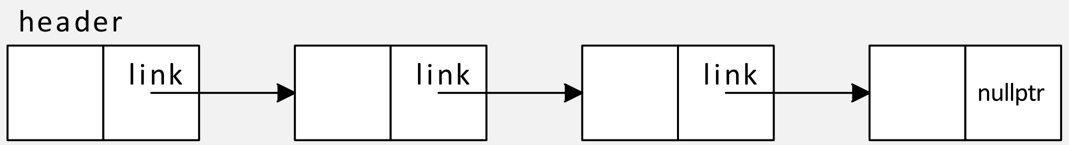
DESTRUCTORS

An inverse and complement to constructors

Delroy A. Brinkerhoff

CONSTRUCTORS AND DESTRUCTORS



```
node header;
header->link = nullptr;


node*  l = &header;

while (l->link != nullptr)
    l = l->link;

node*  temp = new node();
```

- Dynamic data structures
  - Pointers must be initialized
  - Heap memory must be deallocated
- Libraries include startup and shutdown functions
- Too easy for programmers to forget to call the functions

MEMORY LEAKS

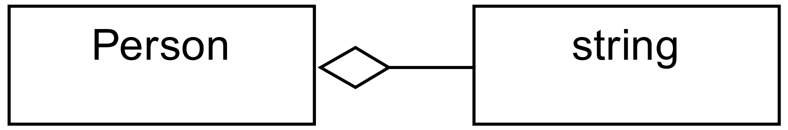
OVERWRITING AN ADDRESS

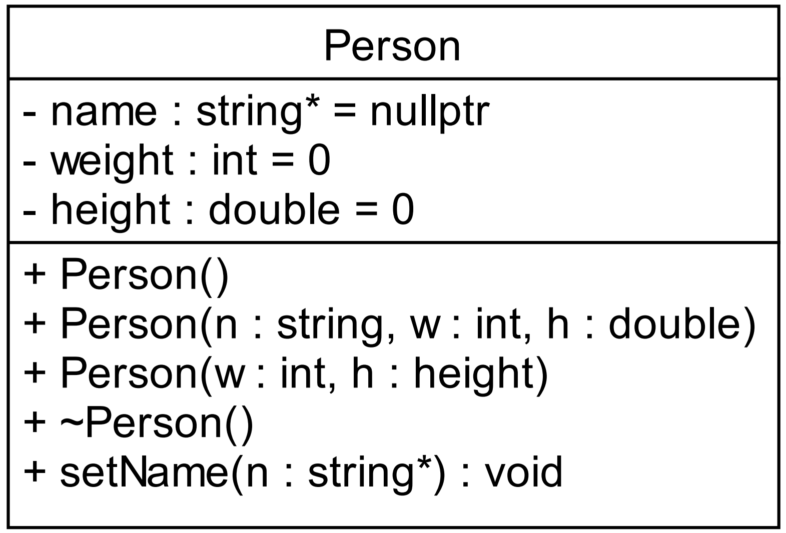
```
Person* p1 = new Person;

p1 = new Person;
```

LOST ADDRESS

```
void f()
{
    Person* p2 = new Person;
    ...
}
```

- Memory allocated for the objects is unreachable, becoming “garbage”
- The operating system reclaims lost memory at program termination
- Destructors help prevent some memory leaks, but not these




Person
- name : string* = nullptr
- weight : int = 0
- height : double = 0
+ Person()
+ Person(n : string, w : int, h : double)
+ Person(w : int, h : height)
+ ~Person()
+ setName(n : string*) : void

WHERE DESTRUCTORS DO HELP

THE FUNDAMENTALS OF OBJECT CONSTRUCTION AND DESTRUCTION

CONSTRUCTOR

```
class Whole
{
    private:
        Part* p = nullptr;
    public:
        Part() : p(nullptr) {}
};
```

DESTRUCTOR

```
Whole::~Whole
{
    if (p != nullptr)
        delete p;
};
```

IMPLICIT DESTRUCTOR CALLS

```
void g()
{
    Person p1("Wally");
}
```

```
void f()
{
    Person* p2 = new Person("Dilbert");
    delete p2;
}
```