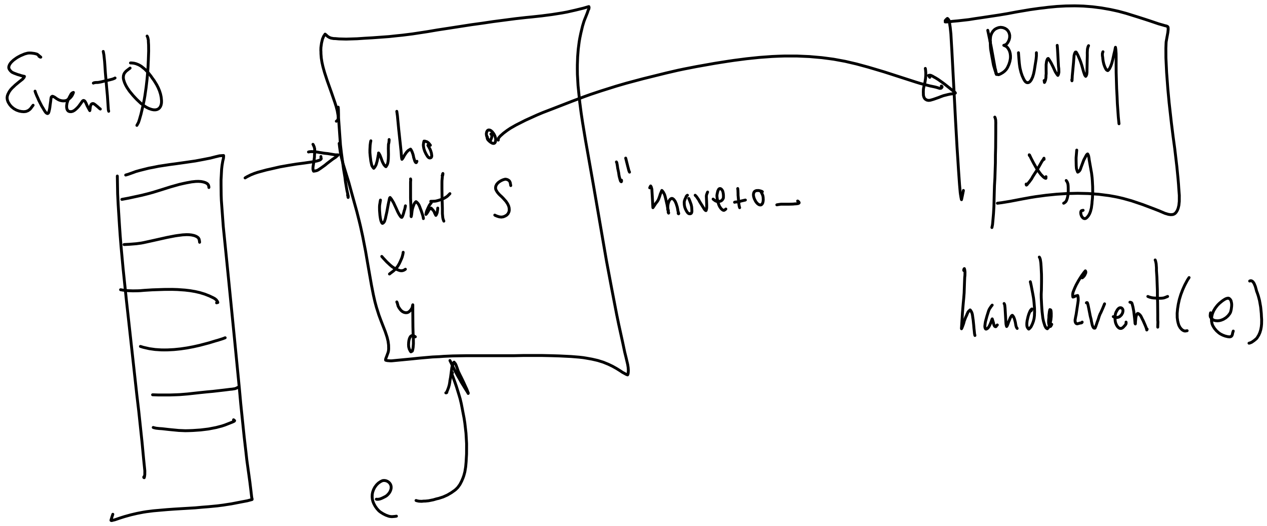
```
class Thing {
        static int n;
        int id;
}

int Thing::n = 0;

Thing::Thing () {
        id = n++;
}
```

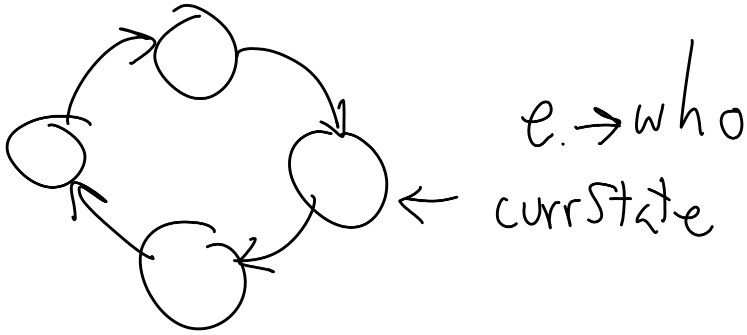
```
CellData c;
Row      r(4,c);
Field    f(4,r);
```

//-- Could we make the syntax more natural?

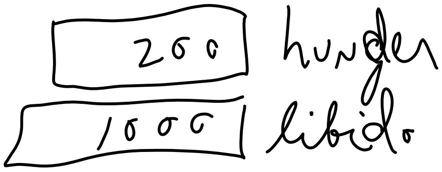
```
f[1,1].put( p );   ?
```

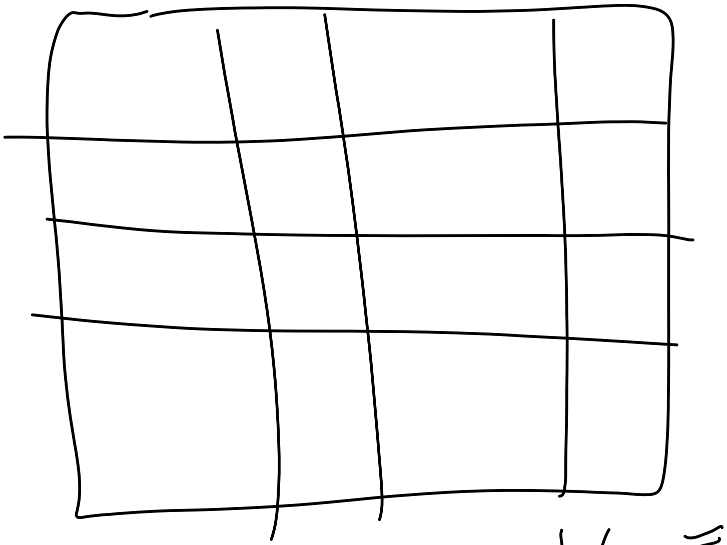
See syntaxCorner/vec2.c

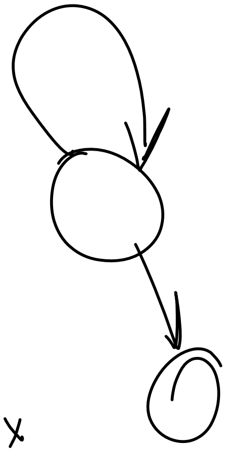




```
e->who->handleEvent(e);
```







$$V_{max} = \frac{\Delta d_{max}}{\Delta t}$$