COMP6771
Advanced C++ Programming

Week 11
Object Oriented Programming

2016

www.cse.unsw.edu.au/~cs6771

Covariants and Contravariants

Let us assume that Class B is a subtype of class A.

- Covariants: What should return types of overriding function be?
  - All member functions of B must return the same or a narrower set of types as A
  - The return type is said to be covariant
- Contravariants: What should parameter types of overriding function be?
  - All member functions of B must take the same or a broader set of arguments compared with the member functions of A
  - The argument type is said to be contravariant.
- Return types are covariant but argument types are contravariant

Overriding Virtuals in Derived Classes

- Virtual functions are inherited but typically overridden:

```
class A {
  virtual A f(int i, int j);
}
class B : public A {
  virtual B f(int i, int j);
}
```

- The overriding f must have the same signature, and the return type B must be A or a derived class of A (i.e., covariant).

Covariant Return Types

- Changing a method's return type when overriding it in a derived class

```
1  class Shape {
2  public:
3     virtual void draw() = 0;
4     virtual Shape* clone() const = 0;
5  };
6  class Circle : public Shape {
7     void draw() { /* draw a circle */ }
8     virtual Circle* clone() const { return new Circle(this*); }
9  };
10 class Square: public Shape {
11   void draw() { /* draw a square */
12   virtual Square * clone() const { return new Square(this*); }
13 };
```

- Benefit:

```
Circle c1;
Circle *c2 = c1.clone();
// down-casting required if clone had returned a shape*
Shape *s = c1.clone();  // ok: covariant
                        // will contravariant return types work?
```

Default Arguments and Virtuals

```
1  struct B {
2    virtual void foo(int i = 1) {
3      std::cout << "B: " << i;
4    }
5  };
6
7  struct D : public B {
8    void foo(int i = 2) {
9      std::cout <<  "D: " <<i;
10   }
11 };
12
13 int main(void) {
14   B *p = new D();
15   p->foo();
16 }
```

Output: D: 1

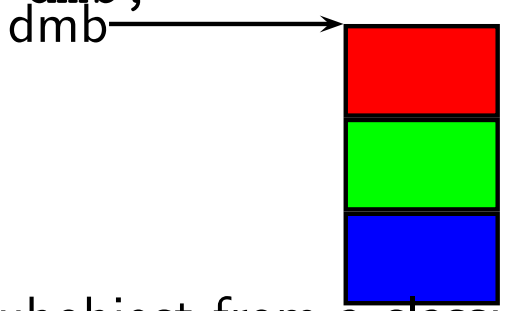
- Default arguments determined based on static type of p
- The function call resolved using dynamic type of p
- Why? Efficiency since the compiler doesn't need to generate code used to determine at run time which default to use

Initialisation and Destruction

- Initialisation order:

```
class Book {...};
class Disc_book: public Book {...};
class Bulk_Disc_book : public Disc_book {...};
Bulk_Disc_book dmb;
```

  - Subobjects



  - Within each subobject from a class:
    1. the data members initialised in declaration order
    2. the class' constructor is called

    The direct base constructor of a class is always called first

- Destruction: (the reverse)

```
1 for each sub-obj from the most to the least derived
2      call its destructor
3      Call its members' dtors in reverse declaration order
```

Member Initialisation List

- Design requirements of a derived class's constructor
  - Call an appropriate immediate base constructor
  - Initialise its own members
- Conventions:
  1. The base class's constructor is listed first
  2. The member constructors are in the order of declaration

```
class Disc_book : public Book {
  ...
  Disc_book(const std::string &book_isbn = "",
            double sales_price,
            std::size_t qty = 0,
            double dist_rate = 0.0)
            : Book(book_isbn, sales_price),
              min_qty(0), discount (0.0) { }
};
```

- If a base constructor is not called by programmer, the compiler will call the default one and scream if it does not exist

Virtual Destructors

- The destructor in Book is virtual

```
Disc_book *p = new Disc_book();
delete p; // dynamic binding
          // call the destructor in Disc_book
```

- If the destructor in Book is nonvirtual

```
Book *p = new Disc_book();
delete p; // static binding
          // call the destructor in Book
```

- The virtual destructor in Book must be defined since it will be called by the destructor in the least derived class

There are no virtual constructors since dynamic binding cannot be realised on an object that is not yet fully constructed.

No move operation synthesised once a destructor is defined

Move Operations vs. Virtual Destructors

- If a class defines a virtual destructor, then it doesn't get synthesised move operations. Neither do its derived classes.
- Must define your own if needed:

```
1 class Book {
2   Book(Book &&) = default;
3   Book& operator=Book(Book &&) = default;
4 }
```

- Use your own if the default is not suitable

Virtuals in Constructors and Destructors

- If a virtual member function is called insider a constructor or destructor, then the version invoked is the one defined for the type of the constructor or destructor itself

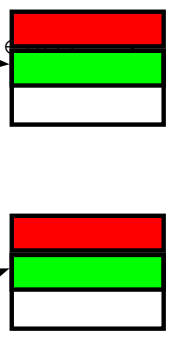
```
1  class A {
2  public:
3    virtual void f() { std::cout << "f in B" << std::endl; }
4  };
5
6  class B : public A {
7  public:
8    B() { f(); }
9    void f() { std::cout << "f in B" << std::endl; }
10   ~B() { f(); }
11 };
12
13 class C : public B {
14 public:
15   void f() { std::cout << "f in C" << std::endl; }
16 };
17
18 int main() {
19   C c;
20   A &a = c;
21   a.f();
22 }
```

What is the output?

Virtuals in Constructors and Destructors

- If a virtual member function is called insider a constructor or destructor, then the version invoked is the one defined for the type of the constructor or destructor itself

```
1  class A {
2  public:
3    virtual void f() { std::cout << "f in B" << std::e
4  };
5
6  class B : public A {
7  public:
8    B() { f(); }
9    void f() { std::cout << "f in B" << std::endl; }
10   ~B() { f(); }
11 };
12
13 class C : public B {
14 public:
15   void f() { std::cout << "f in C" << std::endl; }
16 };
17
18 int main() {
19   C c;
20   A &a = c;
21   a.f();
22 }
```



A constructed
B constructed

C destructed

```
OUTPUT
f in B
f in C
f in B
```

OO Implementation of a Stack

Stack Abstract Base Class (ABC):

```
1  class Stack {
2  public:
3    virtual ~Stack() { }; //virtual dtor
4    virtual void push (const int &item) = 0;
5    virtual void pop () = 0;
6    virtual int& top () = 0;
7    virtual const int& top () const = 0;
8    virtual bool empty () const = 0;
9    virtual bool full () const = 0;
10
11   void operator=(const Stack&) = delete;
12 };
```

Move operations can be provided as explained earlier in the course

Why Deleted Operator= in an ABC?

```
1 B_Stack s1;
2 UB_Stack s2;
3 foo(s1, s2);
4
5 void foo(Stack &s1, Stack &s2) {
6   s1 = s2;
7 }
```

- s1 = s2 makes no sense if s1 and s2 are different stack objects (perhaps for this particular application)
- Making operator= deleted or protected causes a compile-time error

A Bounded stack: B_stack.hpp

```
1  #ifndef B_STACK_HPP
2  #define B_STACK_HPP
3
4  #include "Stack.hpp"
5
6  class B_Stack : public Stack {
7  public:
8    B_Stack (unsigned size_ = 10);
9    B_Stack (const B_Stack &);
10   ~B_Stack ();
11   B_Stack& operator=(const B_Stack &);
12   virtual void push (const int &item);
13   virtual void pop ();
14   virtual int& top ();
15   virtual const int& top () const;
16   virtual bool empty () const;
17   virtual bool full () const;
18 private:
19   unsigned top_, size_;
20   int *stack_;
21 };
22
23 #endif
```

A Bounded stack: B_stack.cpp

```
1  #include "B_Stack.hpp"
2
3  B_Stack::B_Stack (unsigned s)
4  : top_{0}, size_{s}, stack_{new int[s]} { }
5
6  B_Stack::~B_Stack () { delete [] stack_; }
7
8  inline void B_Stack::push(const int &item) {
9     stack_[top_++] = item;
10 }
11
12 inline void B_Stack::pop() { --top_; }
13
14 inline int& B_Stack::top() { return stack_[top_-1]; }
15 inline const int& B_Stack::top() const
16                             { return stack_[top_-1]; }
17
18 inline bool B_Stack::empty() const {
19   return top_ == 0;
20 }
```

A Bounded stack: B_stack.cpp

```
1  inline bool B_Stack::full() const { return top_ == size_; }
2
3  B_Stack::B_Stack(const B_Stack &s)
4  :  top_(s.top_),
5     size_(s.size_),
6     stack_(new int[s.size_]) {
7     for (unsigned i = 0; i < s.size_; i++)
8       stack_[i] = s.stack_[i];
9  }
10
11 B_Stack& B_Stack::operator=(const B_Stack &s) {
12   if (this != &s) {
13     delete [] stack_;
14     top_ = s.top_;
15     size_ = s.size_;
16     stack_ = new int[s.size_];
17     for (unsigned i = 0; i < s.top_; i++)
18       stack_[i] = s.stack_[i];
19   }
20   return *this;
21 }
```

An Unbounded stack: UB_stack.hpp

```
1  #ifndef UB_Stack_HPP
2  #define UB_Stack_HPP
3
4  #include "Stack.hpp"
5
6  class UB_Stack : public Stack {
7  public:
8    UB_Stack () { head_ = nullptr; }
9    UB_Stack(const UB_Stack &);
10   ~UB_Stack ();
11   UB_Stack& operator=(const UB_Stack &s);
12   virtual void push (const int &item);
13   virtual void pop ();
14   virtual int& top ();
15   virtual const int& top () const;
16   virtual bool empty () const { return head_ == nullptr; }
17   virtual bool full () const { return false; };
18 private:
19   class Node;
20   void reverse (Node *);
21   Node *head_;
22 };
23
24 #endif
```

An Unbounded stack: UB_Stack.cpp

```
1  #include "UB_Stack.hpp"
2
3  class UB_Stack::Node {
4    friend class UB_Stack;
5    Node (int i, Node *n = 0) : item_{i}, next_{n} { }
6    ~Node() { delete next_; }
7    int item_;
8    Node *next_;
9  };
10
11 UB_Stack::UB_Stack(const UB_Stack &s) {
12   head_ = nullptr;
13   reverse(s.head_);
14 }
15
16 UB_Stack::~UB_Stack() {
17   delete head_;
18 }
19
20 UB_Stack& UB_Stack::operator=(const UB_Stack &s) {
21   if (this != &s) {
22     delete head_;
23     reverse(s.head_);
24     return *this;
25   }
26 }
```

An Unbounded stack: UB_stack.cpp

```
1  void UB_Stack::push(const int &item) {
2    Node *t = new Node (item, head_); head_ = t;
3  }
4
5  void UB_Stack::pop() {
6    Node *t = head_;
7    head_ = head_->next_;
8    t->next_ = nullptr;
9    delete t;
10 }
11
12 inline int& UB_Stack::top() {
13   return head_->item_;
14 }
15
16 inline const int& UB_Stack::top() const {
17   return head_->item_;
18 }
19
20 void UB_Stack::reverse(Node *h) {
21   if (h != nullptr) {
22     reverse(h->next_);
23     push(h->item_);
24   }
25 }
```

Client Code

```
1  #include "Stack.hpp"
2
3  void foo(Stack *stack) {
4    int i = 1927;
5    stack->push(6771);
6    stack->push(i);
7    stack->pop();
8    std::cout << s->top();
9    // ...
10 }
```

Using our abstract base class, it is possible to write code that does not depend on the stack implementation.

Create Some Stack Objects

- Of course, someone else has to create a concrete stack:

```
1  #include "B_Stack.hpp"
2  #include "UB_Stack.hpp"
3
4  extern void foo(stack *);
5
6  Stack* make_stack (bool use_B_Stack) {
7    if (use_B_Stack)
8       return new B_Stack;
9    else
10     return new UB_Stack;
11 }
12
13 int main(void) {
14   foo(make_stack (false));
15 }
```

- Needs recompilation if B_Stack/UB_Stack has changed

OO in C++

- Polymorphism/dynamic binding achieved with virtual functions
- A class with virtual functions (declared or inherited) is also said to be polymorphic since it exhibits multiple behaviours due to the multiple implementations of virtual functions in derived classes
- Combining ABCs and dynamic binding to separate an application's interface from its implementation

Copy Control

- The Big Three/Five:
  - (Virtual) Destructor
  - Copy constructor
  - Copy Assignment
  - Move constructor
  - Move Assignment
- Need only to provide these for your derived classes (since a base class has been provided with these by its designer
- In general, classes with pointer members need to define their own copy control to manage these members
- The semantics for the compiler-synthesised control-control: memberwise copy, move, assignment and destroy

Virtual Destructors

- Essential when you apply delete to pointers/references to base classes:

```
stack *p = new UB_Stack;
...
delete p;
```

- If `~stack` is nonvirtual, then static binding will cause `stack::~stack` to be invoked, resulting in memory leaks!
- Should generally declare your destructor virtual if the class is intended to be used as a base class
- Destructors cannot be pure virtual
- See Item 14, Meyers' Effective C++

Compiler-Generated (Default) Copy Constructor

- semantics: memberwise copy

```
class Derived : public Base {
  A1 o1
  A2 o2
  ...
}
```

- The default:

```
Derived::Derived (const Derived &d)
: call Base's copy constructor,
  call A1's copy constructor,
  call A2's copy constructor,
  ...
  { }
```

Compiler-Generated Copy Constructors for B_Stack

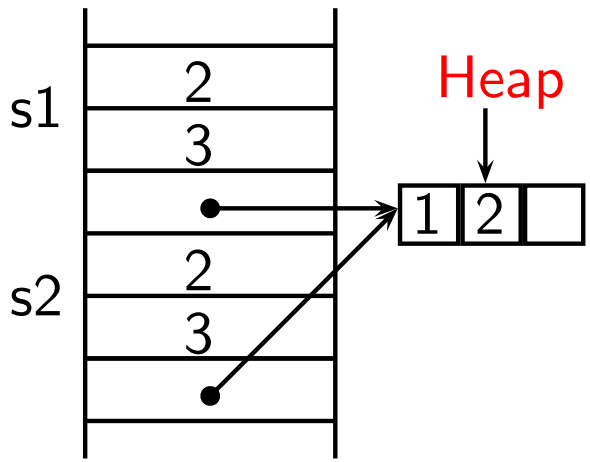
- Shallow copy:

```
B_Stack::B_Stack(const B_Stack &s) {
    top_ = s.top_;
    size_ = s.size_;
    stack_ = s.stack_;
}
```

- Example:

```
B_Stack s1(3);
s1.push(1);
s1.push(2);
B_Stack s2 = s1;
// same as B_Stack s2(s1);
```

Runtime Stack



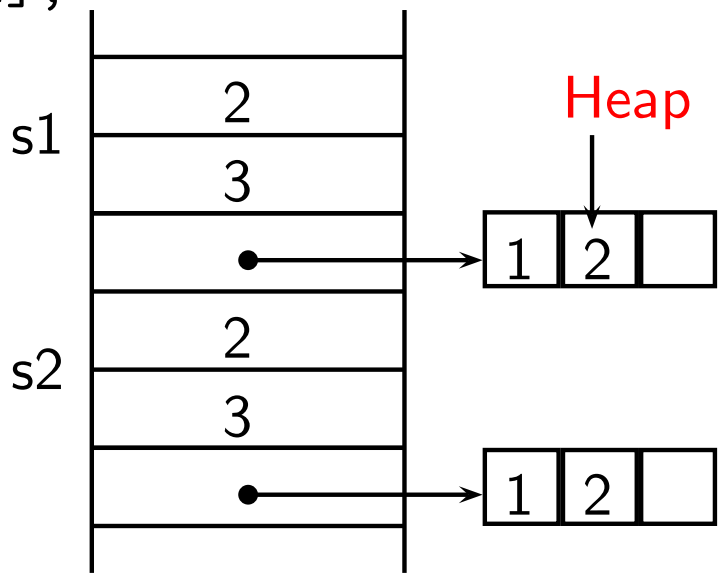
Our Copy Constructor (Value Semantics)

- Deep copy:

```
B_Stack::B_Stack(const B_Stack &s)
:  top_ (s.top_),
   size_(s.size_),
   stack_(new int[s.size_]) {
   for (unsigned i = 0; i < s.size_; i++)
     stack_[i] = s.stack_[i];
}
```

- Example:

```
B_Stack s1(3);
s1.push(1);
s1.push(2);
B_Stack s2 = s1;
```



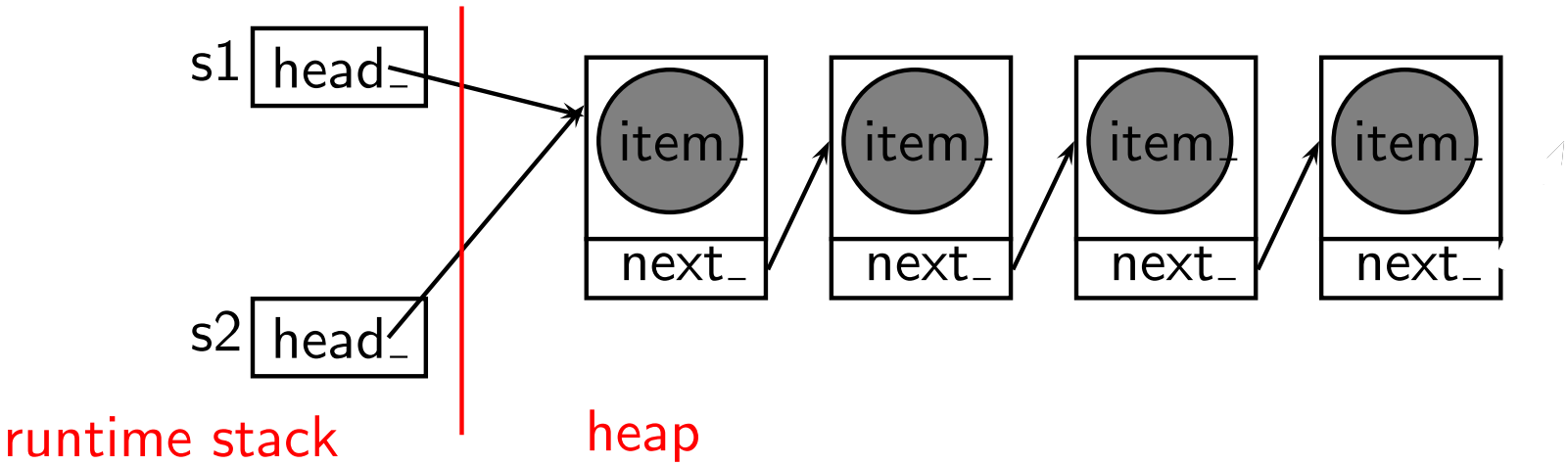
Copy Constructor for UB_Stack

- Shallow copy:

```
UB_Stack::UB_Stack(const UB_Stack &s)
: head_(s.head_) {}
```

- Example:

```
UB_Stack s2 = s1; // copy construction
```



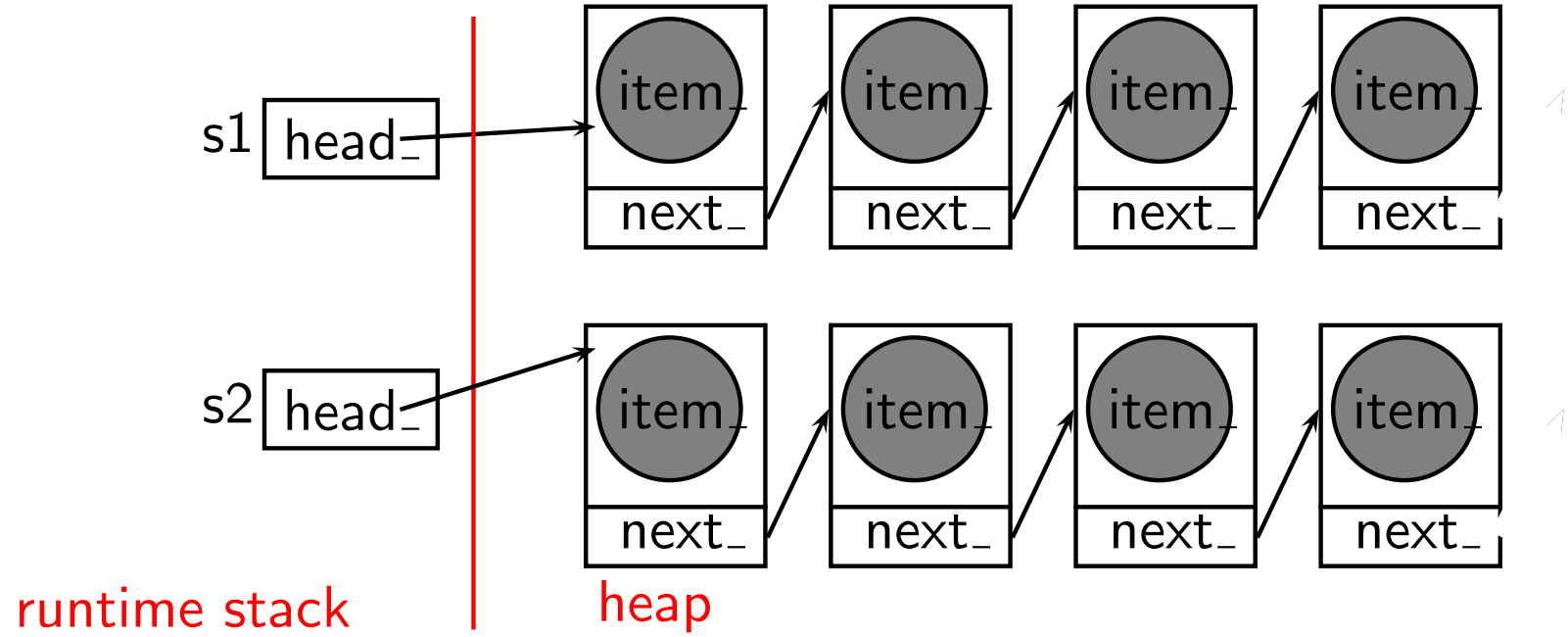
Our Copy Constructor for UB_stack

- Deep copy:

```
UB_Stack::UB_Stack(const UB_Stack &s) {
  head_ = nullptr; reverse(s.head_);
}
```

- Example:

```
UB_Stack s2 = s1; // copy construction
```



Compiler-Generated (Default) Assignment Operator

- semantics: memberwise assignment

```
class Derived : public Base {
  A1 o1
  A2 o2
  ...
}
```

- The default:

```
Derived::Derived (const Derived &d) {
 copy Base's assignment operator
 call A1's assignment operator
 call A2's assignment operator
 ...
}
```

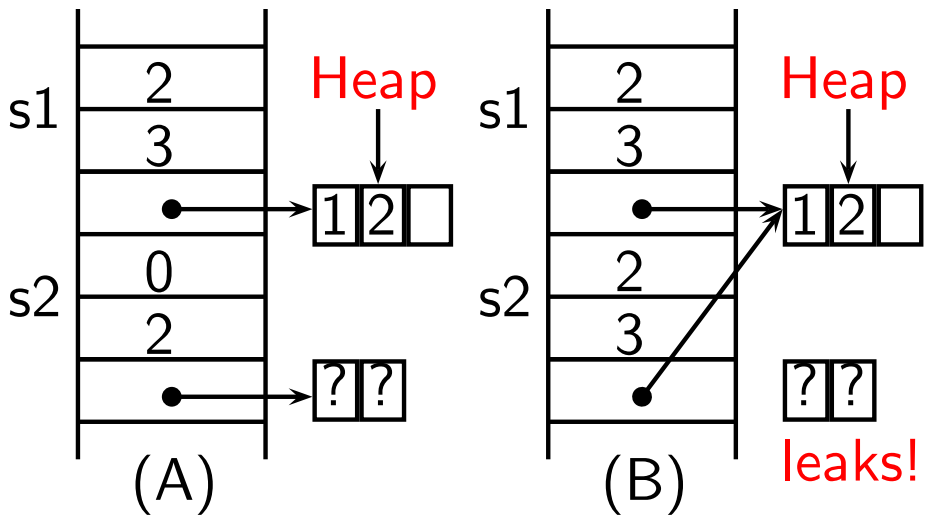
Compiler-Generated Operator= for B_stack

- Shallow copy:

```
B_Stack& operator=(const B_Stack &s) {
  if (this != &s) {
    top_ = s.top_;
    size_ = s.size_;
    stack_ = s.stack_;
  }
  return *this;
}
```

- Example:

```
B_Stack s1(3), s2(2);
s1.push(1);
s1.push(2);
   (A)
s2 = s1;
   (B)
```



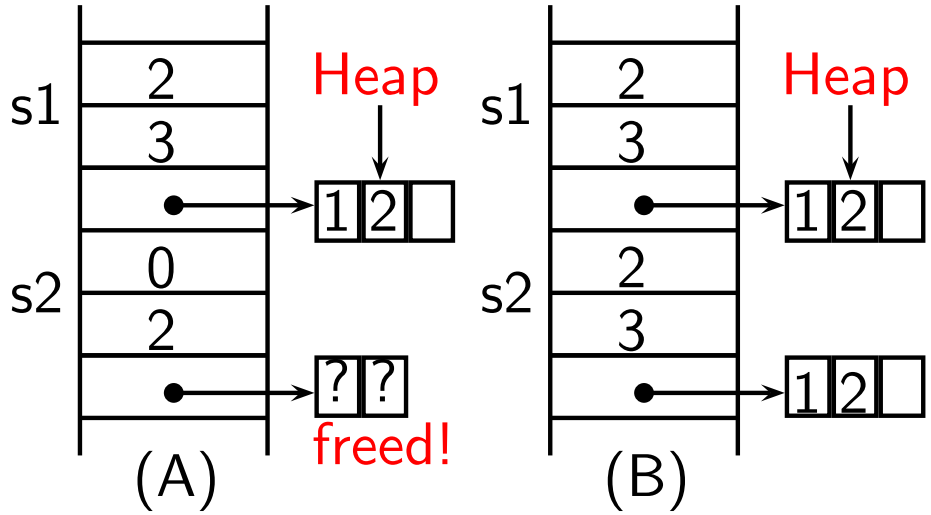
Our Operator= for B_stack

- Deep copy:

```
B_Stack& B_Stack::operator=(const B_Stack &s) {
  if (this != &s) {
    delete [] stack_;
    top_ = s.top_;
    size_ = s.size_;
    stack_ = new int[s.size_];
    for (unsigned i = 0; i < s.top_; i++)
      stack_[i] = s.stack_[i];
  }
  return *this;
}
}
```

- Example:

```
B_Stack s1(3), s2(2);
s1.push(1);
s1.push(2);
   (A)
s2 = s1;
   (B)
```



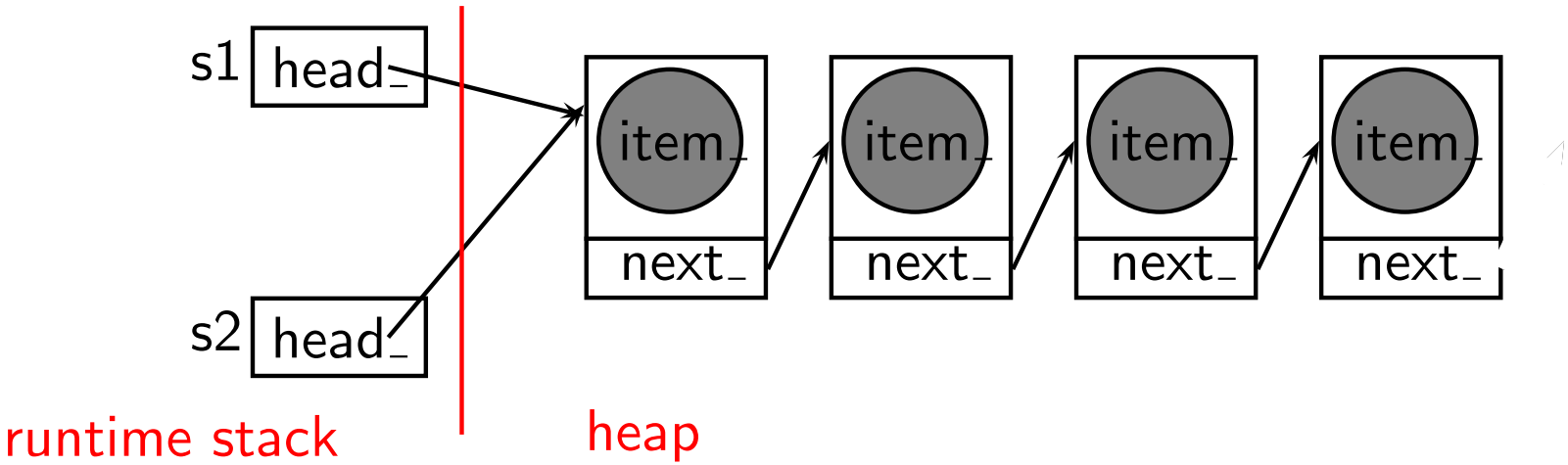
Compiler-Generated Operator= for UB_stack

- Shallow copy:

```
UB_Stack& UB_Stack::operator=(const UB_Stack &s) {
  head_ = s.head_;
  return *this;
}
```

- Example:

```
s2 = s1;
```



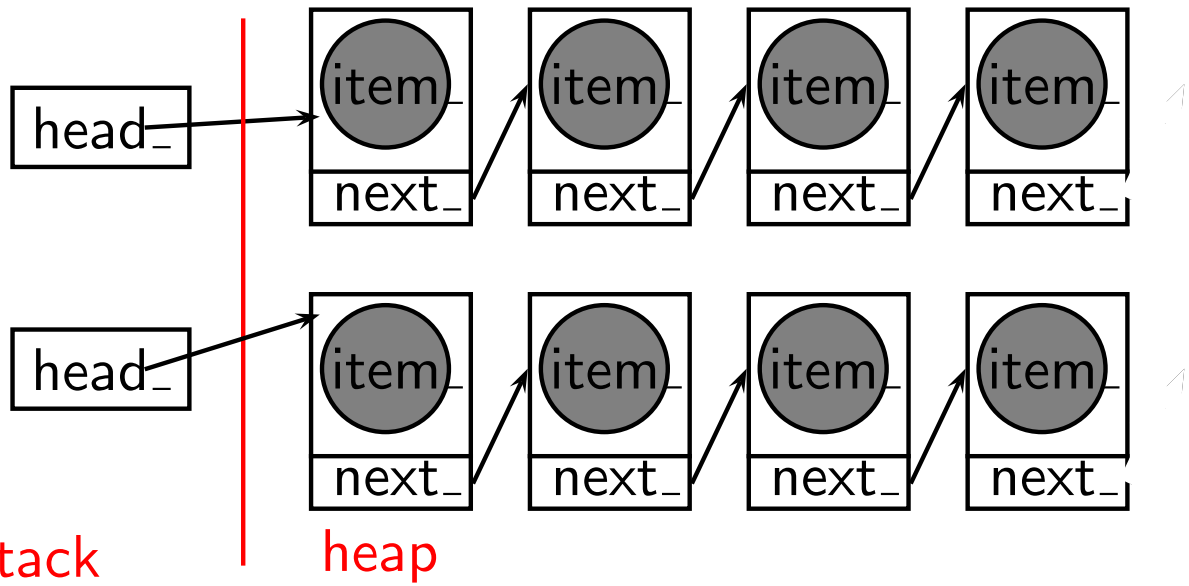
Our Operator= for UB_stack

- Deep copy:

```
UB_Stack& UB_Stack::operator=(const UB_Stack &s) {
  if (this != &s) {
    delete head_;
    reverse(s.head_);
  }
  return *this;
}
```

- Example:

```
s2 = s1;
```



Readings

- Chapter 15 of the text
- Chapter 12, Stroustrup's book (Derived Classes)