MODERN C++ DESIGN PATTERNS

RAII & Rule of Three

by Prasanna Ghali

Plan for Today

- RAII
- Rule of Three
- Adapting constructors to exceptions
- Delegating constructors

What is a Resource?

- Something that is acquired and must be given back [released] or reclaimed by some resource manager …
- Examples of programming resources: memory, locks, sockets, thread handles, file handles, windows, …
- We call object, such as `std::vector`, the *owner* or *handle* of resource for which it is responsible

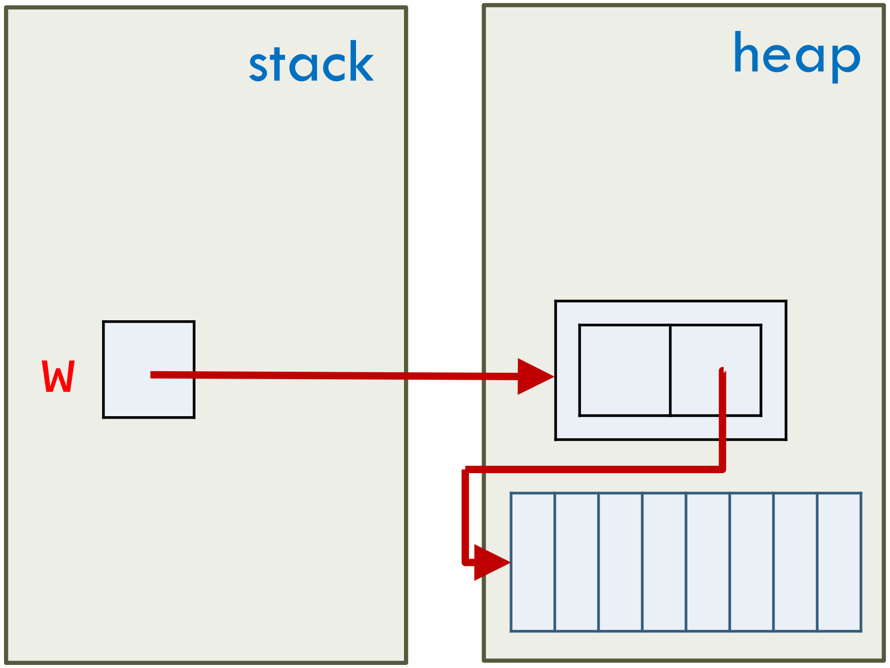
Resource Management

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

Manual Resource Management


```
class X { ... };

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

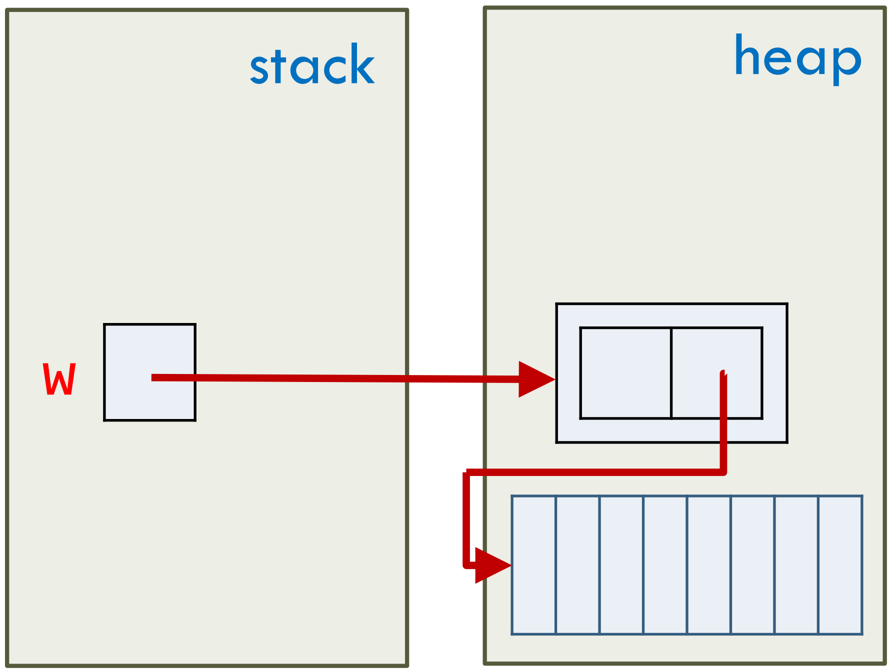


Function f is a source of trouble!!!
Can cause *leaked objects* or *premature deletion* or *double deletion*!!!

Manual Resource Management: Leaked Objects

- People use new and then forget to call delete

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



Manual Resource Management: Premature Deletion


```
// very bad code
void bad() {
  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"
}
```

Manual Resource Management: Double Deletion


```
// terrible 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
}
```

Manual Resource Management: Multiple Exit Points

- Even if people can avoid leaked objects, premature deletion, double deletion, there's the matter of multiple exit points ...

In larger functions with multiple exit points, managing resources becomes even more error prone

```
// function has two exit points and the
// file has to be closed in each one ...
std::string use_file(char const *name) {
  FILE *pf = fopen(name, "r");
  int ch = fgetc(pf);
  if (ch != '$') {
    // forgot to close file!!! ...
    return "invalid file";
  }
  fclose(pf);
  return std::string(1, ch);
}
```

Manual Resource Management: Exceptions (1/2)

- Then 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);
}
```

Manual Resource Management: Exceptions (2/2)

- Attempt to make `use_file` fault tolerant …

```
// use try-catch technique to deal with "exception problem"
void use_file(char const *fn) {
  FILE *pf = fopen(fn, "r");

  // verbose, tedious, and potentially expensive code
  try {
    // ... use pf ...
  } catch (...) {   // catch every possible exception
    fclose(pf);
    throw; // why throw again?
  }

  fclose(pf);
}
```

Resource Management: General Problem

- To find more elegant solution, let's look at general form of problem:

Local variables [with automatic storage duration] allow easy management of multiple resources within single function:
- They're destroyed in reverse order of their construction
- An object is destroyed only if it is fully constructed

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

  // ... use resources ...

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

RAII Idiom: Solution to Resource Management Problem

- Resource acquisition and release problems can be handled using objects of classes with ctors and dtors

Use class-based local object to own a resource during its initialization by having object's ctor acquire the resource

In C++, it is guaranteed that *dtor of object with automatic storage duration is invoked* when object's scope ends naturally or because of `return` statement or because exception occurs

Use this core C++ feature to handle cleanup of resource in object's dtor

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

  // ... use resources ...

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

RAII Idiom (1/2)

- *Resource Acquisition Is Initialization*
  - *So called because it's so common to acquire a resource and initialize a resource-managing object in same statement*
- *Programming principle that ties resource to lifetime of object that acquired the resource*
- *Resources are acquired when object is constructed and released when object is destructed*
- *Fundamental idea of RAII is that stack-based object is responsible for its resource and releases it at end of its lifetime*

RAII Idiom (2/2)

- RAII is an application of Single Responsibility Principle

RAII Idiom Example: Class `FilePtr` (1/5)


```
// Class FilePtr encapsulates FILE* so that FilePtr objects
// are owners of file handles
class FilePtr {
  FILE *p;
public:
  FilePtr(char const *n, char const *a) // open file named n
  : p(fopen(n, a)) {
    if (p == nullptr) throw std::runtime_error("Can’t open file");
  }
  FilePtr(string const& n, char const *a) // DRY: delegate to ctor
  : FilePtr(n.c_str(),  a) { }

  explicit FilePtr(FILE *rhs) : p(rhs) { // assume ownership of rhs
    if (p == nullptr) throw runtime_error("nullptr");
  }
  ~FilePtr() { fclose(p); } // dtor

  // suitable copy operations ...

  operator FILE* () { return p; }
};
```

RAII Idiom Example: Class `FilePtr` (2/5)

- `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 ...
} // calls f.~FilePtr() ...
```

RAII Idiom Example: Class `FilePtr` (3/5)


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

  // ... use f ...

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

RAII Idiom Example: Class `FilePtr` (4/5)

- Resource acquisition is directly tied to object's initialization
- Since p2's ctor will throw an exception, its initialization will fail and therefore, p2's dtor will not be invoked

```
void use_file(char const *fn) {
  FilePtr p1(fn, "r");

  try {
    // p2's ctor will throw an exception ...
    FilePtr p2("non-existent.txt", "r");
  } catch (std::exception const& e) {
    std::cout << e.what() << '\n';
  }
} // p1.~FilePtr() will be invoked
```

RAII Idiom Example: Class `FilePtr` (5/5)

- In modern C++, RAII can be implemented without the need to create a new class `FilePtr` to wrap `FILE*`
- We can do the same using standard library's smart pointers [more on smart pointer later]

RAII: Benefits

- Simplifies resource management
- Reduces overall code size
- Helps ensure program correctness
- Avoids allocations in general code by burying them inside implementations of well designed abstractions
- Avoiding resource leaks by using local [stack-based] objects to manage resources
- Makes error handling using exceptions simple and safe

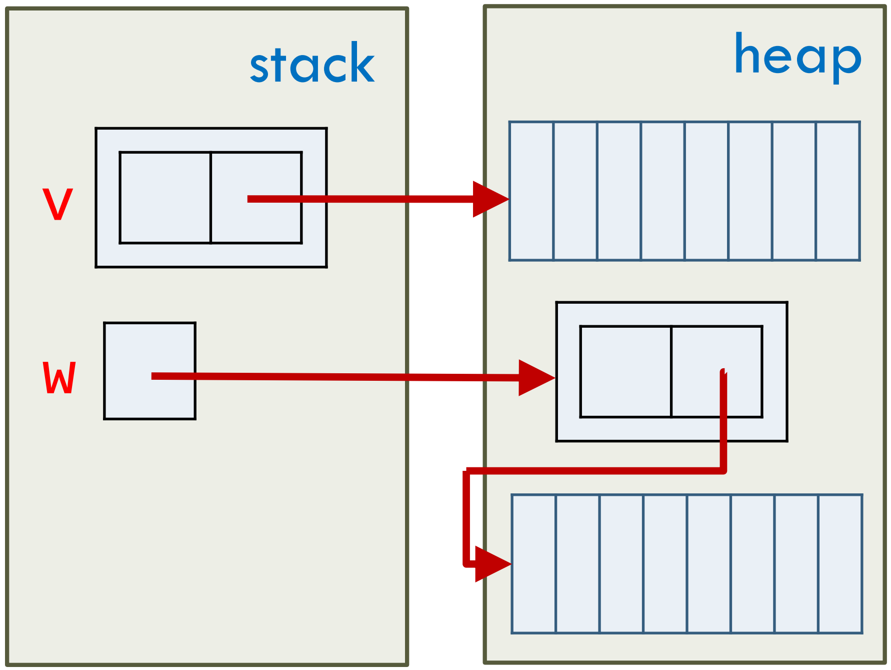
RAII Simplifies Resource Management (1/5)

- When automatic storage duration object goes out of scope, its dtor runs
- When exception occurs, all automatic duration objects that have been fully constructed since last `try`-block began are destroyed in reverse order they were created before any `catch` handler is invoked
- If you nest `try`-blocks, and none of `catch` handlers of inner `try`-block handles that type of exception, then exception propagates to outer `try`-block
  - All automatic duration objects fully constructed within that outer `try`-block are then destroyed in reverse creation order before any `catch` handler is invoked, and so on, until something catches the exception
  - Or your program crashes

RAII Simplifies Resource Management (2/5)


```
class X { ... };

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



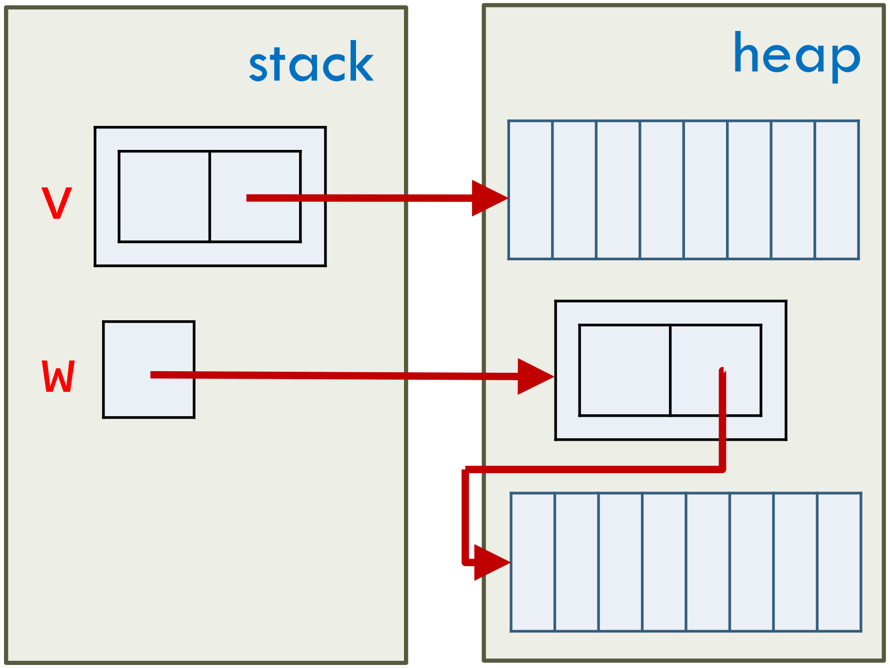
w is a source of trouble but not v!!!

w can cause *leaked object* or *premature deletion* or *double deletion*!!!

But not v!!!

RAII Simplifies Resource Management (3/5)

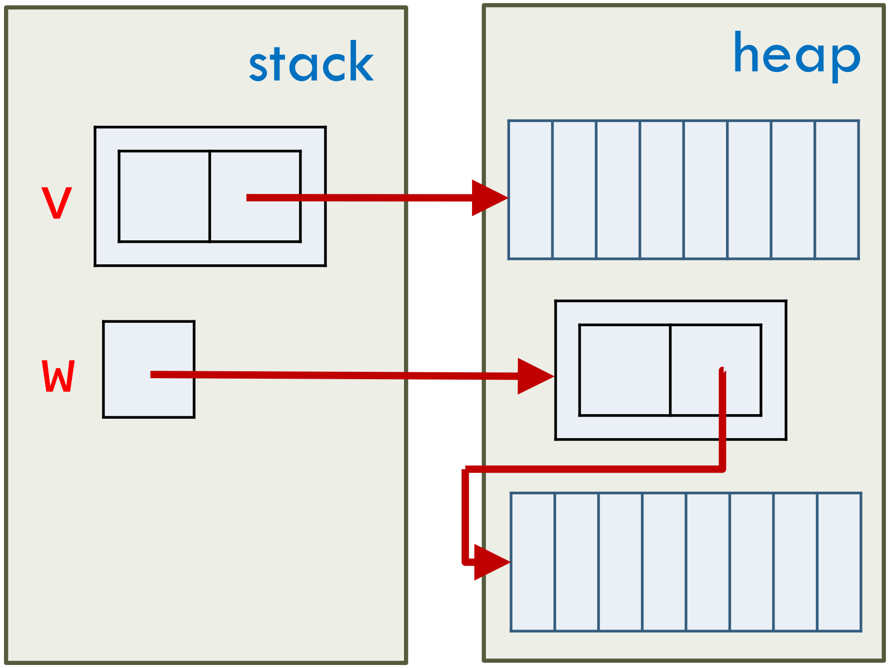

```
void f() {
  X v;
  X *w = new X;
  ...
  delete w;
}
```



RAII Simplifies Resource Management (4/5)


```
class X { ... };

void f() {
  X v;
  X *w = new X;
  // exception thrown here
  delete w;
}
```



w is a source of trouble but not v!!!

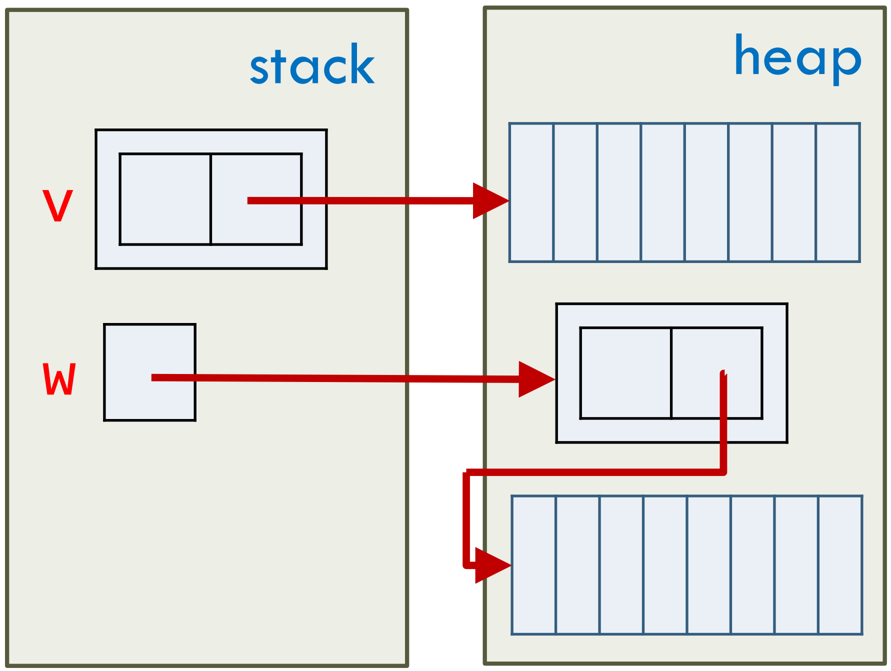
w can cause *leaked object* or *premature deletion* or *double deletion*!!!

But not v!!!

RAII Simplifies Resource Management (5/5)


```
class X { ... };

void f() {
  X v;
  X *w = new X;
  // exception thrown here
  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 1^{st} member is constructed and 2^{nd} member throws an exception during construction, then dtor for 1^{st} member is invoked but not for 2^{nd} 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 (1/5)

- Although C++ doesn't prohibit dtors from emitting exceptions, dtors *must not* throw exceptions
- Why?

RAII Guideline 2 (2/5)

- Two situations in which dtor is called:
  - When object is destroyed under normal conditions [when it goes out of scope or is explicitly deleted]

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

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

```
int main() {
  Widget w;
  // use w
} // w is automatically destroyed here
```

RAII Guideline 2 (3/5)

- Two situations in which dtor is called:
  - When object is destroyed under normal conditions [when it goes out of scope or is explicitly deleted]
  - When object is destroyed by exception-handling mechanism during stack-unwinding part of exception propagation
- No way to distinguish between these conditions inside dtor

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

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

```
void foo() {
  std::vector<Widget> v(10'000);
  // use v
} // v is automatically destroyed here

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

RAII Guideline 2 (4/5)

- As a result, we must write dtors under conservative assumption that exception is active
- Two good reasons for dtor to not throw exception:
  - If exception is thrown while another is active, C++ calls `std::terminate` [which does what its name suggests]
  - If dtor throws exception then dtor won't run to completion and not release resources it was programmed to

RAII Guideline 2 (5/5)

- Although C++ doesn't prohibit dtors from emitting exceptions, dtors *must not* throw exceptions

RAII Guideline 3

- Since compilers may implicitly generate a class's default ctor, copy ctor, copy assignment, and dtor, follow Rule of Three

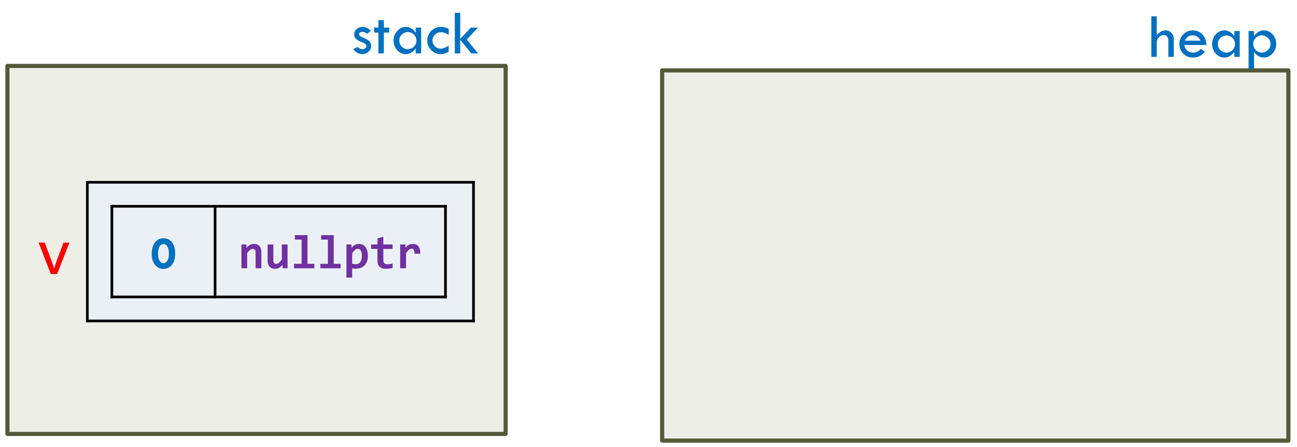
RAII Vec Class: Destructor (1/6)


```
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;
  }
};
```

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

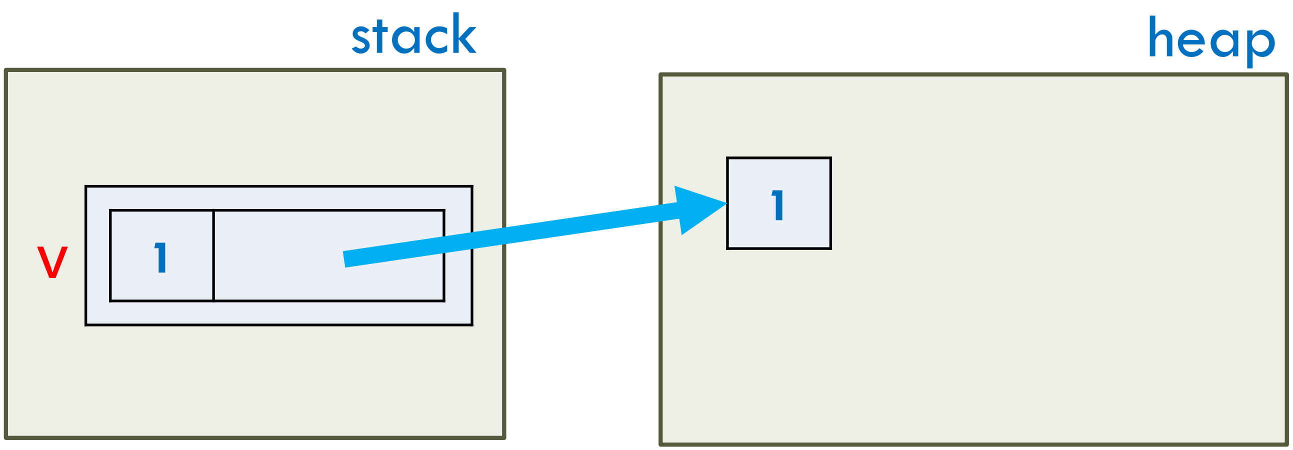
RAII Vec Class: Destructor (2/6)


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



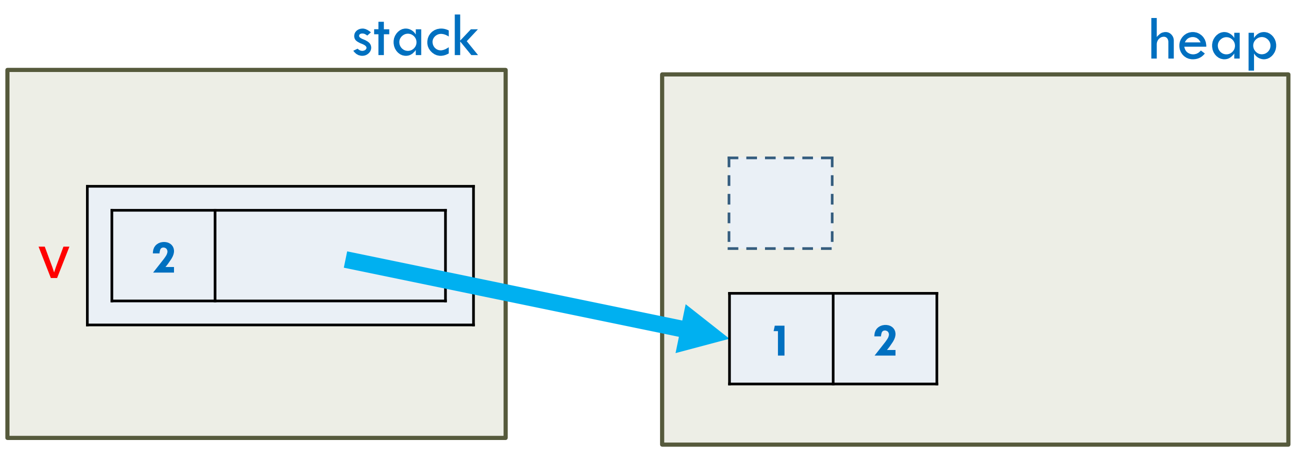
RAII Vec Class: Destructor (3/6)


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



RAII Vec Class: Destructor (4/6)

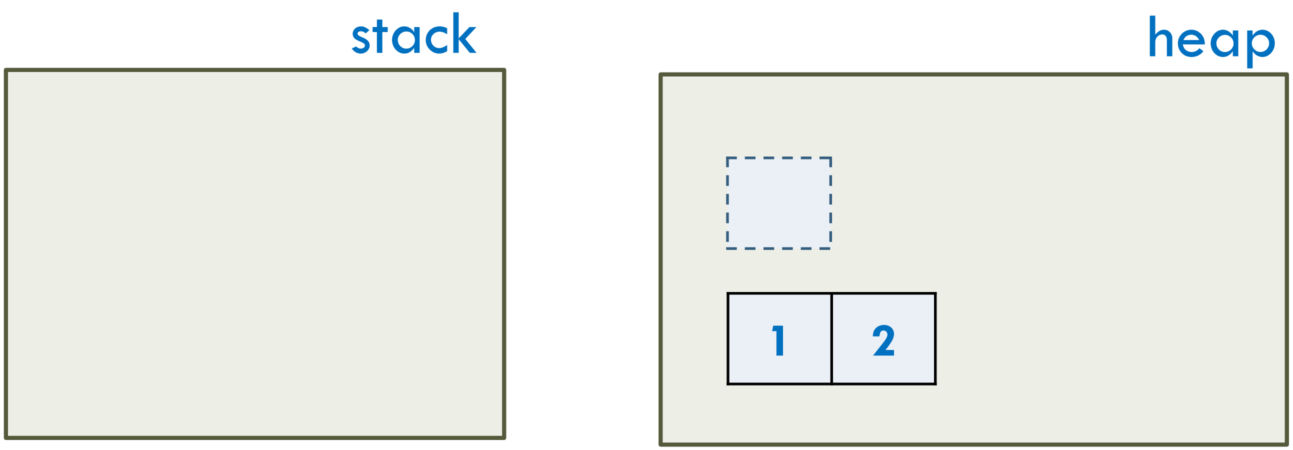

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  v.push_back(2);
}
```



RAII Vec Class: Destructor (5/6)


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

synthesized dtor
causes memory leak!!!



RAII Vec Class: Destructor (6/6)

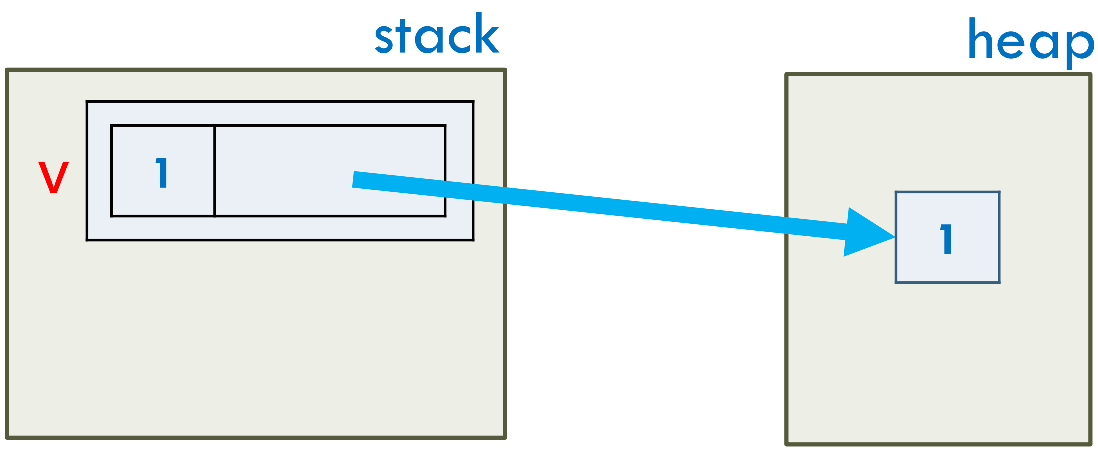

```
class Vec {
  size_t len{};
  int    *ptr{nullptr};
public:
  Vec() = default;
  void push_back(int val) {
    int *tmp_ptr {new int [len+1]};
    std::copy(ptr, ptr+len, tmp_ptr);
    delete [] ptr;
    ptr = tmp_ptr;
    ptr[len++] = val;
  }
  ~Vec() { delete [] ptr; }
};
```

RAII Vec Class: Copy Ctor (1/7)


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

RAII Vec Class: Copy Ctor (2/7)

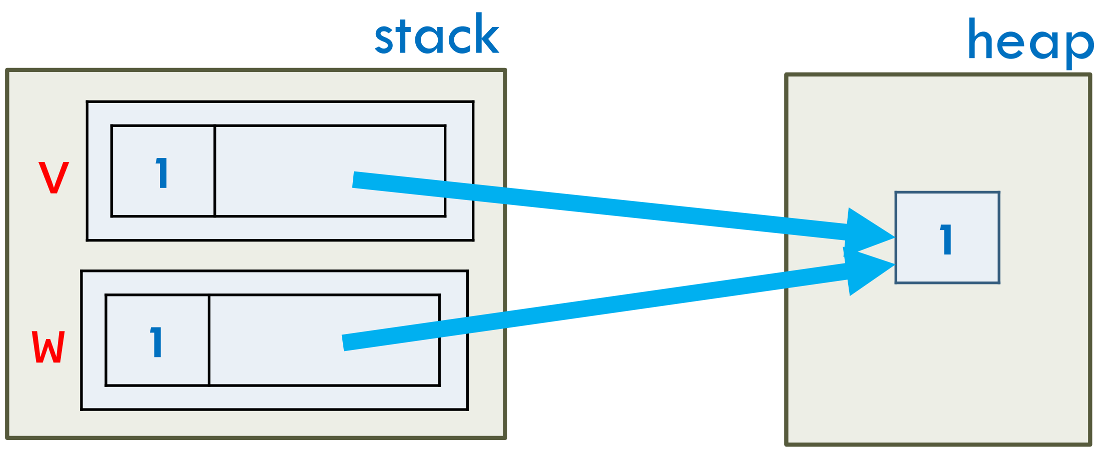

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vec Class: Copy Ctor (3/7)


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

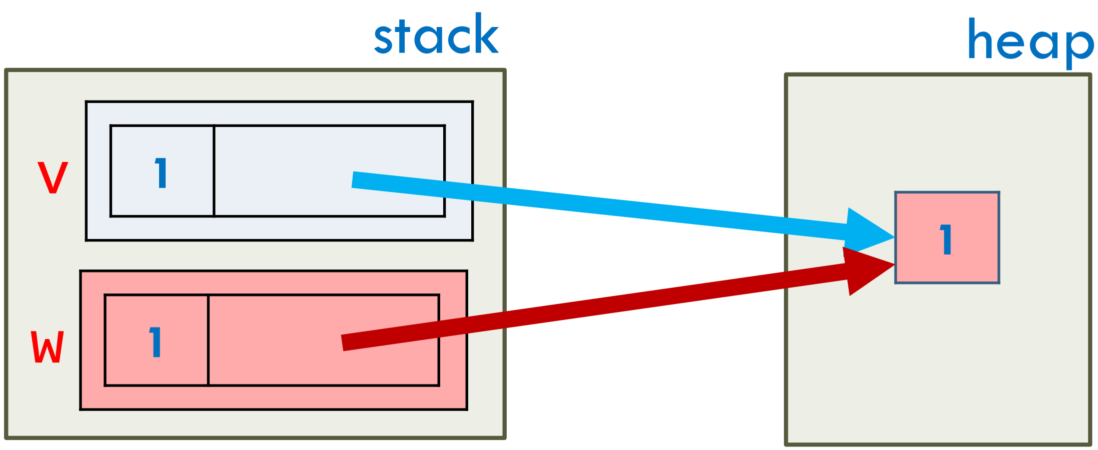
synthesized copy constructor creates shallow clone!!!



RAII Vec Class: Copy Ctor (4/7)


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

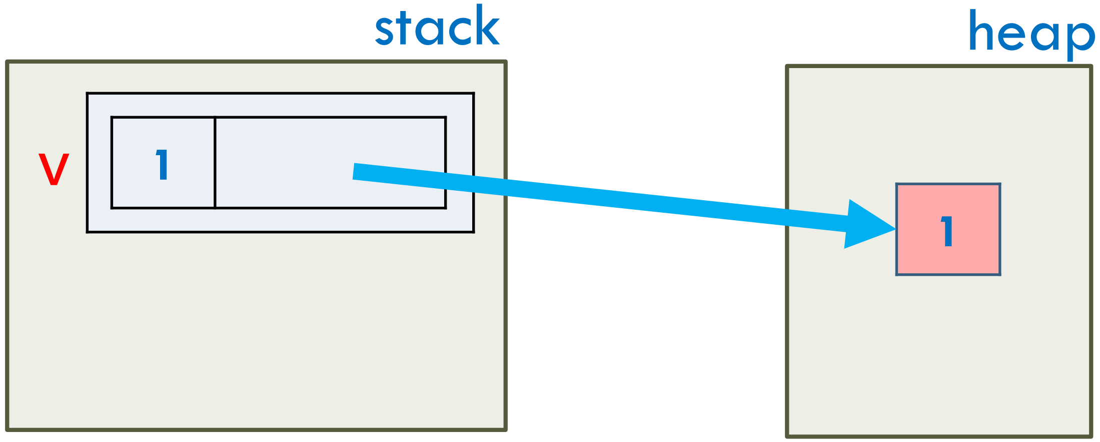
premature deletion!!!



RAII Vec Class: Copy Ctor (5/7)


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

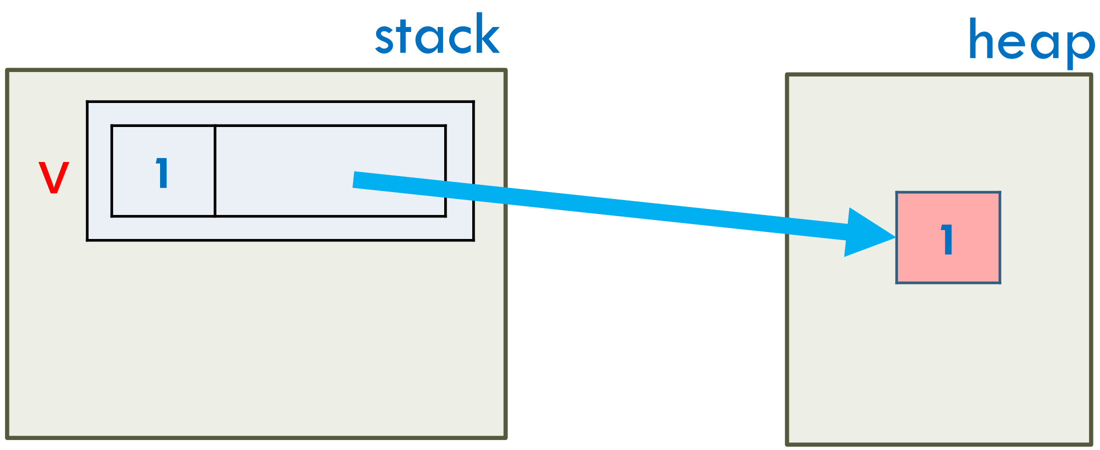
undefined behavior!!!



RAII Vec Class: Copy Ctor (6/7)


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

double deletion!!!



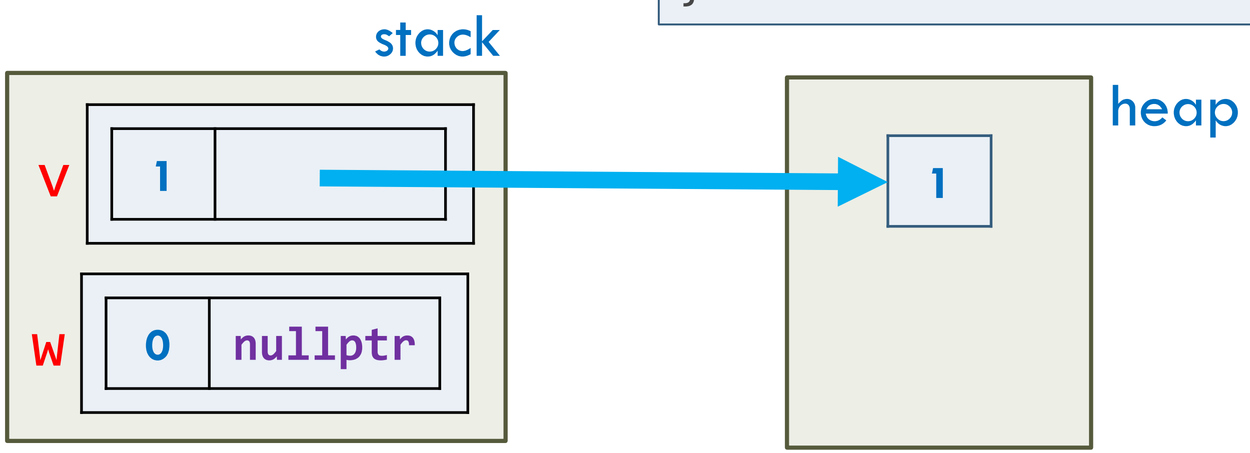
RAII Vec Class: Copy Ctor (7/7)

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 Vec Class: Copy Assignment (1/5)

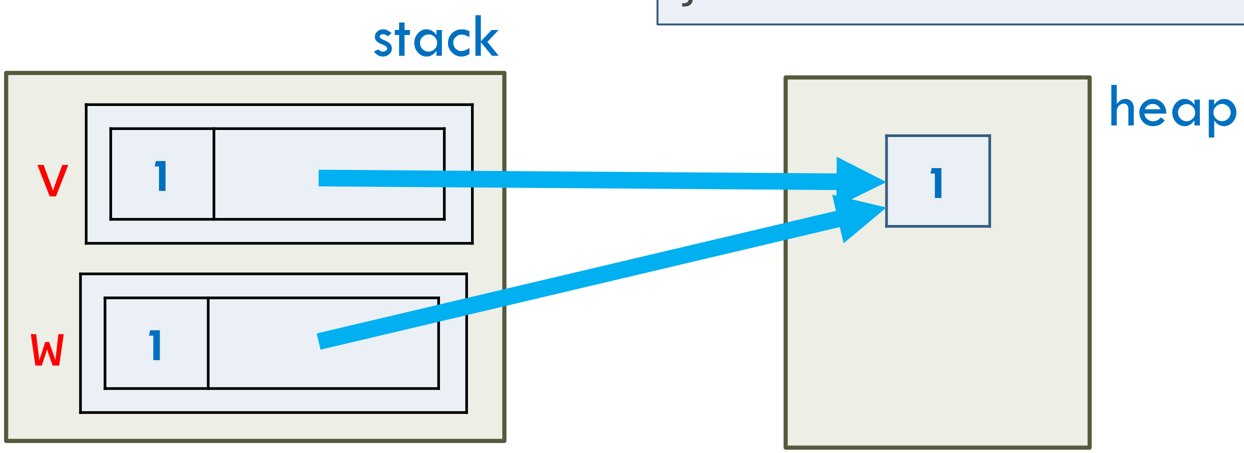

```
// is there a problem?
{
  Vec v;
  v.push_back(1);
  {
    Vec w;
    w = v;
  }
  std::cout << v[0] << '\n';
}
```



RAII Vec Class: Copy Assignment (2/5)


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

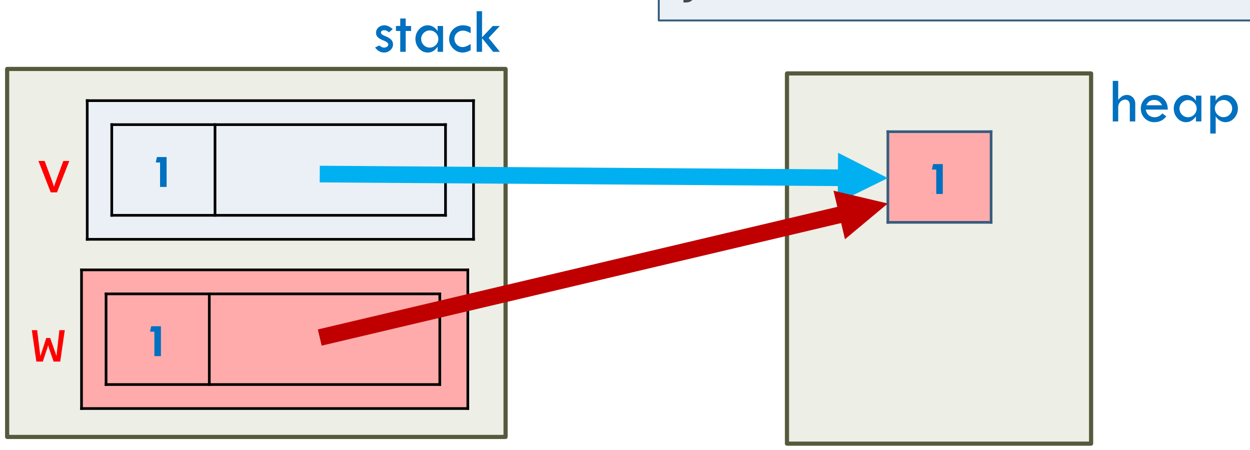
synthesized copy assignment operator performs shallow copy!!!



RAII Vec Class: Copy Assignment (3/5)


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

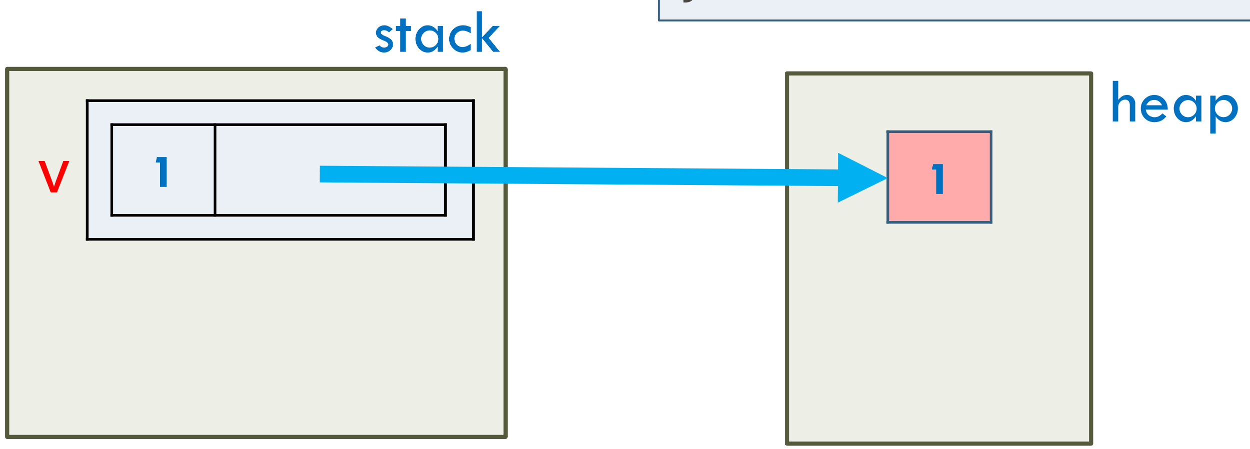
premature deletion!!!



RAII Vec Class: Copy Assignment (4/5)


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

undefined behavior!!!



RAII Vec Class: Copy Assignment (5/5)

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
- Caveat: You'll need to define `swap` function to implement copy assignment operator using *copy-swap idiom*

Compiler Can't Enforce ROT!!!

- Rule of three is good practice but C++ compiler can't enforce it
- Instead, pre-C++11 compilers mandated implicit versions be created if class doesn't explicitly declare them explicitly
  - *If class has no user-declared dtor, dtor is declared implicitly*
  - *If class doesn't explicitly declare copy ctor, one is declared implicitly*
  - *If class definition does not explicitly declare copy assignment operator, one is declared implicitly*

Broken Code!!!


```
// syntactically valid but fundamentally broken code
struct S {
  S(char const *str) : p{new char [strlen(str)+1]} {}
  ~S() { delete [] p; }
private:
  char *p;
};

int main() {
  S x{"whatever"};
  S y{x}; // shallow copy thro’ implicit copy ctor
}
```

Copy Functions Deprecated

- C++11 deprecated implicit definition of copy functions
  - *Implicit definition of copy ctor as defaulted is deprecated if class has user-declared copy assignment operator or user-declared dtor*
  - *Same for implicit definition of copy assignment operator*
  - *In future revision of standard, these implicit definitions could become deleted*
- Behavior continued with C++14/17 …
- All of this means compilers keep generating defaulted copy functions and dtor if no user-defined declarations are found
  - At least now you get warning from compiler

Copy Function Deprecated


```
class R {
public:
  R(int x) : i{x} {}
  ~R() {}
  R& operator=(R const& rhs);
  int I() const { return i; }
  void I(int x) { i = x; }
private:
  int i;
};

int main() {
  R a1{10};
  R a2{a1}; // warning should be issued here
  // other stuff here
}
```

Future Topics ...

- Return value optimization
- Rvalue references
- Move semantics
- Rule of Five and Rule of Zero
- Smart pointers