RAII AND RULE OF THREE

RAII & Rule of Three

by Prasanna Ghali

Plan for Today

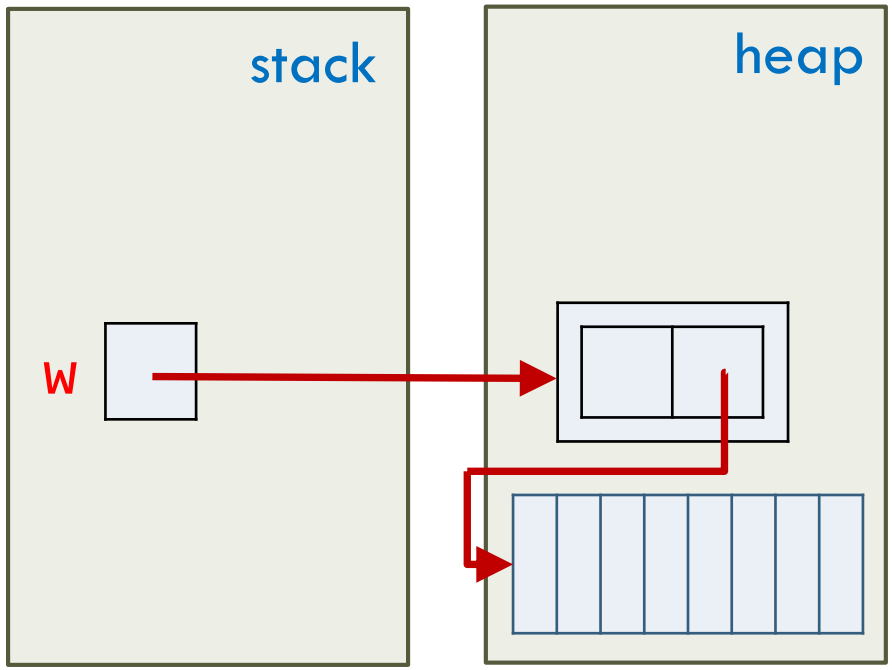
- RAII
- Rule of Three

Resource Management

- Functions open operate this way:
  - Acquire some resources such as memory, locks, sockets, threads, file handles, …
  - Perform some operations
  - Free acquired resources
- Functions can manage resources using:
  - Local objects
  - Pointers

Resource Management Thro' Pointers


```
class X { ... };

void f() {
  X *w = new X;
  ...
  delete w;
}
```

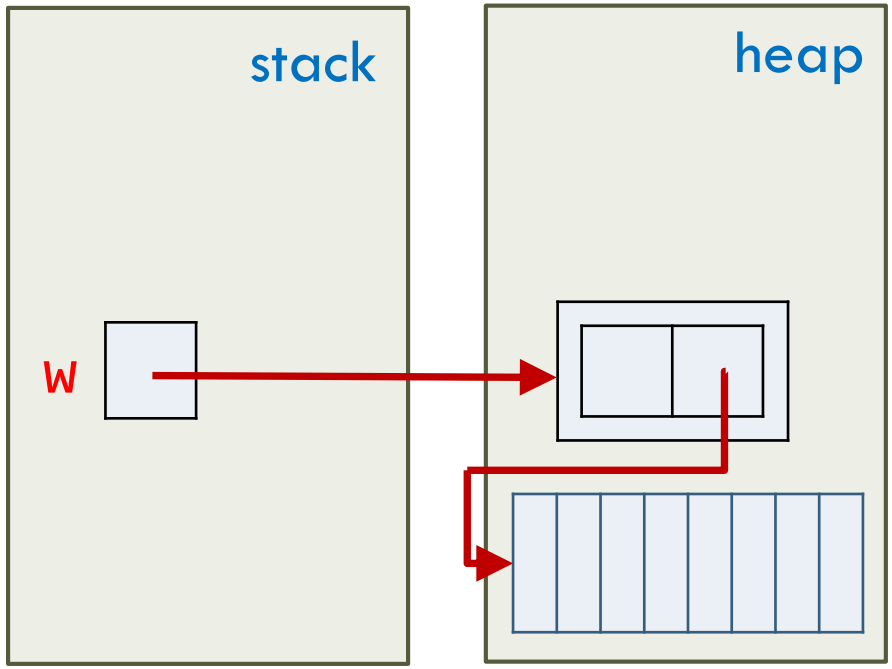


Function f is a source of trouble!!!
Can cause *leaked objects* or *premature deletion* or *double deletion*!!!

Leaked Objects

- People use `new` and then forget to call `delete`

```
void f() {
  X *w = new X;
  ...
  delete w;
}
```



Premature Deletion


```
void g() {
  X *p1 {new X{"abc"}}; // create an object explicitly
  X *p2 {p1};           // potential trouble
  delete p1;            // now p2 doesn't point to a valid object
  p1 = nullptr;         // gives a false sense of safety
  X *p3 = new X{"xyz"}; // p3 may now point to the memory
                        // pointed to by p2
  *p2 = "pqr";          // this may cause trouble
  std::cout << *p3 << '\n';  // may not print "xyz"
}
```

Double Deletion


```
// very bad code
void sloppy() {
  X *p = new X [1000];     // acquire memory
  // ... use *p ...
  delete [] p;             // release memory
  // ... wait a while ...
  delete [] p;             // but sloppy() does not own *p
}
```

Resource Management and Exceptions (1)

- Even if people can avoid leaked objects, premature deletion, double deletion, there's the matter of exceptions …

```
// naïve code
void use_file(char const *fn) {
  FILE *pf = fopen(fn, "r");

  // ... use pf ...
  // what if an exception is thrown here ...

  fclose(pf);
}
```

Resource Management and Exceptions (1)

- An attempt to make `use_file` fault tolerant

...

```
// verbose, tedious, and potentially expensive code
void use_file(char const *fn) {
  FILE *pf = fopen(fn, "r");

  try {
    // ... use pf ...
  } catch (...) {   // catch every possible exception
    fclose(pf);
    throw; // why throw again?
  }

  fclose(pf);
}
```

Resource Management: General Problem (1)

- Fault-tolerant code becomes significantly more complex when several resources must be acquired and released
- To find more elegant solution, let's look at general form of problem:

```
void acquire() {
  // acquire resource 1
  // ...
  // acquire resource n

  // ... use resources ...

  // release resource n
  // ...
  // release resource 1
}
```

Resource Management: General Problem (2)

- Important that resources are released in reverse order of their acquisition
- This resembles behavior of *local objects* created by ctors and destroyed by dtors

```
void acquire() {
  // acquire resource 1
  // ...
  // acquire resource n

  // ... use resources ...

  // release resource n
  // ...
  // release resource 1
}
```

Class FilePtr That Acts Like FILE* (1)

- This means we can handle resource acquisition and release problems using objects of classes with ctors and dtors

```
class FilePtr {
  FILE *p;
public:
  FilePtr(char const *n, char const *a)
  : p(fopen(n, a)) { // open file n
    if (p == nullptr) throw std::runtime_error("Can't open file");
  }
  FilePtr(string const& n, char const *a) // delegating ctor
  : FilePtr(n.c_str(), a) { }

  explicit FilePtr(FILE *rhs) : p(rhs) { // assume ownership of rhs
    if (p == nullptr) throw runtime_error("nullptr");
  }
  // suitable copy operations ...
  ~FilePtr() { fclose(p); } // dtor
  operator FILE* () { return p; }
};
```

Class FilePtr That Acts Like FILE* (2)

- use_file shrinks from left to this minimum on right

```
void use_file(char const *fn) {
  FILE *pf = fopen(fn, "r");

  try {
    // ... use pf ...
  } catch (...) {
    fclose(pf);
    throw;
  }

  fclose(pf);
}
```

```
void use_file(char const *fn) {
  FilePtr f(fn, "r");
  // ... use f ...
}
```

Class FilePtr That Acts Like FILE* (3)


```
void use_file(char const *fn) {
  FilePtr f(n, "r");

  // ... use f ...

  // if exception is thrown, f.~FilePtr() is
  // automatically invoked by runtime environment
}
```

RAII

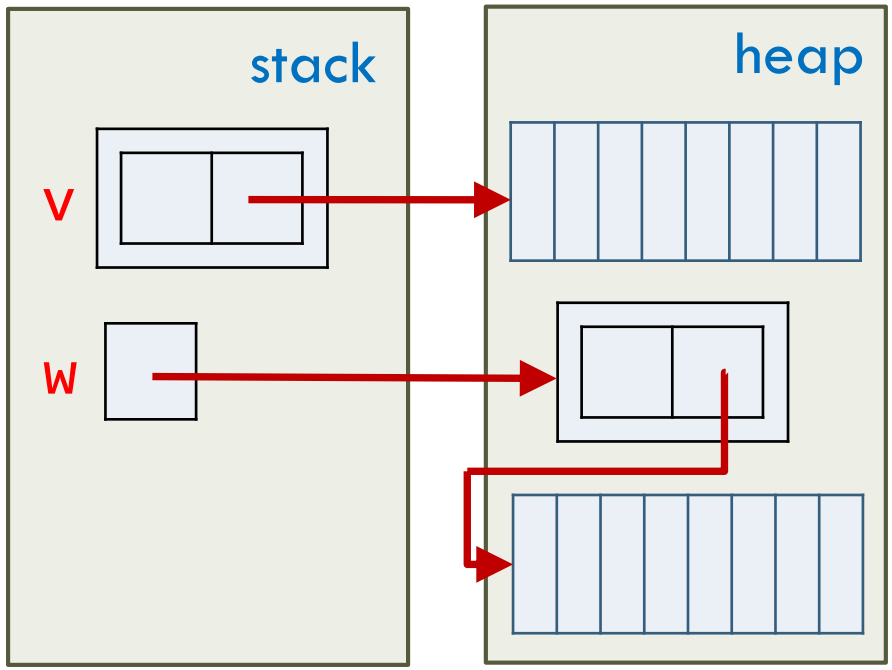
- *Resource Acquisition Is Initialization*
- Technique for managing resources using local objects that relies on properties of ctors and dtors and their interaction with exception handling
- Helps with:
  - Avoiding resource leaks
  - Makes error handling using exceptions simple and safe

How Does RAII Help? (1)

- Imagine program that allocates 45,678 memory blocks in 987 program locations
  - How can you be sure all memory blocks are freed?
  - What is certainty that memory and other resources (mutexes, file handles, network sockets, …) are released when exceptions are thrown?
- *Fundamental idea of RAII is that single local object is responsible for resource and releases it at end of its lifetime*
- RAII is an application of Single Responsibility Principle

How Does RAII Help? (2)

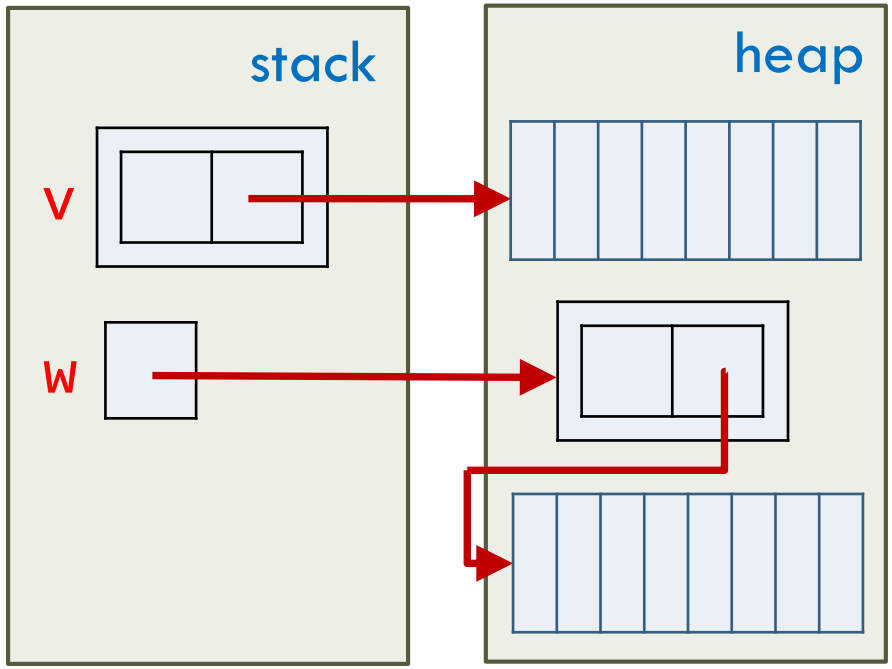

```
class X { ... };

void f() {
  X v;
  X *w = new X;
  ...
  delete w;
}
```



w is a source of trouble!!!
Can cause *leaked objects* or *premature deletion* or *double deletion*!!!

How Does RAII Help? (3)


```
void f() {
  X v;
  X *w = new X;
  ...
  delete w;
}
```



RAII Guideline 1

- *Resource handling class should not manage more than one resource*
  - Motivation: When ctor throws exception, it doesn't need to worry about other resources being released
- Suppose class contains two resource handling class members
  - If 1st member is constructed and 2nd member throws an exception during construction, then dtor for 1st member is invoked but not for 2nd member
  - If both members are constructed and then ctor throws an exception, there will be no leaks since dtors for members will be called

RAII Guideline 2

- Class dtor *must not* throw an exception

```
class Widget {
public:
  // other functions

  // dtor might throw
  // an exception
  ~Widget();
};
```

```
void foo() {
  std::vector<Widget> v(10);
  // use v
} // v is automatically destroyed here

int main() {
  try {
    foo();
  } catch (...) {
    // catch exception thrown by foo
  }
}
```

RAII Vector Class

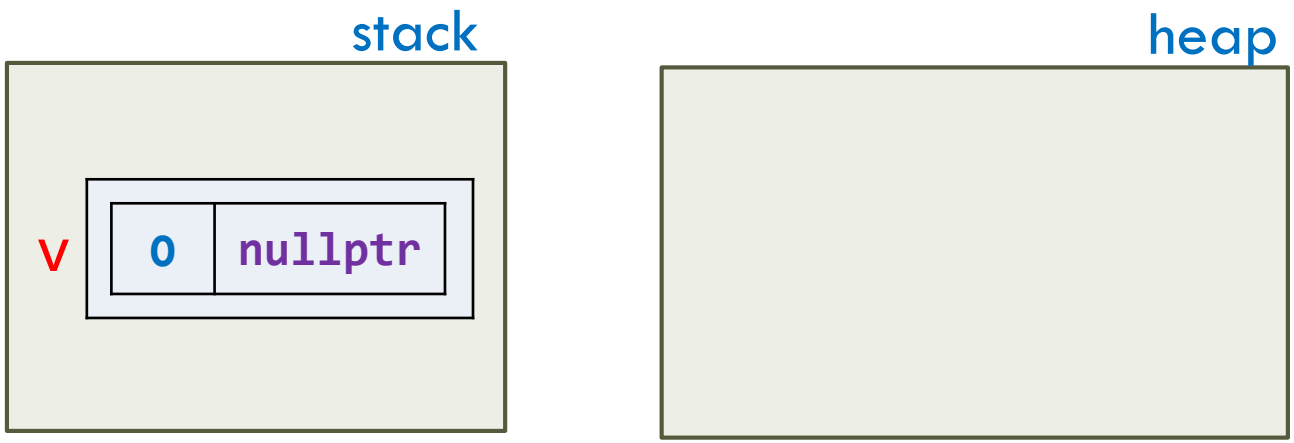

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  void push_back(int val) {
    int *tmp_ptr {new int [len+1]};
    std::copy(ptr, ptr+len, tmp_ptr);
    delete [] ptr;
    ptr = tmp_ptr;
    ptr[len++] = val;
  }
};
```

RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```

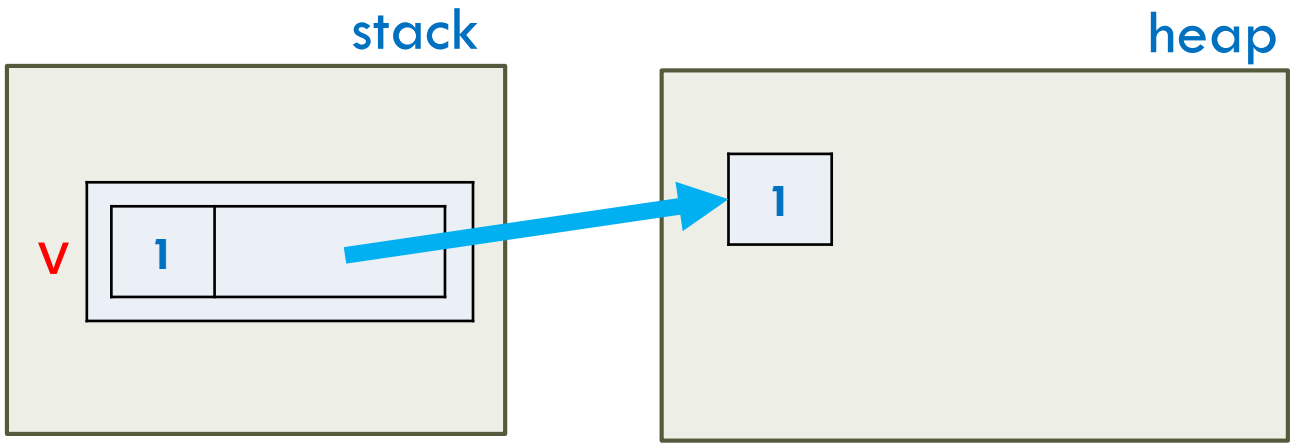
RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```



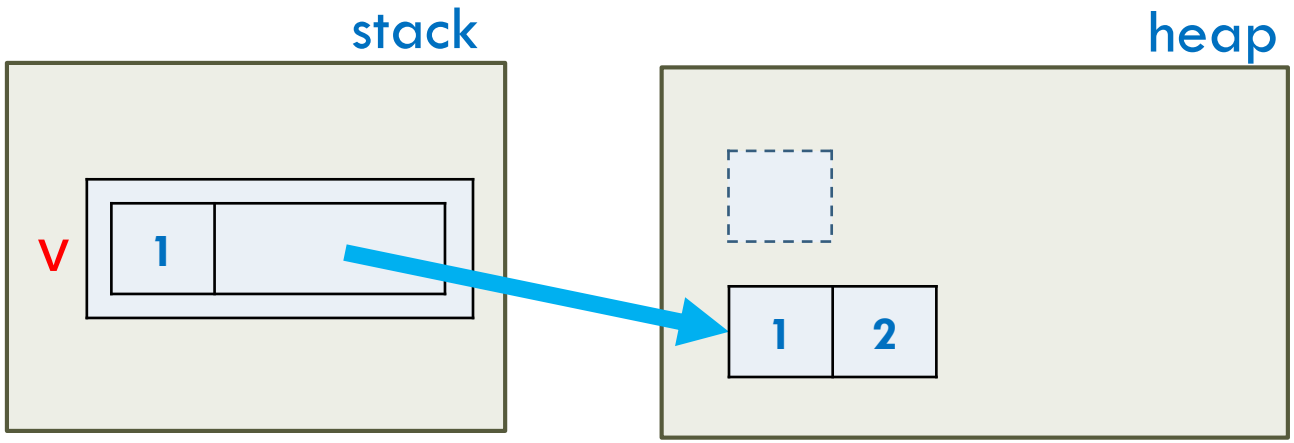
RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```



RAII Vector Class

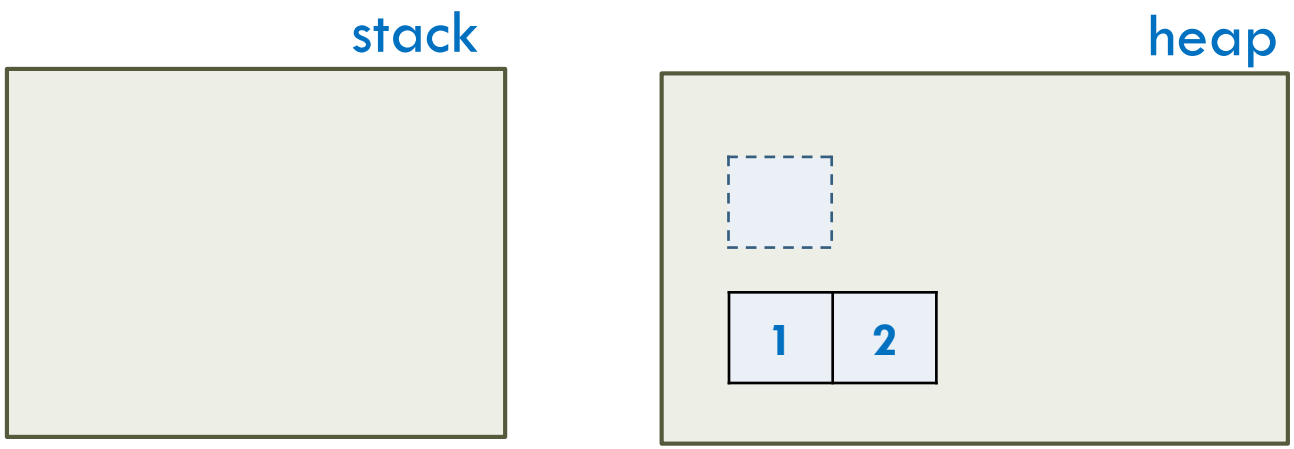

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```



RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```

synthesized dtor
causes memory leak!!!



RAII Vector Class

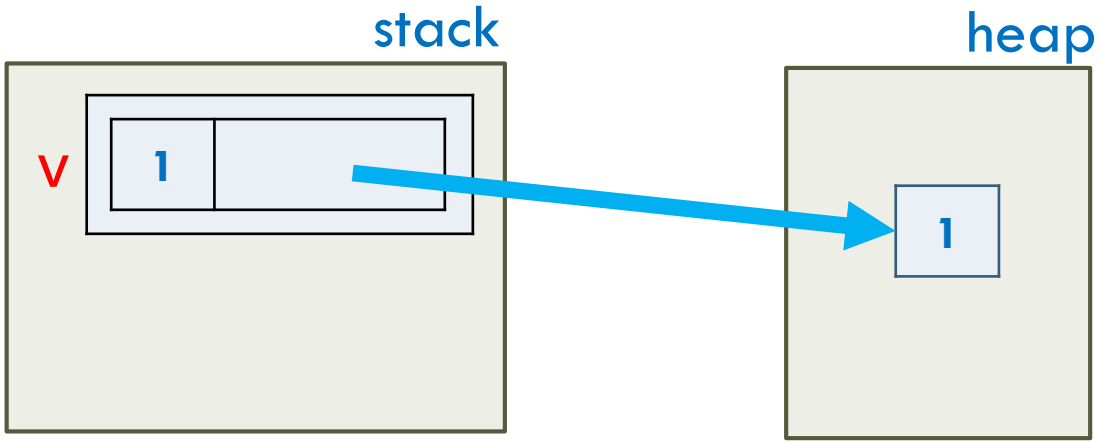

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  void push_back(int val) {
    int *tmp_ptr {new int [len+1]};
    std::copy(ptr, ptr+len, tmp_ptr);
    delete [] ptr;
    ptr = tmp_ptr;
    ptr[len++] = val;
  }
  ~Vec() { delete [] ptr; }
};
```

RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```

RAII Vector Class

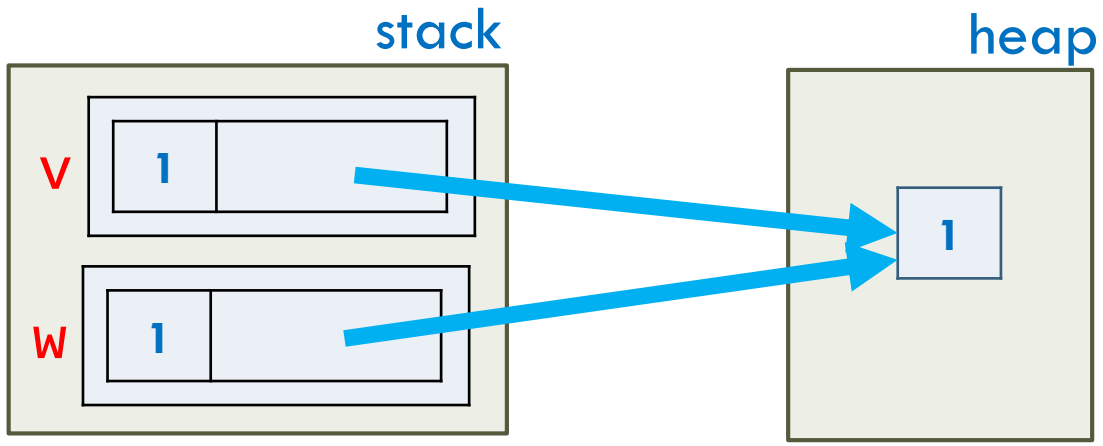

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class

synthesized copy constructor creates shallow clone!!!

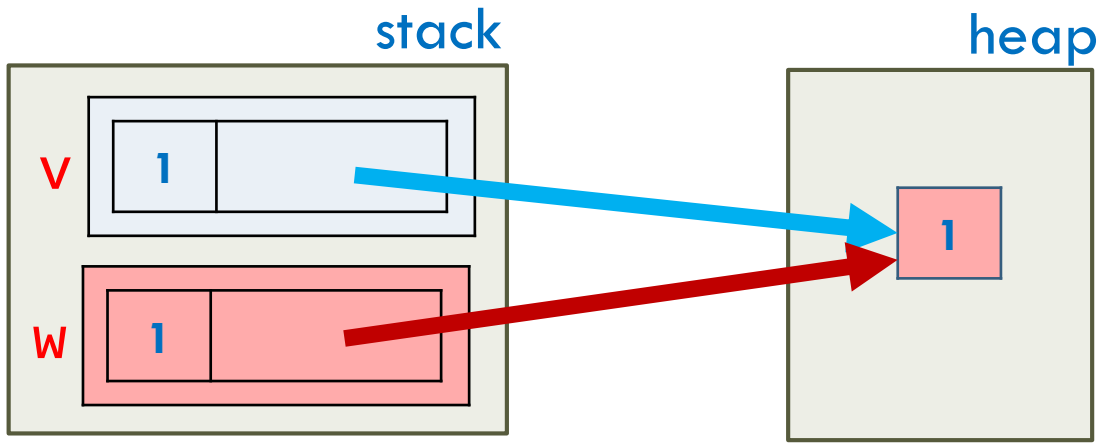
```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class

premature deletion!!!

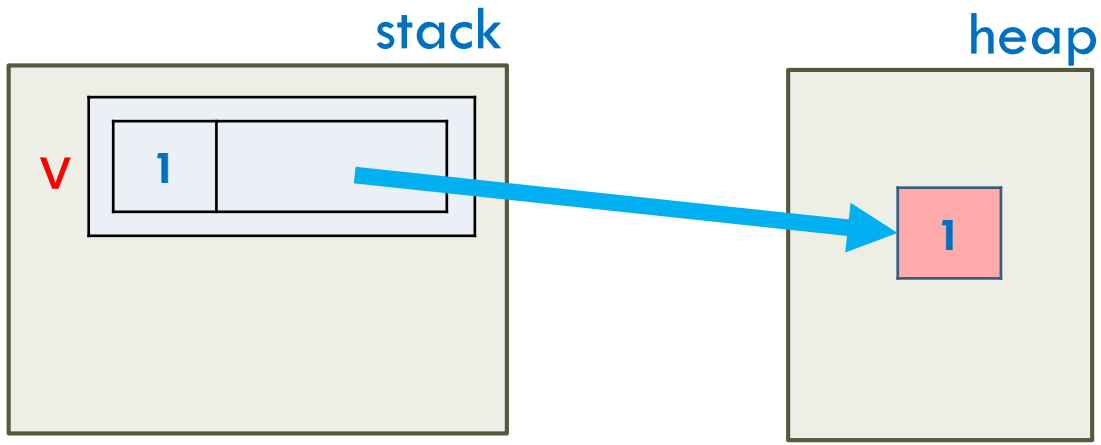
```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class

undefined behavior!!!

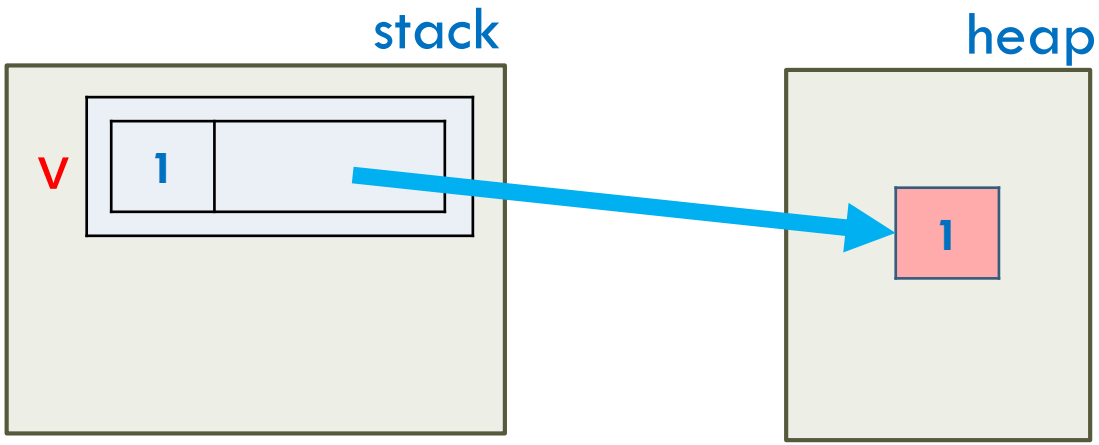
```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```

double deletion!!!



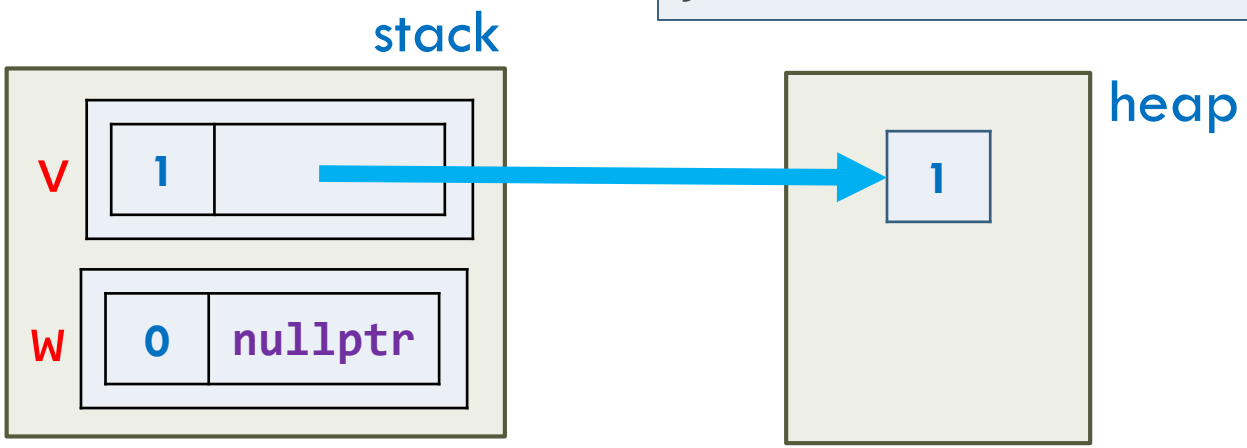
RAII Vector Class

copy constructor

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  ~Vec() { delete [] ptr; }
  Vec(Vec const& rhs)
  : len{rhs.len}, ptr{new int [len]} {
    std::copy(rhs.ptr, rhs.ptr+len, ptr);
  }
};
```

RAII Vector Class

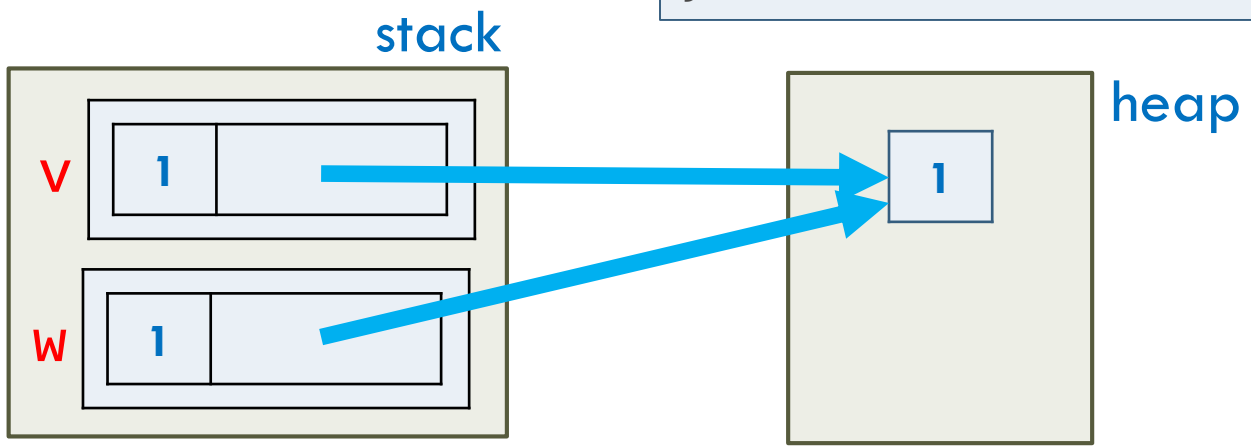

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w;
    w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class

synthesized copy assignment operator performs shallow copy!!!

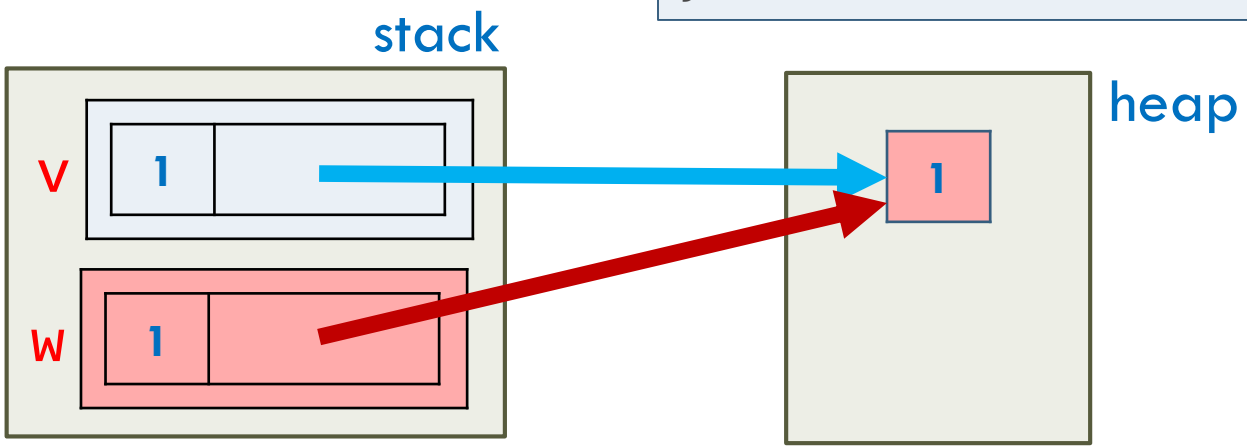
```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w;
    w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w;
    w = v;
  }
  std::cout << v[0] << '\n';
}
```

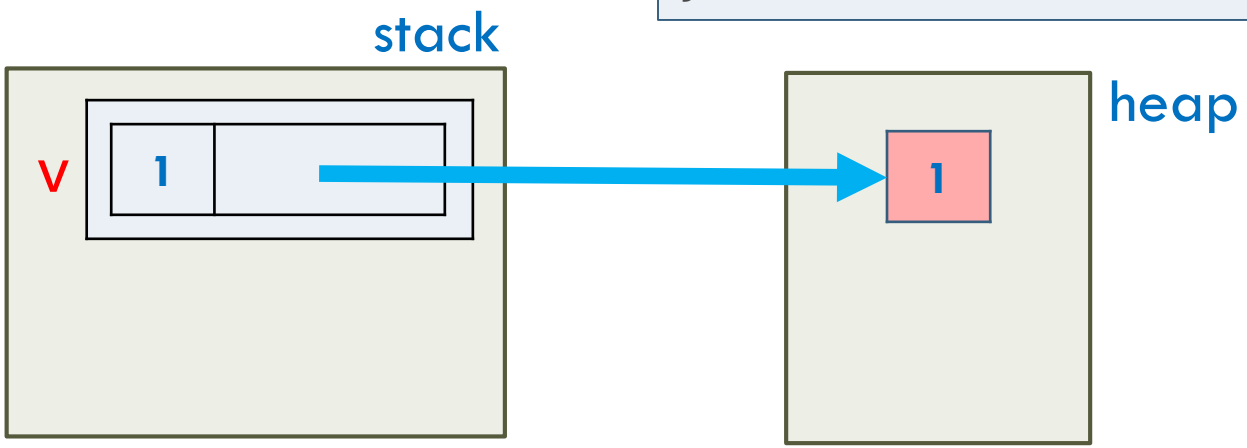
premature deletion!!!



RAII Vector Class


```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w;
    w = v;
  }
  std::cout << v[0] << '\n';
}
```

undefined behavior!!!



RAII Vector Class

copy assignment operator

copy-swap idiom

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  ~Vec() { delete [] ptr; }
  Vec(Vec const& rhs)
  : len{rhs.len}, ptr{new int [len]} {
    std::copy(rhs.ptr, rhs.ptr+len, ptr);
  }
  Vec& operator=(Vec const& rhs) {
    Vec copy{rhs};
    copy.swap(*this);
    return *this;
  }
};
```

RAII Classes: Rule of Three

- If your class manages a resource, you'll need to write three special member functions:
  - Destructor to release the resource
  - Copy constructor to clone the resource
  - Copy assignment operator to release current resource and acquire cloned resource